Choosing the Best Parallelization and Implementation Styles for Graph Analytics Codes: Lessons Learned from 1106 Programs

Yiqian Liu  
Department of Computer Science  
Texas State University  
San Marcos, TX, USA  
y_l120@txstate.edu

Avery VanAusdal  
Department of Computer Science  
Texas State University  
San Marcos, TX, USA  
arv107@txstate.edu

Noushin Azami  
Department of Computer Science  
Texas State University  
San Marcos, TX, USA  
noushin.azami@txstate.edu

Martin Burtscher  
Department of Computer Science  
Texas State University  
San Marcos, TX, USA  
burtscher@txstate.edu

ABSTRACT
Graph analytics has become a major workload in recent years. The underlying core algorithms tend to be irregular and data dependent, making them challenging to parallelize. Yet, these algorithms can be implemented and parallelized in many ways for CPUs and even more ways for GPUs. We took 6 key graph algorithms and created hundreds of CUDA, OpenMP, and parallel C++ versions of each of them, most of which have never been described or studied. To determine which parallelization and implementation styles work well and under what circumstances, we evaluated the resulting 1106 programs on 2 GPUs and 2 CPUs using 5 input graphs. Our results show which styles and combinations perform well and which ones should be avoided. We found that choosing the wrong implementation style can yield over a 10x slowdown on average. The worst combinations of styles can cost 6 orders of magnitude in performance.

CCS CONCEPTS
• Computing methodologies → Massively parallel algorithms; Parallel programming languages.

KEYWORDS
Graph analytics, parallelization and implementation styles

1 INTRODUCTION
With the rise of social networks, search engines, recommender systems, GPS navigators, and data science, graph algorithms for computing communities, shortest paths, frequent motifs, centrality, and so on have become an important workload. However, many of these algorithms exhibit irregular behavior, meaning the resulting control-flow and memory-access patterns are data dependent [11]. As a consequence, their behavior cannot be statically predicted and may change throughout the computation. This makes optimizing and especially parallelizing irregular codes difficult as the amount of parallelism depends on the input and can change dynamically.

Despite these challenges, there are numerous ways to parallelize irregular programs. In fact, the complexities due to their irregular nature create opportunities for dozens of implementation styles. In this paper, we study the hundreds of resulting combinations between parallelization and implementation styles and evaluate how well they perform on various devices and inputs.

An example of different parallelization styles is using thread, warp, or block granularity in GPU codes [48]. Each granularity has benefits and drawbacks. Thread-based parallelization is typically the easiest to implement but may not perform well in the presence of load imbalance. Switching to warps (a group of 32 threads) requires more complex synchronization but enables the use of fast warp-level primitives. Using blocks (a group of up to 1024 threads) further complicates synchronization but can better exploit the fast "shared memory", a software-managed L1 data cache.

An example of different implementation styles is push versus pull, which is common in both CPU and GPU codes [6]. When updating the values stored in the vertices of a graph, a push-style implementation will use the value of a vertex \( v \) to compute a new value with which to update a neighboring vertex. In contrast, a pull-style implementation will use the value of a neighbor to compute a new value with which to update vertex \( v \).

We differentiate code optimizations from parallelization/implementation styles as follows. Parallelization and implementation styles are broadly applicable to many graph algorithms. In contrast, code optimizations tend to be specific to individual programs or a particular implementation of an algorithm. Due to this difference,
programmers are more likely to be able to apply a given parallelization or implementation style when writing a new graph algorithm than they are to be able to apply a given code optimization.

Most of the parallelization and implementation styles we study (cf. Section 2) are orthogonal and can be combined. This yields a large number of unique implementations for a given graph algorithm. In this manner, we have written 100s of versions of 6 key irregular graph algorithms for both CPUs and GPUs. The resulting source codes are available in the Indigo2 benchmark suite [31, 32].

Several widely-used benchmark suites with parallel implementations of irregular graph algorithms already exist, including Lonestar [25] with 14 parallel implementations of 11 graph algorithms and Gardenia [45], an extended version of GAP [7], with 126 parallel implementations of 14 graph algorithms. These suites provide a range of interesting algorithms and inputs to study. However, none of them are designed to provide a large variety of each algorithm, nor do they include enough variations to perform an in-depth evaluation of parallelization and implementation styles.

Our Indigo2 suite fills this gap. We estimate that well over 90% of the code versions it includes have never been studied before. Our performance analysis of these codes reveals that parallelization and implementation styles are an important factor to be taken into account when writing parallel graph codes. This paper makes the following main contributions.

- We describe 13 largely orthogonal parallelization and implementation styles for CPUs and GPUs.
- We combine these styles in hundreds of meaningful ways and apply them to 6 graph analytics problems.
- We evaluate the over 1000 resulting codes on 2 GPUs, 2 CPUs, and 5 input graphs from different domains.
- We provide guidelines for programmers on which styles and combinations to use and under what conditions.

The rest of this paper is organized as follows. Section 2 describes the parallelization and implementation styles we consider. Section 3 summarises related work. Section 4 presents the experimental methodology, including the codes, inputs, and systems used for the measurements. Sections 5 evaluates and discusses the performance of the various parallelization and implementation styles. Section 6 summarizes our findings and draws conclusions.

2 PARALLELIZATION AND IMPLEMENTATION STYLES

The following subsections describe the styles we investigated. We studied many parallel graph codes and the related literature to extract these styles. Hence, we believe we captured many of the frequently used styles, but more styles almost certainly exist.

We illustrate each style on the example of the Bellman-Ford single-source-shortest-path (SSSP) algorithm [12]. Given an undirected weighted graph with no negative cycles and a source vertex, the algorithm computes the shortest distance (i.e., the sum of the edge weights) from the source to every vertex. It starts by setting the distance of the source vertex to 0 and all other distances to $\infty$. For each edge $(u, v)$, a new distance is calculated (i.e., $\text{distance}[u] + \text{weight}(u, v)$) in each iteration. The distance of $v$ is updated if the new distance is shorter. These edge relaxation operations repeat until the algorithm converges.

We wrote our graph codes using three parallel programming models: CUDA, OpenMP, and C++ threads. CUDA programs operate at multiple levels of parallelism. 32 contiguous threads form a warp and execute the same instruction in the same cycle (or are disabled). Sets of up to 32 warps (up to 1024 threads) form a block, and the blocks are grouped into a grid. CUDA provides built-in variables for the thread and block indices as well as the block and grid dimensions. These values are often combined by computing $\text{threadIdx}.x + \text{blockIdx}.x + \text{blockDim}.x$ to form a global index for assigning work to each thread, which we call “gidx” in our codes.

OpenMP is based on pragma compiler directives. Each such directive consists of a name followed by optional clauses. For example, a clause can specify the scheduling to be used or a reduction operation. In Listing 1b below, it selects dynamic scheduling.

C++11 supports multithreading in the standard library. It includes built-in classes and functions for threading, atomic, mutual exclusion, and more. For instance, std::this_thread::get_id() returns the unique thread ID. It enables different scheduling policies (e.g., blocked and cyclic) to be implemented.

2.1 Vertex-based vs. edge-based

Since graphs consist of vertices and edges, we can iterate across either the vertices or the edges [47]. Listing 1a shows vertex-based code where every thread processes a different vertex $v$ and iterates over all neighbors $u$. Listing 1b shows edge-based code that assigns a different vertex $v = (v, u)$ to each thread.

The algorithm to be implemented and the graph representation (e.g., CSR format [21]) typically determine which style is preferable. For instance, if the graph is represented by a set of adjacency lists, it is more natural to employ the vertex-based style. To streamline the discussion, we use this style in the following subsections.

(a) Vertex-based

```c++
if (v < nodes) {
    beg = nbr_idx[v];
    end = nbr_idx[v + 1];
    for (i = beg; i < end; i++) {
        u = nbr_list[i];
        . . .
    }
}
```

(b) Edge-based

```c++
if (e < edges) {
    v = src_list[e];
    u = dst_list[e];
    . . .
}
```

Listing 1: Vertex- and edge-based computations

2.2 Topology-driven vs. data-driven

This style describes two ways in which a program can process the data-structure elements that need processing [41]. If all elements are processed, the computation is topology-driven. In contrast, a data-driven computation would only process the elements that likely need to be updated, which are usually stored in a worklist. For example, topology-driven SSSP applies the relaxation function to all vertices of the graph in each iteration as shown in Listing 2a. Data-driven SSSP only applies the relaxation function to the vertices in the worklist as outlined in Listing 2b. Those vertices were placed in the worklist because their distance changed in the prior iteration.

The topology-driven style tends to yield more parallelism and is easier to implement. The data-driven style is more work efficient and, therefore, often results in better performance, especially for iterative algorithms that operate on high-diameter graphs.
writes the new value if it meets a certain condition. For example, in SSSP, the vertex distance is updated if the new distance is shorter. Since other threads may be updating the same distance value in parallel, simply reading and then independently writing, as is done in the read-write style outlined in Listing 5a, only works in some situations. In particular, the updates must be monotonic and the algorithm must be resilient to temporary priority inversions [35]. The read-modify-write style shown in Listing 5b is more general as it does not suffer from this problem, but it requires an atomic read-modify-write operation, which may be slower and hamper parallelism. Note that, throughout this paper, we assume the shared data values (e.g., the distances) to be scalars and assume load and store instructions to atomically read and write these values [10].

2.6 Non-deterministic vs. deterministic

The unpredictability of thread timing can cause (internal) non-determinism in some parallel codes [9]. For example, in the SSSP code shown in Listing 6a, one thread will read dist [v] while multiple other threads may write that same memory location. Depending on when the read takes place relative to the writes, it will load a different value, resulting in the computation of a different new distance with which the neighbors will be updated. This is not a problem in SSSP as any non-final distance value will be overwritten in a later iteration. Hence, the ultimate result of the computation is deterministic, but it is unpredictable after how many iterations the code will converge. Note that we only study programs in this paper where the final result is deterministic.

Using two arrays, one that is only read (dist1[ ]) and another that is updated (dist2[ ]), as shown in Listing 6b, makes the code internally deterministic. However, in this approach, the computation can no longer take advantage of results generated in the same iteration, which may slow down the execution. On the upside, the deterministic code will always require the same number of iterations for a given input, which can simplify debugging [4].

2.7 Persistent vs. non-persistent

This variation only applies to GPU codes. The persistent style uses as many threads as the GPU can concurrently schedule on its SMs [23]. Hence, a thread may have to process multiple vertices. The non-persistent style launches at least as many threads as the input has vertices and assigns no more than one vertex to each thread. For graphs where the number of vertices exceeds the number of threads that can concurrently run on the SMs, the GPU will automatically schedule batches of threads until all threads have executed. The persistent style is a little more complex to implement but may improve performance in cases where common subexpressions can be precomputed or common data preloaded and then reused.
This variation only applies to CUDA codes. To avoid data races, 

2.8 Thread vs. warp vs. block

This variation only applies to GPU codes. It refers to the granularity 

vertices in power-law graphs [2]. However, it is typically not useful 

memory order to achieve good performance. 

may need to figure out a safe but narrower scope and a more relaxed 

atomic operation. The scope determines whether the operation is 

atomic operations before. The memory order restricts how the 

GPU code [1]. The corresponding 'CudaAtomic' solution shown 

Standard Library that can be used both in and between CPU and 

libcu++

However, these atomics cannot be used in the host code running on 

the CPU. As a remedy, CUDA recently introduced libc++, a C++ 

Standard Library that can be used both in and between CPU and 

GPU code [1]. The corresponding 'CudaAtomic' solution shown 

Listing 7: Persistent and non-persistent threads

Listing 8: Thread, warp, and block parallelization

Listing 10: Different reductions in CUDA

Listing 9: Atomic and CudaAtomic

Listing 11: Different reductions in OpenMP

2.9 Atomic vs. CudaAtomic

This variation only applies to CUDA codes. To avoid data races, 

CUDA provides a set of atomic functions. For example, Listing 9a 

employs an atomicMin() to atomically update a memory location. 

However, these atoms cannot be used in the host code running on 

the CPU. As a remedy, CUDA recently introduced libc++, a C++ 

Standard Library that can be used both in and between CPU and 

GPU code [1]. The corresponding 'CudaAtomic' solution shown 

Listing 9b requires a data type as well as optional memory-

ordering and scope specifications, which were not available for 

atomic operations before. The memory order restricts how the 

surrounding memory accesses can be ordered with respect to the 

atomic operation. The scope determines whether the operation is 

atomic at the block, grid, or system level (including host code).

CudaAtomic’s default scope and memory ordering are chosen 

to ensure program correctness in the most cases, which is also the 

slowest setting. Hence, when using CudaAtomic, the programmer 

may need to figure out a safe but narrower scope and a more relaxed 

memory order to achieve good performance.

(a) Persistent
threads = blockDim.x \times gridDim.x; 
for (v = gdx; v < nodes; v += threads) ... 

(b) Non-persistent
v = gdx; 
if (v < nodes) ... 

Listing 7: Persistent and non-persistent threads

2.10 Reduction styles

Reductions combine multiple values into a single value using a 

binary associative operator [30]. For example, multiple threads may 

need to add the partial sums they computed to a global sum. 

2.10.1 Global-add vs. block-add vs. reduction-add. We employ three 

reduction styles in our GPU codes. The first approach performs 

atomic operations that directly update the shared global variable as 

shown in Listing 10a. The second approach makes use of the 

faster block-level atomics. All threads of a block first compute a 

block-local solution in the 'shared memory', and only one thread 

updates the global solution as shown in Listing 10b. This minimizes 

the number of slower global atomics. The third approach utilizes 

not only shared-memory buffers for local results but also warp-

level primitives to quickly perform warp and block reductions as 

outlined in Listing 10c. This implementation is more complex but 

tends to be faster as it avoids most memory accesses.

(a) Global-add
atomicAdd(&ctr, val);

(b) Block-add
atomicAdd_block(&ctr, val);
warp_ctr = warp_reduction(val);

atomicAdd(&ctr, warp_ctr);

(c) Reduction-add
atomicAdd(&ctr, block_ctr);

Listing 10: Different reductions in CUDA

2.10.2 Atomic-reduction vs. critical-reduction vs. clause-reduction. We also employ three reduction styles in our CPU codes. OpenMP 

and C++ provide atomic operations as well, making it possible for 

each thread to atomically update a shared variable as shown in Listing 

11a. They also provide mutex support, allowing the programmer 

to update the shared variable in a critical section as shown in Listing 

11b. Finally, OpenMP provides a reduction clause as shown in 

Listing 11c that can be used in certain cases. Using a critical section 

typically results in substantial overhead and poor performance, but 

it is the most general of the three approaches.

(a) Atomic reduction
#pragma omp parallel for 
for (i = beg; i < end; i++) {

#pragma omp atomic
sum += val;

}

(b) Critical reduction
#pragma omp parallel for reduction(+, sum)
for (i = beg; i < end; i++) {

sum += val;

}

(c) Clause reduction
#pragma omp parallel for reduction(+, sum)
for (i = beg; i < end; i++) {

sum += val;

}

Listing 11: Different reductions in OpenMP
2.11 Default scheduling vs. dynamic scheduling

OpenMP can automatically parallelize certain for loops with a “parallel for” directive. By default, shown in Listing 12a, it statically assigns each thread a chunk of iterations. In contrast, the dynamic schedule in Listing 12b assigns the loop iterations to the threads at runtime. This improves the load balance but incurs overhead.

(a) Default scheduling

```
#pragma omp parallel for
for (v = beg; v < end; v++) {
    ...
}
```

(b) Dynamic scheduling

```
#pragma omp parallel for schedule(dynamic)
for (v = beg; v < end; v++) {
    ...
}
```

Listing 12: Default and dynamic loop scheduling

2.12 Blocked vs. cyclic

When parallelizing the iterations of a for loop, a blocked schedule assigns a contiguous chunk of iterations to each thread, as shown in Listing 13a. If the iterations’ running times correlate with their loop index, a block distribution can lead to load imbalance. The cyclic schedule in Listing 13b assigns the iterations in a round-robin fashion to the threads, which improves the load balance in this scenario. A blocked schedule usually has better data locality in CPUs because one thread accesses contiguous memory addresses. However, a cyclic schedule has better data locality in GPUs because of coalesced memory accesses (i.e., combining multiple memory accesses into a single memory transaction).

(a) Blocked scheduling

```
beg = tid * nodes / threads;
end = (tid + 1) * nodes / threads;
for (v = beg; v < end; v += threads) {
    ...
}
```

(b) Cyclic scheduling

```
for (v = tid; v < nodes; v += threads) {
    ...
}
```

Listing 13: Blocked and cyclic scheduling

3 RELATED WORK

A plethora of prior publications on parallelizing irregular graph codes exist. Many of them discuss and evaluate at least some implementation styles, but no systematic study of a large number of styles exists. Becchi et al. propose workload consolidation schemes [44] and different parallelization templates [28] to increase the GPU utilization of programs with nested parallelism. Wang et al. characterize dynamically formed parallelism and evaluate codes designed to exploit them [43]. Nasre et al. present morph algorithms and provide insights into how other morph algorithms can be efficiently implemented for GPUs [39]. Similar to these works, we also study general styles that are applicable to a wide range of algorithms.

Most if not all of the parallelization and implementation styles we investigate have been described before. For example, Hong et al. [24] propose a warp-centric programming method to improve the performance of applications with heavily imbalanced workloads. Nasre et al. study data-driven and topology-driven implementations to understand the tradeoffs [38] and investigate high-level methods to eliminate atomics in irregular programs [36]. Pingali et al. discuss different styles to process nodes (e.g., topology-driven and data-driven) and operators that modify the graph (e.g., morphs and local computations) [41]. Our work takes many of these styles and combines them in hundreds of new ways.

3.1 Benchmark suites of irregular programs

There are many benchmark suites of irregular graph codes with programs that are optimized for CPUs or GPUs. Lonestar [26], which contains 14 parallel C++ and CUDA implementations of 11 irregular algorithms, mostly aims to include fast implementations of as many domains as possible. Pannothia [13] includes 8 graph codes implemented in OpenCL from diverse domains. It was designed to show that irregular codes can be parallelized and implemented on GPUs. GraphBIG [34] selects representative data structures, workloads, and data sets from 21 real-world use cases. GAPBS [7] consists of OpenMP implementations of 6 important graph algorithms. This benchmark suite aims to standardize the evaluation of graph processing. GBBS [17] comprises scalable, provably-efficient implementations of over 20 fundamental graph problems for shared-memory multicore machines. Having primarily been designed with performance and diversity in mind, these suites tend to be based on highly optimized codes from various domains. They generally do not include many different styles of the same algorithm.

Indigo [33] is the largest related benchmark suite and the predecessor of the Indigo2 suite presented in this paper. It contains thousands of parallel codes representing 6 common data-access patterns that occur frequently in irregular graph computations. However, they are all microbenchmarks that do not compute anything useful, i.e., they are not complete graph algorithms. GARDENIA [45] includes emerging graph-processing workloads. It is an extended version of GAPBS and comprises 126 parallel implementations of 11 irregular graph algorithms written in OpenMP, CUDA and OpenCL. With over one hundred implementations, it covers the most parallelization and implementation styles of any prior benchmark suite. However, the included implementations are specifically tuned to optimize a given algorithm. The focus is not on generic implementation styles that are applicable to a large body of graph codes. In fact, many publications describe ways to optimize the performance of specific parallel graph codes. For example, focusing just on GPUs, there is work presenting high-performance implementations of breadth-first search [18], single-source shortest path [15], minimum spanning trees [42], community detection [29], strongly connected components [5], graph coloring [3], triangle counting [22], and PageRank [19] to name a few.

In contrast to most of the related work, our benchmark suite is designed for comparing parallelization and implementation styles that broadly apply to graph algorithms. Hence, our focus is on providing a wide diversity of styles. This is why our suite includes between 90 and 256 versions of each of 6 graph algorithms.

4 EXPERIMENTAL METHODOLOGY

4.1 Codes

We selected the 6 graph problems shown in Table 1 for our study. We chose them because they are the most common graph codes in prior benchmark suites (e.g., Lonestar [26] and GBBS [17]). Since not all implementation styles are applicable to every problem, Table 2 lists the included styles.

Since combining the applicable styles yields hundreds of variations, we automated the code-generation process and use configuration files to select the desired versions (i.e., a subset of the codes) as we have done in the predecessor Indigo suite [33].
Table 1: Graph problems used in our study

<table>
<thead>
<tr>
<th>Category</th>
<th>Name and abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connectivity</td>
<td>Connected Components (CC)</td>
</tr>
<tr>
<td>Covering</td>
<td>Maximal Independent Set (MIS)</td>
</tr>
<tr>
<td>Eigenvector</td>
<td>PageRank (PR)</td>
</tr>
<tr>
<td>Substructure</td>
<td>Triangle Counting (TC)</td>
</tr>
<tr>
<td>Shortest path</td>
<td>Breadth-First Search (BFS), Single Source Shortest Path (SSSP)</td>
</tr>
</tbody>
</table>

Table 2: Included implementation styles

<table>
<thead>
<tr>
<th>Styles</th>
<th>CC</th>
<th>MIS</th>
<th>PR</th>
<th>TC</th>
<th>BFS</th>
<th>SSSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex-based, edge-based</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Topology-driven, data-driven</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Duplicates in WL, no duplicates in WL</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push, pull</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read-write, read-modify-write</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deterministic, non-deterministic</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Persistent, non-persistent</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread, warp, block</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomic, CudaAtomic</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global add, block-add, reduction-add</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomic_add, critical_add, clause_red</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default scheduling, dynamic scheduling</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blocked, cycle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Number of code versions (32-bit data type)

<table>
<thead>
<tr>
<th>Language</th>
<th>CC</th>
<th>MIS</th>
<th>PR</th>
<th>TC</th>
<th>BFS</th>
<th>SSSP</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA</td>
<td>168</td>
<td>112</td>
<td>54</td>
<td>72</td>
<td>180</td>
<td>168</td>
<td>754</td>
</tr>
<tr>
<td>OpenMP</td>
<td>36</td>
<td>36</td>
<td>18</td>
<td>12</td>
<td>38</td>
<td>36</td>
<td>176</td>
</tr>
<tr>
<td>C++ threads</td>
<td>36</td>
<td>36</td>
<td>18</td>
<td>12</td>
<td>38</td>
<td>36</td>
<td>176</td>
</tr>
</tbody>
</table>

To keep the running times and the number of code versions manageable, we tested our suite only with 32-bit data types (int, float). However, the 64-bit data-type versions are included in Indigo2. Table 3 shows the breakdown of the 1106 CUDA, OpenMP, and C++ programs we evaluated. Each code verifies its computed solution by comparing it to the solution of a simple serial algorithm.

4.2 Inputs

Since the control-flow and memory-access patterns of irregular programs are input dependent, we selected 5 graphs of various types, origins, sizes, and degree-distributions as inputs. The graph names and other information about them are shown in Tables 4 and 5. We picked these graph sizes to keep the running times reasonable. In many cases, we compute ratios of the throughputs to investigate how the different styles affect performance. To visualize the thousands of resulting ratios, we use a box plot to show the distribution and other pertinent information. It recursively divides the dataset into halves and presents different quantile values. Thicker boxes indicate more data points in the given range. For example, in Figure 1a, the thickest box represents the middle 50% of the ratios.

4.3 Hardware

We present results from 2 systems, i.e., 2 CPUs and 2 GPUs. System 1 has a 3.5 GHz Ryzen Threadripper 2950X CPU with 16 hyper-threaded cores, a 32 MB L3 cache, and 64 GB of main memory. It houses a 1.2 GHz TITAN V GPU with 12 GB of global memory, a 4.5 MB L2 cache, and 5120 processing elements distributed over 80 streaming multiprocessors (SMs). System 2 has dual 2.9 GHz Xeon Gold 6226R CPUs with a total of 32 hyperthreaded cores, two 22 MB L3 caches, and 64 GB of main memory. It houses a 1.74 GHz RTX 3090 GPU with 24 GB of global memory, a 6 MB L2 cache, and 10,496 processing elements distributed over 82 SMs. We use 16 threads for the OpenMP and C++ codes on the first system and 32 threads on the second system. We do not employ hyperthreading as it tends to hurt or not improve the performance of our codes.

4.4 Software

On both machines, the operating system is Fedora Linux 34. We used GCC 11.3.1 with the "-O3 -fopenmp" flags to compile the OpenMP codes and the "-O3 -pthread" flags for the C++ codes on both systems. On System 1, we compiled the GPU programs using NVCC 11.7 with the "-arch=sm_70" flag. On System 2, we compiled the GPU programs using NVCC 11.6 with the "-arch=sm_86" flag.

4.5 Metrics

We ran each of our 1106 programs on the 5 input graphs, resulting in a total of 5530 tests. If a program takes less than 10 minutes for a given input, we ran it 9 times and use the median for computing the throughputs. For the few longer-running codes, we only measured one run. To improve readability, we report the throughputs in giga-edges per second. This is the number of edges in the input graph divided by the runtime and then divided by one billion.

In many cases, we compute ratios of the throughputs to investigate how the different styles affect performance. To visualize the thousands of resulting ratios, we use a box plot to show the distribution and other pertinent information. It recursively divides the dataset into halves and presents different quantile values. Thicker boxes indicate more data points in the given range. For example, in Figure 1a, the thickest box represents the middle 50% of the ratios.

We are not using pthreads per se, but this flag is required by the C++ threading library.
and the line indicates the median. Any outliers that differ substantially from the other ratios are plotted as circles. The dashed blue line in the chart indicates a ratio of 1.0 on the y-axis.

5 RESULTS
Each of the following subsections compares the performance of two or three alternative styles while keeping the other styles fixed. For example, assume we only had the push vs. pull and thread vs. warp styles. To contrast the push and pull styles, we would compare the throughput of thread-level push with that of thread-level pull, warp-level push with warp-level pull, and block-level push with block-level pull. To visualize the results, we divide the throughputs to compute pairwise ratios where applicable. A ratio above 1.0 means the first-named style is faster. We start our discussion with the styles that make the largest performance difference.

5.1 Atomic and CudaAtomic
This subsection compares the conventional Atomic to the recently released CudaAtomic style. Figure 1 summarizes the resulting 1750 throughput ratios obtained on the our two GPUs. No results are included for PR because CudaAtomic does not yet support floats.

The ratio is above 1.0 in almost all cases, implying that the Atomic versions are generally faster than the CudaAtomic versions. In fact, the median ratio is around 10 on the RTX 3090 (Figure 1a) and roughly 100 on the Titan V (Figure 1b) for CC, MIS, BFS, and SSSP, which means Atomic is one to two orders of magnitude faster. The largest ratios we have observed are nearly 100 on the RTX 3090 and well over 1000 on the Titan V. The CudaAtomic throughputs, but not the Atomic throughputs, are much lower on the older Titan V, which is why the ratios are so much higher. On both GPUs, the ratios are markedly lower for TC because TC only contains an atomic add operation whereas the other programs make frequent use of CudaAtomic’s `load()` and `store()` operations.

Overall, CudaAtomic is a nascent library that is easy to use but can slow down programs drastically when employed with the default settings. The programmer needs to explicitly specify the relaxed memory order and the device scope to get similar performance to the Atomic versions (results not shown). As the CudaAtomic codes are so slow, we exclude them from the following subsections to narrow down the ranges of the presented throughput ratios.

5.2 Vertex-based and edge-based
This subsection compares two ways of iterating over graphs, namely vertex- and edge-based codes. Figure 2a summarizes the corresponding throughput ratios obtained on our GPUs. Figure 2b shows the same set of results but from our CPUs. We separately highlight the ratios of the thread-based subset of TC codes in Figure 2c (i.e., excluding the warp- and block-based versions).

5.3 Topology-driven and data-driven
In this subsection, we compare the topology- and data-driven styles. Since the data-driven style can be implemented both with and without allowing duplicates on the worklist, but there is no counterpart for the topology-driven style, we separately compare topology- to data-driven with duplicates in Section 5.3.1 and topology- to data-driven without duplicates in Section 5.3.2.

5.3.1 Topology-driven vs. data-driven with duplicates on the worklist
Figure 3a shows the GPU throughput ratios of topology-driven CC, BFS and SSSP over their data-driven versions with duplicates. Figure 3b shows similar results for the OpenMP and Figure 3c for the C++ programs. MIS only works with the no-duplicates style, and TC and PR do not have data-driven versions.

On the GPUs, all 6 graph codes have cases where a vertex-based code achieves better performance than its edge-based counterpart. BFS, CC, TC, and SSSP also have cases where an edge-based version performs better than vertex-based. With the median ratio being approximately 1, we cannot say in general that either edge- or vertex-based performs better. However, MIS achieves a higher throughput with the vertex-based style on all of our implementations. Its median is close to a factor of 10 in favor of vertex-based. This is because the MIS code typically only visits a few neighbors per vertex, making the vertex-based approach quite load balanced.

The majority of TC codes are faster on the GPUs with the edge-based style. In fact, TC is up to 100 times faster than vertex-based on the socLiveJournal input. The results from Figure 3c highlight that edge-based is essentially always better on the thread-level TC implementations. This is expected because warp- and block-level parallelization mostly helps alleviate load imbalance between vertices with different numbers of neighbors. Edge-based codes do not suffer from such load imbalance because each edge takes roughly the same amount of time to process.

On the CPUs, MIS again receives the most benefit from the vertex-based style. Interestingly, all other codes now have a median that is significantly above 1, suggesting that CPU codes tend to prefer vertex-based implementations, which is not the case for GPUs.

Overall, the speedup (or slowdown) between vertex- and edge-based computations can reach two orders of magnitude on both GPUs and CPUs. We find that some codes like MIS generally fare better with a vertex-based implementation. Many codes seem to prefer a vertex-based style on the CPU but not on the GPU. In particular, thread-based codes running on the GPU exhibit a preference for an edge-based implementation, which enhances load balancing.
The GPU codes prefer the data-driven over the topology-driven implementation. The same is true for OpenMP but not for C++. The big discrepancy in the CPU behavior is because of max and min operations, which must be implemented with slow critical sections in OpenMP but can be done with fast atomics in C++.

5.3.2 Topology-driven vs. data-driven without duplicates on the worklist. Figure 4a compares the topology-driven codes with their data-driven versions without duplicates on the worklist. Figures 4b and 4c show the corresponding results for the CPU codes.

The median ratio of topology-driven over data-driven is less than 1 for all measured GPU codes and above 1 for the C++ codes. It is below 1 for the CC, BFS, and SSSP OpenMP codes. Interestingly, the MIS OpenMP code prefers the topology-driven style. These OpenMP codes have a larger ratio range than any other styles we investigated. In some cases, topology-driven is over 100 times faster. In other cases, data-driven is over a million times faster, especially on high-diameter graphs where a topology-driven implementation may perform huge amounts of useless work in each iteration.

Based on the results from both subsections, we find that the data-driven style, with and without duplicates in the worklist, tends to be the better choice for GPUs. However, all measured programs on all tested devices exhibit some cases where the topology-driven style yields high speedups (e.g., low-diameter graphs) and other cases where the data-driven style is much faster (e.g., high-diameter graphs). Using a topology- or data-driven implementation can result in two orders of magnitude difference in performance for both GPUs and CPUs. We conclude that the graph type in particular should be taken into account when selecting between these two styles.

5.4 Push and pull

This subsection analyzes the effect of the data-flow direction, i.e., push and pull, on the performance. Figures 5a, 5b, and 5c summarize the throughput ratios of push-style over pull-style for 5 graph problems. TC does not support this style.

The median ratios are consistently above 1 for CC, MIS, BFS, and SSSP on all devices. In contrast, the PR medians are a little below 1. Note that the push-style CC, MIS, BFS, and SSSP codes include non-deterministic versions where no synchronization is used, whereas the PR push-style codes only include deterministic versions. Furthermore, the push-style codes combine better with data-driven versions as they simplify the population of the worklist and tend to place fewer useless items on the worklist. We do not have data-driven codes for PR, which may be another contributing factor as to why PR does not follow the ratio trend of the other codes. We conclude that, typically, push is preferential to pull. In extreme cases, the push style can yield 100× speedup in CUDA and C++ programs and 1000× speedup in OpenMP codes.

5.5 Read-write and read-modify-write

We show the throughput ratios of CC, BFS, and SSSP with read-write over their read-modify-write counterparts running on the GPUs in Figure 6a and on the CPUs in Figures 6b and 6c.

The read-write style is slightly faster than read-modify-write in most cases on both GPUs and CPUs. The speedup of read-write can reach up to a factor of 3 on GPUs and over 1000 on CPUs. This difference might be because atomics tend to be slower on CPUs (where they go through the shared L3 cache) than on GPUs (where they go through the shared L2 cache). Since the read-modify-write style applies to more algorithms and typically performs nearly as well, we believe it to be a good choice in most cases.

5.6 Deterministic and non-deterministic

This section evaluates the performance differences of deterministic and internally non-deterministic codes. Figures 7a, 7b, and 7c show the resulting throughput ratios of the CUDA, OpenMP, and C++ CC, MIS, PR, BFS, and SSSP codes.

Aside from PR, the internally non-deterministic style yields higher throughputs for all codes with just a few exceptions. PR behaves differently because it does not include the non-deterministic style for half of its implementations (i.e., the push-style codes). The
Choosing the Best Parallelization and Implementation Styles for Graph Analytics Codes: Lessons Learned from 1106 Programs SC ’23, November 12–17, 2023, Denver, CO, USA

5.7 Persistent and non-persistent
This section analyzes the throughput of persistent and non-persistent codes. As this variation only applies to GPUs, Figure 8 only shows results for the CUDA versions.

Figure 8: Throughput ratios of persistent over non-persistent

Most of the ratios and the medians are very close to 1. This is because the advantage of the persistent style (e.g., precomputing sub-expressions) cannot be exploited in our codes. Hence, the more complex persistent style is only recommended when there is significant work that can be performed once and then used for multiple vertices or edges, such as pre-loading the shared memory with data.

5.8 Thread, warp, and block
Since there are 3 parallelization styles in this category, presenting pairwise ratios would be complicated. Instead, we plot the throughputs of each style to visualize and compare the performance. Figures 9a and 9b plot the throughput of the three styles on the RTX 3090 for the NY roadmap and the soc-LiveJournal graph, respectively. The throughput of thread-based parallelization is red, warp-based is blue, and block-based is yellow.

The throughputs of thread-, warp-, and block-based parallelization are highly correlated with the degree distribution of the input graph. The thread-based codes provide the highest performance on graphs with low degrees and relatively uniform degree distributions such as the NY roadmap. For scale-free graphs such as social networks, where a few vertices have a very high degree, the warp-based implementations yield the highest throughputs. Block-based parallelization tends to be the slowest because none of our inputs have a significant number of vertices with a degree of over 512 (see Table 5) to match the number of threads per block.

5.9 Global-add, block-add, and reduction-add
In this subsection, we compare three methods of performing sum reductions. Figure 10 shows the results for the GPUs. We again present throughputs instead of ratios because we are comparing three styles. The red dots show the throughputs for global-add, the blue dots for block-add, and the yellow dots for reduction-add. This variation only applies to the TC and PR codes.

TC achieves a higher throughput than PR because the PR codes perform many more sum reductions. The block-add style tends to be the slowest, and the reduction-add style is the fastest for PR. This indicates that the block-scope atomicAdd() operations cannot offset the overhead of the global-scope atomicAdd(). Since the reduction-add uses warp primitives for warp and block reduction, it is expected to perform well and is the recommended style.

5.10 Atomic-, critical-, and clause-reduction
This subsection repeats for CPUs what the previous subsection did for GPUs, except the three reduction styles are different. Again, only the PR and TC codes have such reductions. Figure 11 shows the results, where red dots refer to atomic-reduction, blue dots to critical-reduction, and yellow dots to clause-reduction.

Similar to the GPU results above, TC achieves a higher throughput for the three versions than PR. Expectedly, the use of a critical section yields the lowest performance on both codes, and the reduction clause achieves the highest throughput of the three versions.
We recommend programmers avoid using critical sections and even atomics if a reduction clause is supported by the library.

5.11 Default and dynamic scheduling

This subsection analyzes the effects of scheduling loop iterations using OpenMP pragmas. We revert back to showing throughput ratios of the default schedule over the dynamic schedule in Figure 12.

There is almost no difference in throughput for PR, BFS, and SSSP. However, MIS is always faster with the default schedule. CC and TC also prefer the default schedule but have cases that perform better with a dynamic schedule. Overall, we found default scheduling to yield between two orders of magnitude speedup and $10^8$ slowdown in our experiments. The dynamic schedule provides load balancing but has runtime overhead. Since there is not much load imbalance for most of our inputs, load balancing is unnecessary, and the overhead makes the dynamic schedule slower.

5.12 Blocked and cyclic scheduling

This subsection investigates the type of schedule in C++ parallel codes. Figure 13 shows the throughput ratios of a blocked schedule over a cyclic schedule for all inputs on both CPUs.

The choice of schedule does not affect performance much on CC, MIS, BFS, and SSSP. However, PR prefers a blocked schedule whereas TC prefers a cyclic schedule (i.e., 75% of the ratios are below 1). Selecting blocked vs. cyclic scheduling causes between a $10^8$ speedup and a $10^9$ slowdown in the C++ programs. Thus, it is important to select an appropriate scheduling style, but which schedule is best depends on the loop characteristics.

5.13 Correlation with graph properties

As the behavior of our codes is input-dependent, we correlated the throughputs with various graph properties, including the size, average and maximum degree, percentage of the vertices having degrees less than 32 and 512, and the diameter. We found no correlation greater than 0.5 or less than $-0.5$, indicating that these properties do not significantly affect performance. The highest correlation of 0.44 is between warp-based parallelization and the average degree, showing that the warp-based style tends to yield better performance with higher-degree inputs, which is expected.

5.14 Best-performing styles

Figure 14 categorizes the best-performing code versions over all inputs, algorithms, and programming models along 6 pairs of dimensions. We selected these 6 pair-dimensions because they are applicable to all three studied programming models. Each row shows, for a particular model, what percentage of the best-performing codes uses the style of that column. For example, the CUDA row presents the best-performing CUDA codes (i.e., the highest throughput code for every algorithm and input from 30 code versions), and the vertex-based column shows that 80% thereof are vertex-based. Percentages above 50 are shaded red and below 50 are shaded blue.

There are 3 columns in Figure 14 that are entirely red, indicating that the vertex-based, push, and non-deterministic styles dominate the best performing codes across different programming models. Additionally, C++ Threads strongly prefers topology-driven while the other two models prefer data-driven for best performance.

5.15 Style combinations

Since our CUDA programs include the most versions and their style preferences are similar to OpenMP, we further analyze which style combines better with which style in Figure 15. Every matrix entry presents the performance improvement/loss (i.e., larger/smaller than 1.0) when combining the style of the row with that of the column. A “warmer” shading indicates a higher performance improvement. For example, the 1.63 in the vertex-based row and push-style column is the throughput ratio of the median of all vertex styles that include push over the median of all vertex styles that do not include push. The resulting matrix is asymmetric because the baseline (i.e., the median throughputs of style$_x$ without style$_y$) differs for each entry.

The push, non-deterministic, and non-persistent columns are mostly red, which suggests combining them with any style may improve performance. The warp column is also red because warp-based processing yields higher throughputs for high-degree graphs. Vertex/edge-based and topo/data-driven are mixed and dup/non-dup and read-write/read-modify-write are mostly blue, indicating that there is no general preference for these pairs.

Figure 14: Percentage of each style in best performing codes

Figure 15: Ratio of the median throughputs of style$_x$ with style$_y$ over style$_x$ without style$_y$ for the CUDA codes
5.16 Programming guidelines

Based on the results of all prior sections, we arrived at the following general guidelines and recommendations.

- High-degree inputs prefer warp-based parallelization in CUDA. Otherwise, the style preference is not significantly correlated with the input graph properties.
- We recommend using the non-deterministic and push styles for CUDA, OpenMP, and C++.
- When possible, avoid using default CudaAtomic in GPU codes and critical sections in OpenMP and C++ programs.
- Whether to use a vertex- or edge-based implementation depends on the algorithm.
- Edge-based tends to yield better performance when combined with a topology-driven approach in CUDA.
- As using persistent threads in CUDA rarely improves performance, we recommend a non-persistent implementation.
- Since default and blocked scheduling are relatively safe for CPUs but dynamic and cyclic scheduling may improve the performance, we suggest using default/blocked during development and then testing other schedules.
- C++ prefers the topology-driven style because the worklist overhead often cannot offset the work-efficiency benefit.

5.17 Comparison with third-party codes

To demonstrate that our unoptimized codes yield reasonable performance, we compare them to the optimized Lonestar [26] CPU and Gardenia [46] GPU codes. We refer to these Lonestar and Gardenia codes as “baseline” codes. For this comparison, we use the best-performing style (the style that has the highest average throughput over all inputs) for each of our codes and plot the speedups over the baseline codes in Figure 16. Speedups above 1 (i.e., the dashed blue line) mean our codes are faster. Figure 16a does not show MIS results since MIS is not included in Gardenia [46].

Our PR and TC codes outperform the CPU baselines but are slower on the GPUs because the Gardenia codes include an optimization that removes redundant edges. The performance of CC is on par with the baselines across the different devices and programming models. Our BFS codes are faster on the GPUs and similar to the baseline on the CPUs. Lastly, our SSSP codes are generally slower. This is because both Lonestar and Gardenia include worklist optimizations. Gardenia employs two extra arrays that make the code as efficient as the data-driven approach but without the overhead of maintaining a worklist. Lonestar combines the data-driven approach with a priority scheduler that processes the vertices in ascending distance to reduce the total amount of work.

Table 6 lists the average speedup of the best-performing style over the baseline for each algorithm. For example, the “1.97” in the CUDA row and BFS column means our BFS CUDA codes are 1.97× faster on average (i.e., geometric mean). The right-most column presents the geometric mean for each programming model.

Overall, we find that, even though our codes do not include optimizations, they still yield reasonable performance. The optimized baselines do not outperform our codes in many cases, indicating that choosing the right implementation style is at least as important as incorporating program-specific code optimizations.

### Table 6: Average speedup over baseline codes

<table>
<thead>
<tr>
<th>Language</th>
<th>BFS</th>
<th>SSSP</th>
<th>CC</th>
<th>MIS</th>
<th>PR</th>
<th>TC</th>
<th>Geomean</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA</td>
<td>1.97</td>
<td>0.40</td>
<td>1.11</td>
<td>N/A</td>
<td>0.45</td>
<td>0.43</td>
<td>0.70</td>
</tr>
<tr>
<td>OpenMP</td>
<td>0.90</td>
<td>0.10</td>
<td>0.89</td>
<td>6.55</td>
<td>2.86</td>
<td>5.11</td>
<td>1.54</td>
</tr>
<tr>
<td>C++ threads</td>
<td>1.14</td>
<td>0.07</td>
<td>0.51</td>
<td>21.14</td>
<td>12.47</td>
<td>3.04</td>
<td>1.80</td>
</tr>
</tbody>
</table>

6 SUMMARY AND CONCLUSIONS

We study 13 sets of parallelization and implementation styles for irregular graph codes (cf. Table 2). Each set includes 2 or 3 alternatives. Since the styles are largely orthogonal, they can be combined in many unique ways. We applied them to 6 graph analytics problems to create the 2212 codes in the Indigo2 benchmark suite [31, 32].

We evaluate half of these codes (the 32-bit data-type programs) on 2 GPUs and 2 CPUs using 5 graphs from various domains. The results show that selecting the right parallelization/implementation styles is key, especially on GPUs. For example, choosing the wrong style can yield a 10× slowdown on average. The worst combinations of styles can cost 6 orders of magnitude in performance.

Our findings can serve as guidelines to help programmers write efficient parallel code. For example, we found that the push, modify-write, and clause-reduction styles are typically preferential to their alternatives. Programmers should avoid using critical sections and CudaAtomics with the default settings. The deterministic and global/block-add styles make it easier to write and debug code but hurt performance somewhat. The best way to process the graph (vertex-based or edge-based), decide what to compute (topology-driven or data-driven), and choose a granularity (thread, warp, or block) depends on the code as well as the input graph’s degree distribution and diameter. We show that scheduling can significantly affect performance on CPUs, but the default and blocked schedules are relatively safe choices. We hope the release of our Indigo2 benchmark suite will open up research opportunities to study additional aspects of parallelization and implementation styles.

ACKNOWLEDGMENTS

This work has been supported in part by the National Science Foundation under Award Number 1955367 and by an equipment donation from NVIDIA Corporation. We thank Ganesh Gopalakrishnan, John Jacobson, and the anonymous reviewers for their help and feedback to improve this paper.
