SAPPHIRE: A TOOL FOR TEACHING PARALLEL PROGRAMMING IN HUNDREDS OF DIFFERENT WAYS

Y. Liu, N. Azami, A. VanAusdal, M. Burtscher

Texas State University (UNITED STATES)

Abstract

Parallel programming has become an essential module in computer science education. However, most courses and textbooks only provide limited coverage of how to parallelize modern workloads. To fill this gap, we created Sapphire, a suite of seven graph analytics algorithms that we parallelized using hundreds of different styles. The suite includes over 2000 CUDA, OpenMP, and C++ codes as well as diverse inputs to elicit various behaviors from these parallel programs. Moreover, for each code, it contains annotated versions with intentionally planted bugs.

Sapphire enables educators to systematically teach how to parallelize graph algorithms, how to track down and eliminate common parallelization bugs, and to study the effects that different implementation styles and inputs have on performance. It also makes it easy to create many demos, hands-on exercises, and assignments. Thus, Sapphire can help students gain a deep understanding of how to develop efficient parallel programs on the example of interesting and relevant workloads.

Keywords: Parallel algorithms, parallel programming, graph analytics, implementation styles, education.

1 INTRODUCTION

Multi-core hardware is now ubiquitous, including in smartphones and tablets. However, much of our software is still serial, meaning it cannot leverage the performance and energy-efficiency benefits of the hardware parallelism. This disconnect underscores the importance of teaching parallel programming.

Many courses and textbooks on parallel programming already exist. Most of them use regular programs [1] with predictable memory accesses and control flow, such as dense array and matrix operations [2]–[10]. These codes are historically important and make it easy to cover basic parallelization techniques. However, based on our experience, students tend to find them unengaging or even boring. Moreover, they may never need to parallelize such codes in practice as efficient parallel libraries exist. This is why we recommend teaching parallel programming on more fascinating and relevant codes in at least the later parts of the course (or textbook), as we have done with great success in our courses.

For instance, graph analytics (Section 2) is a rapidly growing and already widely used domain with many interesting algorithms for processing social networks [11], analyzing medical data [12], building recommendation systems [13], and so on. Many of these codes are irregular [1], meaning they exhibit input-dependent and dynamically changing parallelism, control flow, and memory-access patterns. Teaching parallel programming on such examples is more forward looking and provides opportunities for covering a much wider variety of parallelization and implementation styles than is possible with regular codes. Learning about these styles is important as they can affect performance by orders of magnitude.

This paper presents Sapphire, an educational tool for our parallel programming courses. It can automatically apply widely-used implementation styles (Section 4) to a set of graph algorithms, yielding thousands of variations. It supports CUDA, OpenMP, and C++ threading. The resulting codes are correct and resemble implementations that a programmer might write. Additionally, it contains special code versions with intentionally planted bugs, such as data races and out-of-bound memory accesses. Moreover, Sapphire comes with real-world inputs and graph generators to produce countless synthetic inputs for running these input-dependent codes and studying their performance.

Our tool enables teachers to systematically explain general methods for parallelizing programs, show a wide range of source-code examples, discuss and measure the advantages and disadvantages of each style, demonstrate the effects of parallel programming bugs, and analyze how to combine styles to produce efficient implementations (Section 4). Sapphire also provides interesting assignments to help the students understand and apply this knowledge (Section 5). The Sapphire suite is open sourced under the name Indigo3 at https://github.com/burtscher/Indigo3Suite/.

2 BACKGROUND

Breadth-First Search (BFS) is perhaps the simplest graph traversal algorithm that is used in many applications. It labels all vertices with the shortest distance (in number of edges) from a given source vertex in the graph. Section 4 describes different parallelization styles on the example of BFS.

2.1 Graph analytics example

Fig. 1 shows the BFS algorithm. Using the graph from Fig. 2 as input and vertex 0 as the source, Table 1 shows the BFS computation step by step.

Algorithm 1 Parallel breadth-first search

Input: Graph $G = (V, E)$ and source vertex s

```
1:  for all vertices v ∈ V do
2:      if v = s then
3:          dist[v] ← 0
4:      else
5:          dist[v] ← ∞
6:      end if
7:  end for
8:  updated ← true
9:  while updated do
10:     updated ← false
11:     for all vertices v ∈ V do
12:         for all neighbors n ∈ adj[v] do
13:             if dist[n] > dist[v] + 1 then
14:                 dist[n] ← dist[v] + 1
15:                 updated ← true
16:             end if
17:         end for
18:     end for
19: end while
```

Output: Vertex is labeled with the shortest distance from s

Figure 1. Algorithm for parallel BFS

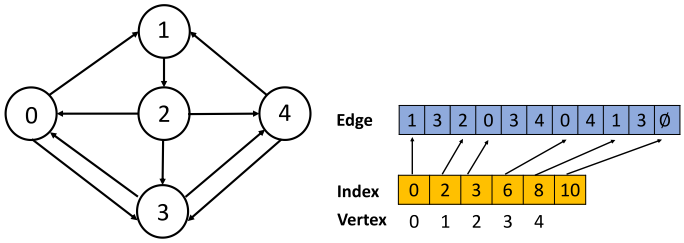

Figure 2. Example graph (left) and its CSR representation (right)

Table 1. Distance values computed in each step of BFS on the example graph

Vertex	Init	Iter1	Iter2	Iter3
0	0	0	0	0
1	∞	1	1	1
2	∞	∞	2	2
3	∞	1	1	1
4	∞	∞	2	2

The algorithm starts by initializing the distance of the source to 0 and all other distances to ∞. In the first iteration, every active vertex v (i.e., whose distance is not ∞) calculates a new distance (i.e., dist[v]+1). The new distance for vertices 1 and 3 is 1, which is smaller than their current distances, so they are updated, as shown in the Iter1 column of Table 1. Similarly, in the second iteration, vertices 0, 1, and 3 calculate new distances to their neighbors and find shorter distances for vertices 2 and 4. The next iteration is the final iteration because no new shorter distances are found.

2.2 Programming models

Sapphire currently supports three parallel programming models, namely CUDA, OpenMP, and C++ threads. CUDA supports multiple levels (e.g., thread, warp, and block) of parallelism. It provides built-in variables for accessing the thread and block indices as well as the block and grid dimensions. A unique global index for assigning work to each thread can be calculated by these values. We use “gidx” (e.g., in Fig. 3) to indicate this index. OpenMP is based on compiler directives, which are expressed by pragmas that may contain optional clauses. These clauses can specify various options. For instance, in Fig. 3, Examples 9a and 9b select different scheduling strategies. C++11 introduced built-in classes and functions in the standard library to support multithreading. These classes and functions can be used for threading, atomics, mutual exclusion, and more. For instance, std::this_thread::get_id() returns the unique thread ID. As shown in Examples 10a and 10b in Fig 4, the variable “tid” refers to the thread ID, enabling the implementation of different schedules.

2.3 Compressed sparse row graph format

The Compressed Sparse Row (CSR) format is one of the most widely used graph representations [14]. It is based on two dense arrays: an edge array and an index array. The edge array holds the concatenated adjacency lists of all vertices. The index array holds the starting position (index) of each adjacency list plus an extra element indicating the end of the last list. Fig. 2 shows an example graph and its CSR representation. All Sapphire codes are based on CSR, enabling users to provide their own inputs.

3 RELATED WORK

3.1 Existing textbooks

Textbooks on parallel programming generally focus on regular programs. In fact, most either do not include any irregular programs [4], [9] or only briefly introduce them by providing simple examples (e.g., tree traversal) [3], [5]–[7], [10]. Only a few textbooks [2], [8], [15] include chapters that illustrate how to parallelize graph algorithms, but none of them discuss different implementation styles. Moreover, surprisingly few parallel-programming textbooks cover debugging.

3.2 Parallel benchmark suites

Many suites with parallel codes exist. They target a variety of algorithms, application domains, programming languages, etc. For example, PARSEC [16], Rodinia [17], SHOC [18], Parboil [19], and Chai [20] consist of mostly regular parallel codes. Lonestar [21], Pannotia [22], GraphBIG [23], GAPBS [24], GARDENIA [25], and GBBS [26] contain irregular graph algorithms similar to those in Sapphire. However, none of these suites are particularly suitable for teaching purposes as they only contain a few codes and inputs, few or no labeled variations, no buggy codes, and the codes tend to be quite long. In contrast, Sapphire consists of short codes, where the main computation fits on a slide, and comprises many inputs and labeled variations of each base algorithm (including buggy implementations) to perform systematic studies. It contains 2276 bug-free codes and tens of thousands of buggy codes.

DataRaceBench [27], RMARaceBench [28], and Indigo [29] include buggy kernels. Among these three suites, only Indigo contains a large number of codes. However, they are all microbenchmarks, that is, code patterns that do not compute anything useful. In contrast, Sapphire includes full-fledged graph algorithms that are more useful for teaching and cover more bug types.

3.3 Automatic code generation

The source code annotation and variation in CREST [30], DLBENCH [31], and Indigo2 [32] inspired the code generation process in our suite. Sapphire is a superset of Indigo2 as it includes more graph

algorithms, covers more parallelization approaches, and contains buggy codes. These labelled buggy codes enable the teaching of how to find and eliminate parallelism bugs.

4 PARALLELIZATION AND IMPLEMENTATION STYLES

This section explains the various styles. Figs. 3 and 4 illustrate how each style affects the BFS code. The key differences are highlighted in red. The various styles exist because they have different pros and cons. Which style is preferable depends on the situation.

```
v = gidx;
if (v < nodes) {
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg; i < end; i++) {
    n = nbr_list[i];
    if (dist[n] > dist[v] + 1) {
      dist[n] = dist[v] + 1;
      go_again = true;
} } }
```
1a Vertex-based

```
e = gidx;
if (e < edges) {
  v = src_list[e];
  n = dst_list[e];
  if (dist[n] > dist[v] + 1) {
    dist[n] = dist[v] + 1;
    go_again = true;
} }
```
1b Edge-based

```
v = gidx;
if (v < nodes) {
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg; i < end; i++) {
    n = nbr_list[i];
    if (dist[n] > dist[v] + 1) {
      dist[n] = dist[v] + 1;
      go_again = true;
} } }
```
2a Topology-driven

```
idx = gidx;
if (idx < wl1size) {
  v = wl1[idx];
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg; i < end; i++) {
    n = nbr_list[i];
    if (dist[n] > dist[v] + 1) {
      dist[n] = dist[v] + 1;
      wl2[atomicAdd(wl2size, 1)] = n;
} } }
```
3a Duplicates in worklist

```
v = gidx;
if (v < nodes) {
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg; i < end; i++) {
    n = nbr_list[i];
    if (dist[n] > dist[v] + 1) {
      dist[n] = dist[v] + 1;
      go_again = true;
} } }
```
4a Push

```
v = gidx;
if (v < nodes) {
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg; i < end; i++) {
    n = nbr_list[i];
    if (dist[n] > dist[v] + 1) {
      dist[n] = dist[v] + 1;
      go_again = true;
} } }
```
5a Read-write

```
idx = gidx;
if (idx < wl1size) {
  v = wl1[idx];
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg; i < end; i++) {
    n = nbr_list[i];
    if (dist[n] > dist[v] + 1) {
      dist[n] = dist[v] + 1;
      wl2[atomicAdd(wl2size, 1)] = n;
} } }
```
2b Data-driven

```
idx = gidx;
if (idx < wl1size) {
  v = wl1[idx];
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg; i < end; i++) {
    n = nbr_list[i];
    if (dist[n] > dist[v] + 1) {
      dist[n] = dist[v] + 1;
      if (atomicMax(stat[v], iter) != iter) {
        wl2[atomicAdd(wl2size, 1)] = n;
} } } }
```
3b No duplicates in worklist

```
v = gidx;
if (v < nodes) {
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg; i < end; i++) {
    n = nbr_list[i];
    if (dist[v] > dist[n] + 1) {
      dist[v] = dist[n] + 1;
      go_again = true;
} } }
```
4b Pull

```
v = gidx;
if (v < nodes) {
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg; i < end; i++) {
    n = nbr_list[i];
    newd = dist[v] + 1;
    if (atomicMin(dist[v], newd) > newd) {
      go_again = true;
} } }
```
5b Read-modify-write

```
v = gidx;
if (v < nodes) {
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg; i < end; i++) {
    n = nbr_list[i];
    if (dist[n] > dist[v] + 1) {
      dist[n] = dist[v] + 1;
      go_again = true;
} } }
```
6a Non-deterministic

```
v = gidx;
if (v < nodes) {
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg; i < end; i++) {
    n = nbr_list[i];
    if (dist[n] > dist[v] + 1) {
      dist_new[n] = dist[v] + 1;
      go_again = true;
} } }
```
6b Deterministic

```
threads = blockDim.x * gridDim.x;
for (v = gidx; v < nodes; v += threads) {
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg; i < end; i++) {
    n = nbr_list[i];
    if (dist[n] > dist[v] + 1) {
      dist[n] = dist[v] + 1;
      go_again = true;
} } }
```
7a Persistent

```
v = gidx;
if (v < nodes) {
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg; i < end; i++) {
    n = nbr_list[i];
    if (dist[n] > dist[v] + 1) {
      dist[n] = dist[v] + 1;
      go_again = true;
} } }
```
7b Non-persistent

```
int dist[nodes];
v = gidx;
if (v < nodes) {
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg; i < end; i++) {
    n = nbr_list[i];
    newd = dist[v] + 1;
    if (atomicMin(dist[v], newd) > newd) {
      go_again = true;
} } }
```
8a Atomic

```
cuda::atomic<int> dist[nodes];
v = gidx;
if (v < nodes) {
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg; i < end; i++) {
    n = nbr_list[i];
    newd = dist[v] + 1;
    if (dist[v].fetch_min(newd) > newd) {
      go_again = true;
} } }
```
8a CudaAtomic

```
#pragma omp parallel for
for (v = 0; v < nodes; v++) { … }
```
9a Default scheduling

```
#pragma omp parallel for \
schedule (dynamic)
for (v = 0; v < nodes; v++) { … }
```
9b Dynamic scheduling

```
beg = tid * nodes / threads;
end =  (tid + 1) * nodes / threads;
for (v = beg; v < end; v++) { … }
```
10a Blocked scheduling

```
for (int v = tid; v < nodes; v += threads) {…}
```
10b Cyclic scheduling

Figure 3. BFS example codes using various general and specific styles

4.1 General styles

Vertex-based vs. edge-based. Graphs can be processed by iterating over their vertices or their edges [33]. Example 1a in Fig. 3 shows that every thread processes a different vertex v based on the unique global thread index (gidx) and iterates over all neighbors. Example 1b shows edge-based code that assigns a different edge to each thread.

Topology-driven vs. data-driven. The topology-driven [34] approach in Example 2a simply processes all elements. In contrast, the data-driven approach in Example 2b only processes the elements that likely need to be updated, which are stored in a worklist (wl).

Duplicates in worklist vs. no duplicates in worklist. Data-driven implementations can choose whether or not to allow duplicate elements on the worklist [35]. In Example 3a, each thread can push a vertex onto the worklist regardless of whether the worklist already contains it. In Example 3b, where duplicates are not allowed, the threads only add a vertex to the worklist if it is not already present.

Push vs. pull. The data flow in programs that update vertex data can be either push-based, where data is pushed from a vertex to its neighbors, or pull-based, where data is pulled from the neighbors to the vertex [36]. In push-style BFS, illustrated in Example 4a, a thread reads the vertex distance, adds 1, and updates the neighbor if the new distance is shorter. In pull-style BFS, shown in Example 4b, the thread reads the distance of the neighbor, adds 1, and updates the vertex distance if it is shorter.

Read-write vs. read-modify-write. Many graph algorithms conditionally update vertex data, where a thread reads the current value, performs a computation, and writes a new value. For example, in the BFS algorithm, the vertex distance is only updated if the new distance is shorter. This read-write approach works in certain situations, such as in Example 5a, because the updates are monotonic and the algorithm is resilient to temporary priority inversions. The read-modify-write style shown in Example 5b is more general but requires an atomic operation that may be slower and lowers parallelism.

Non-deterministic vs. deterministic. The unpredictable timing of threads can introduce internal non-determinism in parallel codes [37]. Consider Example 6a, where multiple threads may write an element of the dist array that is read by another thread. Depending on which thread performs the last write before the read, a different value may be read, leading to a different computation of a new distance. Any non-final distance value will be overwritten in subsequent iterations, meaning the final result is deterministic, but the number of iterations may differ from run to run. To make the code also internally deterministic, Example 6b uses two arrays, one that is only read (dist) and another that is updated (dist new).

4.2 Styles for specific programming models

This subsection describes styles that are applicable to specific programming models.

Persistent vs. non-persistent. This style only applies to GPU codes. The persistent style, shown in Example 7a in Fig. 3, uses as many threads as the GPU can concurrently schedule on its SMs [38], meaning a thread may need to process multiple vertices. In contrast, the non-persistent style in Example 7b launches as many threads as the input has vertices and assigns no more than 1 vertex to each thread.

Atomic vs. CudaAtomic. CUDA provides atomic functions to avoid data races. For instance, Example 8a uses atomicMin() to update a memory location. However, such functions cannot be employed in the host code running on the CPU. To address this limitation, CUDA recently introduced libcu++, a C++ Standard Library that can be used in both CPU and GPU code [39]. The corresponding “CudaAtomic” solution shown in Example 8b requires a special data type with an optional memory-ordering and scope specification, which was not available for atomic operations before.

Default scheduling vs. dynamic scheduling. OpenMP provides a convenient way to parallelize certain “for” loops using a “parallel for” directive. By default, as shown in Example 9a, this directive statically assigns each thread a contiguous chunk of loop iterations. In contrast, the dynamic schedule in Example 9b assigns iterations at runtime whenever a thread is ready to execute another iteration.

Blocked vs. cyclic. When parallelizing “for” loops, a blocked schedule assigns contiguous iterations to each thread, as shown in Example 10a. The cyclic schedule in Example 10b assigns the iterations in a round-robin fashion to the threads.

Thread vs. warp vs. block. This variation refers to the granularity at which a GPU program processes the vertices. Threads, warps, and blocks are the three hardware-supported granularities. In thread-based BFS, each thread processes all neighbors of a vertex as shown in Example 11a of Fig. 4. In warp- or block-based BFS, the entire warp or block processes the neighbors of a single vertex, respectively, as shown in Examples 11b and 11c.

Global-add vs. block-add vs. reduction-add. Reductions are widely used in parallel computing to combine multiple values into a single value using a binary associative operator [40]. Three different implementations are common in GPU codes. The first approach directly updates a shared global variable using atomic operations, as shown in Example 12a. The second approach in Example 12b takes

advantage of faster block-level atomics. The third approach uses warp-level primitives and a shared-memory buffer for the local results to perform the reduction, as presented in Example 12c.

Atomic-reduction vs. critical-reduction vs. clause-reduction. CPU codes also employ different reduction styles. OpenMP and C++ provide atomic operations, enabling each thread to update a shared variable atomically, as shown in Example 13a. Mutexes are also supported, allowing the programmer to update shared variables in critical sections, as shown in Example 13b. Additionally, OpenMP provides a reduction clause, as shown in Example 13c.

```
v = gidx;
if (v < nodes) {
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg; i < end; i++) {
    n = nbr_list[i];
    newd = dist[v] + 1;
    if (atomicMin(dist[v], newd) > newd) {
      go_again = true;
} } }
```

11a Thread

```
v = gidx;
if (v < nodes) {
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg + lane; i < end; i+= WarpSize) {
    n = nbr_list[i];
    newd = dist[v] + 1;
    if (atomicMin(dist[v], newd) > newd) {
      go_again = true;
} } }
```

11b Warp

```
v = gidx;
if (v < nodes) {
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg + tid; i < end; i+= blockDim) {
    n = nbr_list[i];
    newd = dist[v] + 1;
    if (atomicMin(dist[v], newd) > newd) {
      go_again = true;
} } }
```

11c Block

```
atomicAdd(result, value);
```

12a Global-add

```
atomicAdd_blk(block_result, value);
__syncthreads();
if (tid == 0)
  atomicAdd(&result, block_result);
```

12b Block-add

```
warp_result = warp_reduce(value);
__syncthreads();
block_result = blk_reduce(warp_result);
__syncthreads();
if (tid == 0)
  atomicAdd(result, block_result);
```

12c Reduction-add

```
#pragma omp parallel for
for (i = beg; i < end; i++) {
  #pragma omp atomic
  sum += value;
}
```

13a Atomic reduction

```
#pragma omp parallel for
for (i = beg; i < end; i++_ {
  #pragma omp critical
  sum += value;
}
```

13b Critical reduction

```
#pragma omp parallel for reduction(+: sum)
for (i = beg; i < end; i++) {
  sum += value;
}
```

13c Clause reduction

Figure 4. BFS example codes for styles with three alternatives

5 USING SAPPHIRE IN THE CLASSROOM

The following subsections describe ways in which we used or are planning on using Sapphire.

5.1 Presenting code examples

To enhance the students' understanding of each parallelization/ implementation style and the associated tradeoffs, instructors can use excerpts of the Sapphire codes, as illustrated in Figs. 3 and 4. Such code snippets can serve as examples for introducing various programming styles and for highlighting the differences between them.

In addition to BFS, Sapphire includes six more graph algorithms. They are single-source-shortest-path, connected components, maximal independent set, PageRank, triangle counting, and minimum spanning tree. In fact, Sapphire includes a separate version of each algorithm for each style and for all meaningful combinations of styles, that is, hundreds of runnable versions of each algorithm. The filename indicates the used styles. This extensive collection of codes enables instructors not only to comprehensively explain every style but also to do so within the context of their preferred algorithm. Moreover, it allows instructors to re-emphasize the learned material on additional examples, demonstrates how the styles can be combined, and offers the flexibility to introduce different examples when teaching the course the next time.

5.2 Executing performance studies

All codes in Sapphire are executable and compatible with graphs in CSR format (Section 2.3). This enables users to experiment with their own inputs, promoting hands-on learning. Sapphire accommodates various devices and encompasses graphs spanning diverse domains, allowing comprehensive performance experiments. Such experiments can illustrate several critical points.

- How the selected style or combination of styles impacts execution time
- How different algorithms prefer different styles
- How CPU and GPU implementations of the same algorithm may benefit from different styles
- How the graph properties affect which styles yield higher performance

For example, the topology-driven, push-style, deterministic, read-modify-write, thread-based, atomic BFS implementation is 113× faster than the corresponding version that uses default CudaAtomic. Teachers can easily change the scope and/or memory order in the CudaAtomic type declaration to demonstrate how the performance improves. Another example is the pull, persistent, block, globalAdd implementation of PageRank, which is 249× faster on our GPU than the corresponding push version.

5.3 Demonstrating common bugs

Sapphire includes common bugs, which are clearly labeled in the file name and using comments in the source code. This allows instructors to take students through a hands-on journey of exploring different bugs. They can execute the buggy codes to illustrate how the various bugs impact the program behavior. For instance, a program with data races may produce different output from run to run, and a livelock bug causes the program to run forever. Furthermore, the instructor can compare the buggy code with its bug-free version, allowing students to easily see the problems that cause the bug. Such practical demonstrations can help deepen students' understanding of potential coding pitfalls.

5.4 Applying styles to other codes

Since the styles we discuss in this paper are broadly applicable, they can also be used on other graph algorithms that the students may be interested in. For example, an instructor may select an important graph analytics problem, collaborate with the students to create a baseline implementation, and subsequently explore each style in detail. This process involves discussing which styles apply to the problem at hand and demonstrating how to add the chosen style to the code. In case clarification is needed, the Sapphire codes can serve as a valuable reference, offering plenty of examples to better understand the applicability of each style.

6 USING SAPPHIRE FOR ASSIGNMENTS

Hands-on exercises and assignments are essential to solidify the learned material as they allow the students to experiment with and apply the new knowledge to gain a deeper understanding of the parallelization and implementation styles. Sapphire supports a wealth of such exercises and assignments, as we outline in the following subsections.

6.1 Identify the style

With seven different graph problems to choose from and thousands of codes in total, Sapphire provides hundreds of examples of each style and multiple examples of each meaningful combination of styles. This enables instructors to select one or more files, anonymize them by changing the file name, and have the students analyze the C++, OpenMP, or CUDA code to identify which style(s) it employs. The difficulty of such exercises can be adjusted by choosing codes that combine fewer or more styles. Alternatively, the teacher can provide multiple anonymized source files and ask the students to find matching pairs that use the same set of styles.

6.2 Complete the code skeleton

To further solidify the learned material, teachers can ask their students to write part of the code themselves. Rather than requiring the implementation of an entire algorithm using a specific style, the instructor can take a few codes from Sapphire, remove some of the statements, and ask the students to fill in the missing parts. This makes it easy to adjust the level of difficulty, to focus on the aspects that the students struggle with the most, and to vary the assignments from semester to semester.

For example, a teacher can remove the red statements from the push and pull BFS codes in Fig. 3 and ask the students to recreate them. This hands-on exercise is likely to help students better understand how the two styles work and how they differ. By measuring the execution time of the resulting codes, the students can further learn about the performance implications of each style.

6.3 Fix the bugs

Instead of removing statements, instructors can also modify the Sapphire codes to introduce bugs that the students must track down and fix. Certain parallelization bugs, such as data races, can be difficult to find because they are thread-timing dependent and may not lead to wrong results every time the code is executed. Introducing such software defects allows the teacher to not only drive home how a style is correctly implemented but also how to debug parallel programs. Based on our experience, students tend to struggle particularly with debugging, yet this aspect is not typically covered in detail in current textbooks on parallel programming.

Inserting realistic software defects that trigger interesting behaviors is surprisingly challenging. To aid instructors in this endeavor, Sapphire includes a special directory with versions of its codes that already contain intentionally planted bugs. The filename indicates which specific bugs are present and what the corresponding bug-free code is (for reference). Sapphire covers a range of bugs, including data races, deadlocks, atomicity violations, threading mistakes, and incorrectly mixing synchronization. It also includes bugs that are not related to parallelism (e.g., overflow, out-of-bounds memory accesses, operator precedence errors, and using uninitialized data) as well as codes with multiple bugs. This variety enables teachers to select the kind of bug(s) they want to focus on, to change assignments between course offerings, and to control the difficulty of the exercise.

7 CONCLUSION

Teaching parallel programming has been part of the ACM Curricular Guidelines since 2013 and is now a requirement for ABET accreditation in the USA in computer science. This paper describes Sapphire, an educational tool to assist in teaching parallel programming. It offers a wide range of parallel graph codes, which are widely used in real-world applications but can be challenging to implement. Sapphire supports 13 distinct sets of parallelization and implementation styles for graph analytics codes. Each set encompasses 2 or 3 alternative styles, allowing for the creation of many unique combinations. In total, Sapphire contains 2276 codes that are based on 7 graph analytics problems. It further contains 10,000s of annotated versions of these codes with planted bugs, providing a comprehensive collection of examples of each style and bug. Sapphire can be used both in the classroom and for assignments. Using the code examples in class can help the teacher explain each style and show how to combine them. Each style has its pros and cons, allowing students to gain insights into the tradeoffs involved. Sapphire also enables instructors to run experiments that demonstrate how the implementation style and program input affect performance. Additionally, it can serve as a reference since the styles are general and can be used on other graph algorithms. By exploring the various styles, students can learn how to select efficient implementation styles for a given algorithm and input type.

The exercises and assignments supported by Sapphire are interesting and practical, enabling students to apply the learned material to graph analytics codes. For example, the instructor can select one or more codes and ask students to identify the styles. Similarly, the teacher can ask the students to write part of the code to complete a code skeleton. Moreover, since Sapphire includes a set of buggy codes, the instructor can design exercises and assignments for finding and fixing bugs. Through these hands-on activities, students can acquire a deeper understanding and become better programmers. We hope our work will enable instructors to teach parallel programming in a new and engaging way.

REFERENCES

[1] M. Burtscher, R. Nasre, and K. Pingali, “A quantitative study of irregular programs on GPUs,” in 2012 IEEE International Symposium on Workload Characterization (IISWC). IEEE, 2012, pp. 141–151.

[2] V. Kumar, A. Grama, A. Gupta, and G. Karypis, Introduction to parallel computing. Benjamin/Cummings Redwood City, CA, 1994, vol. 110.

[3] B. Wilkinson and M. Allen, “Parallel programming - techniques and applications using networked workstations and parallel computers (2.ed.),” 1998.

[4] M. J. Quinn, “Parallel programming,” TMH CSE, vol. 526, p. 105, 2003.

[5] M. McCool, J. Reinders, and A. Robison, Structured parallel programming: patterns for efficient computation. Elsevier, 2012.

[6] Y. Sun, T. Baruah, and D. Kaeli, Accelerated Computing with HIP, 12 2022.

[7] J. Sanders and E. Kandrot, CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional, 2010.

[8] P. Pacheco, An introduction to parallel programming. Elsevier, 2011.

[9] D. B. Kirk and W. H. Wen-Mei, Programming massively parallel processors: a hands-on approach. Morgan Kaufmann, 2016.

[10] B. Schmidt, J. Gonzalez-Dominguez, C. Hundt, and M. Schlarb, Parallel programming: concepts and practice. Morgan Kaufmann, 2017.

[11] S. A. Myers, A. Sharma, P. Gupta, and J. Lin, “Information network or social network? the structure of the twitter follow graph,” in Proceedings of the 23rd International Conference on World Wide Web, 2014, pp. 493–498.

[12] A. Bustamam, K. Burrage, and N. A. Hamilton, “Fast parallel Markov clustering in bioinformatics using massively parallel computing on GPU with CUDA and ellpack-r sparse format,” IEEE/ACM Transactions on Computational Biology and Bioinformatics, vol. 9, no. 3, pp. 679–692, 2012.

[13] N. Sivaramakrishnan and V. Subramaniyaswamy, “GPU-based collaborative filtering recommendation system using task parallelism approach,” in 2018 2nd International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 2018 2nd International Conference on, 2018, pp. 111–116.

[14] J. Dongarra, “Compressed row storage,” http://www.netlib.org/utk/ people/JackDongarra/etemplates/node373.html, accessed: 2023-4-26.

[15] J. Shun, Shared-Memory Parallelism Can Be Simple, Fast, and Scalable. Association for Computing Machinery and Morgan & Claypool, 2017, vol. 15.

[16] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The parsec benchmark suite: Characterization and architectural implications,” in Proceedings of the 17th international conference on Parallel architectures and compilation techniques, 2008, pp. 72–81.

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in 2009 IEEE international symposium on workload characterization (IISWC). IEEE, 2009, pp. 44–54.

[18] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, “The scalable heterogeneous computing (shoc) benchmark suite,” in Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, 2010, pp. 63–74.

[19] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, “Parboil: A revised benchmark suite for scientific and commercial throughput computing,” Center for Reliable and High-Performance Computing, vol. 127, 2012.

[20] J. Gomez-Luna, I. El Hajj, L.-W. Chang, V. Garcıa-Floreszx, S. G. De Gonzalo, T. B. Jablin, A. J. Pena, and W.-m. Hwu, “Chai: Collaborative heterogeneous applications for integrated-architectures,” in 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2017, pp. 43–54.

[21] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali, “Lonestar: A suite of parallel irregular programs,” in 2009 IEEE International Symposium on Performance Analysis of Systems and Software. IEEE, 2009, pp. 65–76.

[22] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, “Pannotia: Understanding irregular GPGPU graph applications,” in 2013 IEEE International Symposium on Workload Characterization (IISWC). IEEE, 2013, pp. 185–195.

[23] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, “Graphbig: understanding graph computing in the context of industrial solutions,” in SC ’15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2015, pp. 1–12.

[24] S. Beamer, K. Asanovic, and D. Patterson, “The gap benchmark suite,” arXiv preprint arXiv:1508.03619, 2015.

[25] Z. Xu, X. Chen, J. Shen, Y. Zhang, C. Chen, and C. Yang, “Gardenia: A graph processing benchmark suite for next-generation accelerators,” ACM Journal on Emerging Technologies in Computing Systems (JETC), vol. 15, no. 1, pp. 1–13, 2019.

[26] L. Dhulipala, J. Shi, T. Tseng, G. E. Blelloch, and J. Shun, “The graph-based benchmark suite (GBBS),” in Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), 2020, pp. 1–8.

[27] C. Liao, P.-H. Lin, J. Asplund, M. Schordan, and I. Karlin, “Dataracebench: a benchmark suite for systematic evaluation of data race detection tools,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2017, pp. 1–14.

[28] S. Schwitanski, J. Jenke, S. Klotz, and M. S. Muller, “Rmaracebench: A microbenchmark suite to evaluate race detection tools for rma programs,” ser. SC-W ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 205–214. [Online]. Available: https://doi.org/10.1145/3624062.3624087

[29] Y. Liu, N. Azami, C. Walters, and M. Burtscher, “The Indigo program verification microbenchmark suite of irregular parallel code patterns,” in 2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2022, pp. 24–34.

[30] S. Unkule, C. Shaltz, and A. Qasem, “Automatic restructuring of GPU kernels for exploiting inter-thread data locality,” in International Conference on Compiler Construction. Springer, 2012, pp. 21–40.

[31] A. Qasem, A. M. Aji, and G. Rodgers, “Characterizing data organization effects on heterogeneous memory architectures,” in 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2017, pp. 160–170.

[32] Y. Liu, N. Azami, A. Vanausdal, and M. Burtscher, “Choosing the best parallelization and implementation styles for graph analytics codes: Lessons learned from 1106 programs,” in SC ’23: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2023, p. 11111.

[33] P. Zhang and G. Chartrand, Introduction to graph theory. Tata McGraw- Hill, 2006.

[34] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Mendez-Lojo et al., “The TAO of parallelism in algorithms,” in Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, 2011, pp. 12–25.

[35] R. Nasre, M. Burtscher, and K. Pingali, “Data-driven versus topology-driven irregular computations on GPUs,” in 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. IEEE, 2013, pp. 463–474.

[36] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, “To push or to pull: On reducing communication and synchronization in graph computations,” in Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, 2017, pp. 93–104.

[37] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun, “Internally deterministic parallel algorithms can be fast,” in Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, 2012, pp. 181–192.

[38] K. Gupta, J. A. Stuart, and J. D. Owens, “A study of persistent threads style GPU programming for GPGPU workloads,” in 2012 Innovative Parallel Computing (InPar). San Jose, CA, USA: IEEE, 2012, pp. 1–14.

[39] “Nvidia, libcu++,” https://nvidia.github.io/libcudacxx/, May 2021, accessed: 2022-11-09.

[40] R. J. Lipton, “Reduction: A method of proving properties of parallel programs,” Communications of the ACM, vol. 18, no. 12, pp. 717–721, 1975.

[41] A. Aviram, S.-C. Weng, S. Hu, and B. Ford, “Efficient system-enforced deterministic parallelism,” Communications of the ACM, vol. 55, no. 5, pp. 111–119, 2012.