An Evaluation of UPC++ by Porting Shared-Memory Parallel Graph Algorithms

Preprint, compiled January 1, 2020

Alexander J. Ding*

Commonwealth School

Abstract

Unified Parallel C++ (UPC++), a C++ library, attempts to address the programming difficulty introduced by distributed parallel systems and still take advantage of the model's high scalability by exposing an API that represents the distributed memory as a contiguous global address space, similar to that of a shared-memory parallel system. Though previous work, including the various benchmarks by UPC++ developers, has demonstrated the library's effectiveness in simple tasks and in porting distributed-memory parallel algorithms that are often implemented in OpenMPI, there lacks an assessment of the ease and effectiveness of porting shared-memory parallel algorithms into UPC++. We implement a number of graph algorithms in OpenMP, a common shared-memory parallel library, and port them into UPC++ in a locality-aware, communication-averse manner to evaluate the convenience, scalability, and robustness of UPC++. Tests on both a single-node, multi-core system and the NERSC supercomputer (a multi-node system), with a plethora of real and random input graphs, demonstrate a number of prerequisites for high scalability in our UPC++ implementation: large input graphs, dense input graphs, and dense operations. Similar tests on our OpenMP implementation function as control, proving the algorithms' performance in shared-memory systems. Despite the relatively straightforward and naive porting from OpenMP, we still achieve competitive performance and scalability in dense algorithms on large inputs. The porting demonstrates UPC++'s ease of usage and good porting potential, especially when compared with other distributed libraries like OpenMPI. Finally, we extrapolate a distributed graph processing system on UPC++, optimized with a hybrid top-down/bottom-up approach, to simplify future distributed graph algorithm implementations.

***K*eywords** Parallel computing · Distributed systems · UPC++

1 Introduction

Recent years have seen CPU frequency growth slowing down as Moore's Law, which estimates that the number of transistors on a microchip doubles every two years, reaches its physical limit due to source-to-drain leakage and overheating [1]. The resulting hardware bottleneck for serial computing has pushed researchers to increasingly rely on optimizing parallel systems on both hardware and software levels to improve high performance computing. One key metric for such optimization is how well parallel systems scale, that is, as more and more processors are added to the system, how much its performance improve.

Within parallel systems, memory can be shared by or distributed amongst processors. Existing shared-memory parallel systems are easy to program and often involve nothing more than an implementation of the **parfor** (**parallel for**) for most use cases, but both their runtime and memory scale poorly as the systems become large. Distributed systems, in which nodes are connected together by a networks instead of any shared memory, achieve generally better scaling by sacrificing ease of programming. Instead of simply accessing local memory, nodes have to send messages through the network to read and write on other nodes' memory, and the added complexity of message management must be handled properly in order translate good hardware scalability to good production scalability.

Unified Parallel C++ (UPC++) [2] is a C++ library, built on the model of Unified Parallel C, supporting Partitioned Global Address Space (PGAS) programming. The library's memory model's goal is to take advantage of the distributed-memory infrastructure for its memory and processor scalability while simplifying the programming by logically representing the distributed system as a shared-memory one.

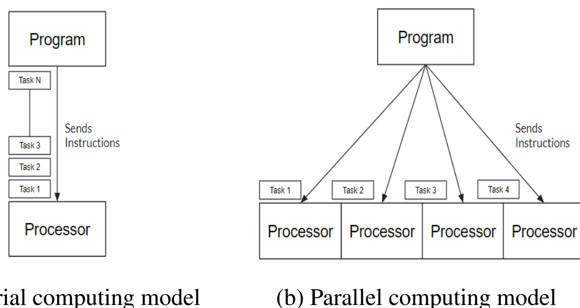
The original UPC++ paper [2] conducted a number of benchmarks such as random accessing, and experiments porting distributed-memory parallel algorithms such as like the OpenMPI-based LULESH [3]. In this paper, we explore UPC++'s potential for porting shared-memory parallel algorithms, broken down into an evaluation of our porting's performance compared with the original shared-memory implementation, its scaling, and the ease of porting. We implement common graph algorithms, including Breadth-First Search, PageRank, Bellman-Ford, and Connected Components, using OpenMP [4], a shared-memory library, on C++ and port them into a distributed version using UPC++ in a manner that maximizes node operation locality and minimizes communication. We run two rounds of testing on a variety of real graphs, retrieved from Stanford SNAP Large Network Dataset Collection [5], and random graphs. First, we test the algorithms on a single-node, multi-core system, varying the number of nodes used to compare our UPC++ implementation's single-node performance and scaling with OpenMP. Then, we run the same tests on our UPC++ implementations on a distributed multi-node, multi-core system on the supercomputing platform of the National Energy Research Scientific Computing Center (NERSC) to investigate our UPC++ implementation's scaling on a distributed system. We end our analysis with a discussion of UPC++'s ease of use compared to traditional distributed parallel systems.

Finally, we extrapolate from our UPC++ graph algorithm implementations a common compute-reduce round-based design paradigm for distributed graph algorithms and create a distributed graph processing system similar to Gemini [6] that supports dynamic scheduling between the top-down and bottom-up approach based on frontier size. The system handles the details of loading and partitioning input graphs, as well as the optimization and control logic shared by most graph algorithm, to allow programmers to focus on the unique logic of each algorithm during implementation.

This work and our experimental results give insight into how and when to use UPC++. The results confirm that distributed programming requires large input data and dense computation tasks to be effective due to its significant communication overhead, while shared-memory parallelism is still preferable when possible. Our relatively simple porting into UPC++ still achieves good scaling (up to 13.6x 32-node speedup in PageRank) and even better single-node performance than its OpenMP counterpart provided that the above prerequisites are met. The few changes needed to port our algorithms demonstrate the UPC++'s ease of use, thanks to its PGAS memory model and powerful API, especially when compared to dated yet popular distributed libraries such as OpenMPI.

2 BACKGROUND

2.1 Parallel Computing



(a) Serial computing model (b) Parallel computing model

Figure 1: Serial vs. parallel computing models

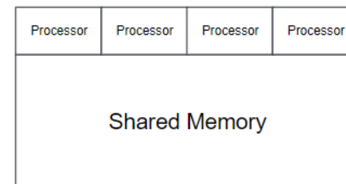
Traditional computing is done in a **serial** model, in which a single processor handles all the work needed to complete a program. The program sends a series of instructions to the processor, and the processor executes them one at a time. Since computation is serial, the complexity of an algorithm can be estimated by the total number of instructions sent to a processor using a Random Access Machine (RAM) model.

Parallel computing, on the other hand, uses a collection of processing elements (like the processors in Figure 1(b)) that can execute different tasks at the same time to cooperate to solve problems quickly. A major motivation for using parallel computing is to achieve a speedup, given by

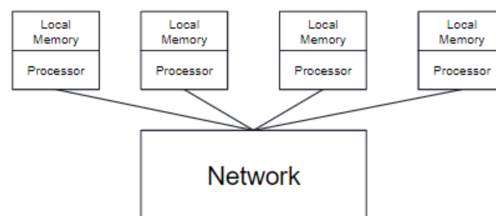
$$speedup_p = \frac{execution_time_1}{execution_time_p} \tag{1}$$

where $execution_time_p$ is the execution time using p processors. A parallel system introduces several complications. First, communication between the processors limits the maximum speedup by adding overhead to the system. Second, the program must assign work in a balanced manner amongst the processors to minimize idle processors waiting on others to finish work. Lastly, the processors must be coordinated to agree on the current state of the program and avoid data races.

2.2 Shared-Memory vs Distributed-Memory Parallelism



(a) Shared-memory model



(b) Distributed-memory model

Figure 2: Serial vs. parallel computing models

Within parallel computing, there are different models of representing memory.

Shared-memory parallelism is defined by a common view of the memory that all the processors share. A common example is a multi-core personal computer. Inter-processor communication is fast, as it simply entails accessing the shared memory. It is easy to program, as all processors share a single view of the memory and the program does not need to work to share any data between the processors. However, shared-memory system does not scale well easily, due to the logistical challenges of fitting more memory and processors on a single machine without experiencing significantly decreased memory access speed.

Distributed-memory parallelism lacks the shared memory and instead connects the processors by a network. One common example is a cluster of nodes connected together by the internet. Each node needs to explicitly pass messages along the network in order to read or write non-local data, and the latency in this message-passing communication, which is much slower than memory access, can become a significant overhead if not managed and minimized. Despite these challenges, a distributed system scales much better, as adding another machine to a network introduces little overhead. Therefore, distributed systems are frequently used for high-performance computing and for processing data too big to fit on a shared-memory system.

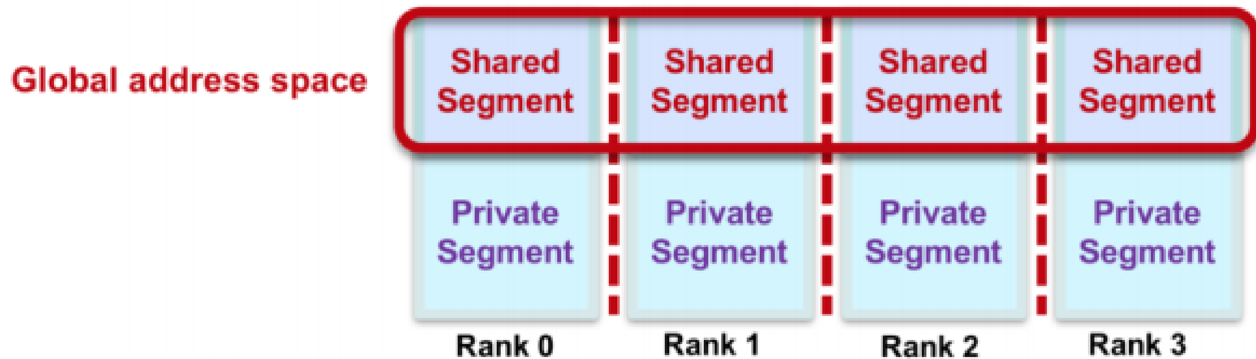


Figure 3: PGAS memory model, from UPC++ v1.0 Programmer's Guide [7]

It must be noted that the two levels of parallelization can and often are interspersed in a single system. In a cluster, individual nodes, for example, can each have multiple cores and utilize them for shared-memory parallelization on a local level.

2.3 *Unified Parallel C++*

Unified Parallel C++ (UPC++) [2] is a C++ library supporting high-performance computation by exposing a Partitioned Global Address Space (PGAS), which maps the local memory of each thread to a logical partition of a global memory address space. The programmer can use the same API calls to access data anywhere on the address space, which is represented using a special global pointer, whether in reality the data lies on a local memory address or on a different node connected to by the network; the library handles the logistics of manipulating local data in the former case and sending the appropriate message in the latter case. UPC++ imposes a clear separation between the global address space and a local private space such that only values on the shared segment (i.e., values with a global address) can be shared.

UPC++ runs on physically distributed systems and presents an abstraction of the distributed memory that imitates shared-memory parallelism. UPC++ explicitly casts all memory operations and supports Remote Procedure Call using a system of **FUTURE** and **PROMISE** to encourage programmers to consider the cost of communication. Additionally, UPC++ provides collectives to manage member nodes efficiently.

Though the library can work with clusters connected through UDP protocols over general internet, it is mainly designed for high-performance computing on supercomputer frameworks such as Cray Systems, whose computing nodes are connected by dedicated, high-speed network [8]. UPC++ can be used alongside shared-memory libraries like OpenMP to provide two levels of parallelization or be mixed with other distributed communication frameworks like OpenMPI.

The original paper on UPC++ [2] has demonstrated excellent performance at large scale, up to 30,000 cores on Cray XC30. It shows almost perfect strong scaling on Embree, a distributed ray tracing software that is mostly embarassingly parallel. A more recent paper [9] has shown linear weak scaling of distributed hash tables implemented using UPC++ up to 34816 cores on Cray XC40. The previous benchmarks are either too generic or highly specialized and optimized. An evaluation for the library's potential as a production tool for more general-purpose, real-world high-performance computing tasks such as graph algorithms is called for.

3 METHODOLOGY

We implement two similar versions of a suite of graph algorithms, one using OpenMP, a well-established shared-memory parallel library, and another using UPC++, which is distributed. The reason behind an OpenMP implementation is to demonstrate the efficiency and scalability of the algorithms themselves and establish a baseline for comparison, which helps us more effectively comment on how the use of UPC++ affects the final performance.

3.1 *Algorithms*

We implement a number of graph algorithms using both UPC++ and OpenMP: Breadth-First Search (BFS), Bellman-Ford, Connected Components, and PageRank.

3.1.1 *Breadth First Search*

A ***breadth-first-search*** looks through an unweighted graph $G = (V, E)$, starting with a root vertex $v \in V$, and computes a breadth-first search tree. We frame (and simplify) the algorithm by computing vector D of size $|V|$, where $D[i]$ is the smallest number of vertices between i and v. We define $D[v]$ as 0, and $D[j]$ where j is not connected to v as $D[j] = \infty$.

Algorithm 1 features two implementations of BFS, depending on the choice between **STEPSPARSE** and **STEPDENSE**. The two versions share a common overall structure. A frontier, including all the vertices, and all vertices in a frontier are the same distance away from the root vertex. The computation is divided into rounds. Each round, the algorithm explores vertices distance *round* away from the root vertex—i.e., the unexplored neighbors of the current frontier—, sets their distance values, and collects them as the next frontier. The top-down and bottom-up steps both execute the above steps, but they differ in efficiency depending on the frontier size.

Algorithm 1 BREADTHFIRSTSEARCH

```
 1: procedure BREADTHFIRSTSEARCH(G, v)
 2:     D ← {∞, ..., ∞}
 3:     D[v] ← 0    ▷ All distances start at ∞, but the root node has distance 0
 4:     DNext ← D
 5:     frontier ← {v}
 6:     frontierNext ← {}
 7:     round ← 1
 8:     while frontier ≠ {} do    ▷ Done exploring if frontier is empty
 9:         STEP ← CHOOSE(STEPSPARSE, STEPDENSE)    ▷ Chooses between top-down and bottom-up
10:         STEP(G, frontier, frontierNext, D, DNext, round)
11:         frontier ← frontierNext
12:         frontierNext ← {}
13:         SWAP(D, DNext)
14:         DNext ← D
15:         round ← round + 1
16:     return D
17:
18: procedure STEPSPARSE(G, frontier, frontierNext, D, DNext, round)
19:     for v ∈ frontier do    ▷ Iterate over the frontier
20:         for n ∈ v.neighbors do
21:             if DNext[n] = ∞ then    ▷ If vertex is not set yet, round ≤ DNext[n]
22:                 DNext[n] ← round
23:                 frontierNext ← frontierNext ∪ {n}
24:
25: procedure STEPDENSE(G, frontier, frontierNext, D, DNext, round)
26:     for v ∈ G.V do    ▷ Iterate over all vertices
27:         if DNext[v] = ∞ then    ▷ If vertex is not set yet, check if any neighbor is explored
28:             for n ∈ v.neighbors do
29:                 if n ∈ frontier then
30:                     DNext[v] ← round
31:                     frontierNext ← frontierNext ∪ {v}
32:                     break
```

Algorithm 2 BELLMANFORD

```
 1: procedure BELLMANFORD(G, v)
 2:     D ← {∞, ..., ∞}
 3:     D[v] ← 0    ▷ All distances start at ∞, but the root node has distance 0
 4:     DNext ← D
 5:     frontier ← {v}
 6:     frontierNext ← {}
 7:     round ← 1
 8:     while frontier ≠ {} & round < |G.V| do    ▷ Done exploring if frontier is empty or there are negative loops
 9:         STEP ← CHOOSE(STEPSPARSE, STEPDENSE)    ▷ Chooses between top-down and bottom-up
10:         STEP(G, frontier, frontierNext, D, DNext, round)
11:         frontier ← frontierNext
12:         frontierNext ← {}
13:         SWAP(D, DNext)
14:         DNext ← D
15:         round ← round + 1
16:     if round = |G.V| then
17:         return "Negative Loops"    ▷ There are negative cycles
18:     return D
19:
20: procedure STEPSPARSE(G, frontier, frontierNext, D, DNext, round)
21:     for v ∈ frontier do    ▷ Iterate over the frontier
22:         for n ∈ v.neighbors do
23:             if D[v] + G.W[v, n] < DNext[n] then
24:                 DNext[n] ← D[v] + G.W[v, n]    ▷ Relax the edge if possible
25:                 frontierNext ← frontierNext ∪ {n}
26:
27: procedure STEPDENSE(G, frontier, frontierNext, D, DNext, round)
28:     for v ∈ G.V do    ▷ Iterate over all vertices
29:         for n ∈ v.neighbors do
30:             if n ∈ frontier then
31:                 if D[n] + G.W[v, n] < DNext[v] then
32:                     DNext[v] ← D[n] + G.W[v, n]
33:                     frontierNext ← frontierNext ∪ {v}
```

For a small frontier, the top-down step is much more efficient, as its outer for-loop iterates few elements. However, for a frontier whose size is a significant fraction of the total vertices, the sparse step allows the inner for-loop to terminate early and avoid wasted computation. Previous work [10] has shown that heuristically minimizing the number of edges scanned each round by choosing between the two versions can be highly effective. For simplicity, this paper uses the heuristic function:

$$\text{STEP} \leftarrow \begin{cases} \textsc{StepSparse} & \textsc{Size}(frontier) < \frac{\textsc{Size}(G.V)}{20} \\ \textsc{StepDense} & \text{otherwise} \end{cases} \quad (2)$$

A few notes are due here. First, for BFS, we can get by without using two copies of D, but we choose not to for conformity with the overall architecture of the other algorithms. Second, all the algorithms discussed work on directed graphs just as easily. Simply modify v.neighbors in **STEPSPARSE** to v.out_neighbors and v.neighbors in **STEPDENSE** to v.in_neighbors. We stick with undirected graphs in our pseudocode for clarity.

The other graph algorithms implemented can all be treated with this frontier-round and hybrid top-down/bottom-up method. We discuss them briefly below.

3.1.2 Bellman-Ford

Bellman-Ford computes the distance of the shortest path between a root vertex v and each vertex, as a distance vector D. It takes a weighted graph $G = (V, E, W)$, in which the weight vector W represents the distance of each edge.

Note that allowing the existence negative weights would possibly lead to negative cycles. A simple heuristic determines their existence: if the algorithm runs for more than $|V|$ rounds, there must be a negative cycle. As each round relaxes all edges once,

Algorithm 3 ConnectedComponents

```
1: procedure ConnectedComponents(G)
2:     D ← {1, 2, ..., |G.V|}  ▷ Vertices start in own component
3:     DNext ← D
4:     frontier ← {1, 2, ..., |G.V|}
5:     frontierNext ← {}
6:     round ← 1          ▷ Keeping this variable for consistency
7:     while frontier ≠ {} do          ▷ Done if frontier is empty
   ▷ Chooses between top-down and bottom-up
8:         Step ← Choose(StepSparse, StepDense)
9:         Step(G, frontier, frontierNext, D, DNext, round)
10:        frontier ← frontierNext
11:        frontierNext ← {}
12:        Swap(D, DNext)
13:        DNext ← D
14:        round ← round + 1
15:    return D
16:
17: procedure StepSparse(G, frontier, frontierNext, D, DNext, round)
18:    for v ∈ frontier do          ▷ Iterate over the frontier
19:        for n ∈ v.neighbors do
20:            if D[v] < DNext[n] then
21:                DNext[n] ← D[v]
22:                frontierNext ← frontierNext ∪ {n}
23:
24: procedure StepDense(G, frontier, frontierNext, D, DNext, round)
25:    for v ∈ G.V do          ▷ Iterate over all vertices
26:        for n ∈ v.neighbors do
27:            if n ∈ frontier then
28:                if D[n] < DNext[v] then
29:                    DNext[v] ← D[n]
30:                    frontierNext ← frontierNext ∪ {v}
```

and a (non-cyclic) shortest path is at most of length $|V|$, any graph free of negative cycles would be fully explored within $|V|$ rounds.

The rest of the details are the same as Algorithm 1.

3.1.3 Connected Components

For an unweighted graph $G = (V, E)$, a connected component $C \subset V$ satisfies the requirement that all its vertices can reach each other through a finite number of edges. The ***Connected components*** algorithms divides all vertices into connected components. Namely, it computes a vector D of size $|V|$, where vertices in the same component share the same value in D, and vertices that are not connected have different values in D.

We can use the same setup as BFS. We initialize $D[i] = i$ for all i, representing our initial assumption that each vertex belongs in a different component. Each round, we check all edges connected to the frontier and for each edge connecting vertices v, w, we set $D[v]$ and $D[w]$ to be $min(D[v], D[w])$, effectively joining two components over many rounds. The remaining details of the sparse and dense steps are immediate.

Algorithm 4 PageRank

```
1: procedure PageRank(G)
2:     D ← {1/|G.V|, 1/|G.V|, ..., 1/|G.V|}
3:     DNext ← D
4:     round ← 1
5:     maxIters ← M
6:     while round < maxIters do          ▷ Done after max rounds
7:         Step ← StepDense
8:         Step(G, D, DNext round)
9:         Swap(D, DNext)
10:        DNext ← D
11:        round ← round + 1
12:    return D
13:
14: procedure StepDense(G, D, DNext, round)
15:    for v ∈ G.V do          ▷ Iterate over all vertices
16:        DNext[v] ← (1−γ)/|G.V|
17:        for n ∈ v.neighbors do
18:            DNext[v] ← DNext[v] + D[n]/|n.neighbors|
```

3.1.4 PageRank

PageRank is originally used by Google to rank the relative importance of websites. It takes an unweighted graph $G = (V, E)$ and a damping factor $0 \leq \gamma \leq 1$. It maintains a vector D of length $|V|$, whose values sum to 1. It initializes the all entries to be $\frac{1}{|V|}$ and iterates over all vertices until their values converge onto some arbitrary threshold:

$$D[i] = \frac{1-\gamma}{|V|} + \gamma \sum_{u \in \text{v.neighbors}} \frac{D[u]}{|u.neighbors|} \quad (3)$$

For our purposes, we modify the algorithm so that it ends after a certain number of iterations to avoid excessively long runtime during tests.

Since we update all vertices each round, there is no need to maintain a frontier or have StepSparse. Note, however, that the frontier-round paradigm used in the other algorithms can easily be maintained by adding each vertex to the next frontier only when $round \leq max_iters$.

3.2 Implementation

We implement the parallel versions of these direction-optimized graph algorithms for OpenMP and UPC++. This involves representing the data efficiently, parallelizing the serial algorithms, and, for UPC++, partitioning the data.

3.2.1 Representation

First, to efficiently fit graphs into the memory, we use Compressed Sparse Rows (CSR) representation. For a directed, unweighted graph G, we maintain two vectors O (offsets) and E (edges) with sizes $\text{Size}(G.V)$ and $\text{Size}(G.E)$, respectively. For vertex v that has m neighbors, write

$$O[v] = \begin{cases} O[v-1] + k & v > 0 \\ 0 & v = 0 \end{cases} \quad (4)$$

Algorithm 5 Primitives

1: **procedure** CAS(addr, oldVal, newVal)
2:     **if** $*addr \neq oldVal$ **then**
3:         **return** false
4:     $*addr \leftarrow newVal$
5:     **return** true
6:
7: **procedure** PriorityUpdate(addr, newVal, **Ordering** ← lambda a b: $a < b$)
8:     $oldVal \leftarrow *addr$
9:     **while Ordering**($newVal$, $oldVal$) **do**
10:         **if CAS**($addr$, $oldVal$, $newVal$) **then**
11:             **return** true
12:         $oldVal \leftarrow *addr$
13:     **return** false
14:
15: **procedure** PlusScan(AIn, AOut)
    ▹ Outputs the cumulative sum of an input array in the output array
16:
17: **procedure** Filter(AIn, AOut, **P**)
    ▹ Outputs values filtered from the input array in the output array as a continuous block
    ▹ Returns the number of values that pass the filter
18:
19: **procedure** SumFlags(AIn)
20:     $AOut \leftarrow \{0, 0, ..., 0\}$ ▹ $|AIn|$ 0's
21:     **PlusScan**(AIn, $AOut$, $lambdat : t$)
22:     **return** $AOut[|AOut| - 1]$

Algorithm 6 BreadthFirstSearch on OpenMP (helpers)

1: **procedure** DenseToSparse(denseFrontier, frontierSize, sparseFrontier)
2:     **parfor** $i \in [0, frontierSize - 1]$ **do** ▹ Iterate over the frontier
3:         **if** $denseFrontier[i]$ **then**
4:             $sparseFrontier[i] = i$
5:         **else**
6:             $sparseFrontier[i] = -1$
7:     **Filter**($sparseFrontier$, $sparseFrontier$, $frontierSize$, lambda x: $x \geq 0$)
8:
9: **procedure** SparseToDense(sparseFrontier, frontierSize, denseFrontier)
10:     **parfor** $i \in [0, frontierSize - 1]$ **do** ▹ Iterate over the frontier
11:         $denseFrontier[sparseFrontier[i]] \leftarrow$ true
12:
13: **procedure** StepSparse(G, frontier, frontierNext, D, DNext, round)
14:     **parfor** $v \in frontier$ **do** ▹ Iterate over the frontier
15:         **for** $n \in v$.neighbors **do**
16:             **if CAS**($DNext[n]$, ∞, $round$) **then** ▹ Atomic update to avoid repeated entry into sparse frontier
17:                 $frontierNext[n] \leftarrow n$
18:     **return Filter**($frontierNext$, $frontier$, $|G.V|$, lambda x: $x \geq 0$)
19:
20: **procedure** StepDense(G, frontier, frontierNext, D, DNext, round)
21:     **parfor** $v \in G.V$ **do** ▹ Iterate over all vertices
22:         **if** $DNext[v] = \infty$ **then** ▹ If vertex not set yet
23:             **for** $n \in v$.neighbors **do**
24:                 **if** $n \in frontier$ **then**
25:                     $DNext[v] \leftarrow round$
26:                     $frontierNext[v] \leftarrow$ true
27:                     **break**
28:     **return SumFlags**($frontierNext$, $|G.V|$)

And for $k \in [0, m - 1]$,

$$E[O[v] + k] = G.V[v].neighbors[k] \tag{5}$$

Each vertex i's neighbors are $\{E[k] \mid k \in [O[i], O[i + 1])\}$. A weighted graph has another vector W similar to E, where $W[k]$ is the weight of the directed edge represented by $E[k]$.

Note that the neighbors can be incoming neighbors or outgoing neighbors for a directed graph. By storing both versions of CSR representation, we can iterate through or count any vertex's incoming and outgoing neighbors in constant time. We implement a custom Graph class to abstract these functionalities.

To port our algorithms into UPC++, we need to partition the graph amongst the distributed nodes when loading. In our straightforward implementation, we divide the graph evenly, i.e., in a system with n nodes, node i owns vertices $[i\frac{|V|}{n}, (i + 1)\frac{|V|}{n}]$ and their edges; in other words, these vertices are local to node i. When implementing the algorithms, each node only processes its local vertices.

3.2.2 OpenMP Implementation

Implementing the algorithms for OpenMP requires a number of parallel primitives. The **parallel-for** (**parfor**) schedules iterations of a for loop amongst the processors, and it is built into OpenMP as a preprocessor directive. **Compare-and-swap** (**CAS**) is an atomic version of the provided pseudocode, allowing safe writing within parallel loops, and it is provided as part of the standard library of C++. **Priority update** takes an address, a value, and an arbitrary ordering (i.e., the priority part of the priority update; one common example is the numerical ordering of real numbers), and it uses **CAS** to ensure that the given address will eventually have a value with at least the same priority as the given value.

We also use **plus scan**, which gives the cumulative sum of an array, and **filter**, which filters values from an array and packs those which satisfy a given predicate in another array. Their parallel implementations are directly taken from the Problem Based Benchmark Suite [11] and therefore not describe in detail. Lastly, we define a helper function **sum flags** (**SumFlags**), which sums the number of true's in a Boolean array.

Now we're ready to parallelize the algorithms. Let's take BFS as an example. The basic layout is the same as algorithm 1, but we make a few changes:

1. We change the outer for loops into **parfor**.
2. We atomically write to the D vector using **CAS** to avoid data races.

Algorithm 7 **BREADTHFIRSTSEARCH** on OpenMP (main)

```
procedure BreadthFirstSearch(G, v)
    D ← [∞, ..., ∞]
    D[v] ← 0                                  ▹ All distances start at ∞, but the root node has distance 0
    DNext ← D

    sparseFrontier ← [v, −1, −1, ..., −1]
    sparseFrontierNext ← [−1, −1, ..., −1]
    frontierSize ← 1

    denseFrontier ← [false, ..., false]       ▹ No need to initialize dense frontier correctly; we start in sparse mode
    denseFrontierNext ← denseFrontier

    wasSparse ← true                          ▹ Maintain a variable tracking last round's mode
    round ← 1

    while frontierSize ≠ 0 do                 ▹ Done exploring if frontier is empty
        shouldBeSparse ← frontierSize < |V|/20   ▹ Chooses between top-down and bottom-up
        if shouldBeSparse then
            if !wasSparse then                ▹ Convert if last round was in a different mode
                DenseToSparse(denseFrontier, |G.V|, sparseFrontier)
            wasSparse ← true
            frontierSize ← StepSparse(G, sparseFrontier, sparseFrontierNext, D, DNext, round)
        else
            if wasSparse then                 ▹ Convert if last round was in a different mode
                SparseToDense(sparseFrontier, frontierSize, denseFrontier)
            wasSparse ← false
            frontierSize ← StepDense(G, denseFrontier, denseFrontierNext, D, DNext, round)
        Swap(D, DNext)
        DNext ← D
        parfor i ∈ [0, |G.V| − 1] do          ▹ Reset next round's variables
            sparseFrontierNext[i] = −1
            denseFrontierNext[i] = false
            DNext[i] = D[i]
        round ← round + 1
    return D
```

3. We maintain two versions of the frontier, sparse and dense, and convert between them as needed using **PARFOR**.

Specifically, we represent the dense frontier as a Boolean array of size $|V|$, where if *denseFrontier*$[i]$ is true if and only if vertex i is in the frontier; and we represent the sparse frontier as a vector of the vertex indices of frontier members. In implementation, it is represented by an array of size $|V|$ and a *frontierSize* variable. The dense version sacrifices some efficiency in enumeration (which is hardly noticeable when the frontier is dense) for an efficient membership check, while the sparse version allows for an efficient enumeration of a small frontier.

Note that in **STEPSPARSE**, we first treat *frontierNext* as effectively a Boolean array, where *frontierNext*$[i] = -1$ if vertex i is not in the frontier, and *frontier*$[i] = i$ if it is. We then turn this into a contiguous vector via a call to **FILTER**.

Moreover, in **STEPDENSE**, we can directly assign values within parallel for loops without atomic operations because the within one round, the values are the same. This is not generally true for other algorithms, but we can easily use **PRIORITYUPDATE** to modify the steps.

As another example, we examine our implementation of connected components. The main body of the algorithm can be translated from our sketch, Algorithm 3, the similar to the way BFS is translated. Indeed, the same could be said about all other algorithms used. Therefore, we only present **STEPSPARSE** and **STEPDENSE** of connected components.

3.2.3 UPC++ Implementation

We port the graph algorithm suite into UPC++, keeping the algorithms' structure the same our OpenMP version. Each node maintains a sparse frontier and a dense frontier, which are synchronized at the end of each round along with the value vector D. In light of the distributed infrastructure, we modify our code to respect locality. Each node is responsible for processing only frontier members that are local, adding to the next round's frontier and updating values as needed. In sparse mode, each node updates the values of the neighbors of its local vertices. In dense mode, each node updates the values of local vertices.

Algorithm 8 CONNECTEDCOMPONENTS on OpenMP

1: **procedure** STEPSPARSE(G, frontier, frontierNext, D, DNext, round)
2: **parfor** $v \in frontier$ **do** ▷ Iterate over the frontier
3: **for** $n \in v$.neighbors **do**
4: **if** PRIORITYUPDATE($DNext[n]$, $D[v]$) **then** ▷ Atomic update to avoid repeated entry into sparse frontier
5: $frontierNext[n] \leftarrow n$
6: **return** FILTER($frontierNext$, $frontierNext$, $|G.V|$, lambda x: $x \geq 0$)
7:
8: **procedure** STEPDENSE(G, frontier, frontierNext, D, DNext, round)
9: **parfor** $v \in G.V$ **do** ▷ Iterate over all vertices
10: **for** $n \in v$.neighbors **do**
11: **if** $DNext[v] \in D[n]$ **then** ▷ No need to atomically update, as only one thread could update a vertex at a time
12: $DNext[v] \leftarrow D[n]$
13: $frontierNext[v] \leftarrow$ true
14: **return** SUMFLAGS($frontierNext$, $|G.V|$)

Algorithm 9 SYNCROUNDSPARSE on UPC++

1: **procedure** REDUCEALL(src, dst, op)
▷ UPC++ collective. Reduces each element of an array *src* across all nodes and outputs at *dst*.
2:
3: **procedure** BARRIER
▷ UPC++ collective. Blocks execution until all nodes reach this line.
4:
5: **procedure** SYNCROUNDSPARSE(frontierNext, DNext)
6: REDUCEALL($frontierNext$, $frontierNext$, MAX)
7: REDUCEALL($DNext$, $DNext$, MIN)
8: BARRIER()

Algorithm 10 SYNCROUNDDENSE on UPC++

1: **procedure** BROADCAST(src, origin)
▷ UPC++ collective. Broadcasts an array *src* from node *origin*.
2:
3: **procedure** SYNCROUNDDENSE(frontierNext, DNext)
4: **for** $i \in [0, NumNodes - 1]$ **do**
5: BROADCAST($frontierNext[i\frac{|V|}{n} : (i+1)\frac{|V|}{n}]$, i)
6: BROADCAST($DNext[i\frac{|V|}{n} : (i+1)\frac{|V|}{n}]$, i)
7: BARRIER()

To transfer our OpenMP implementation, we first replace the generic C++ arrays with the global pointers supplied by UPC++ so that they can be communicated between nodes. A global pointer [7] represents an address on the global address space, and if this address also corresponds to a memory address local to the node, it can be downcasted into a regular C++ pointer.

The main body of the algorithm, except for a change in the array implementation, is the same as before. Likewise, the conversion functions between sparse and dense frontiers are serialized since each node is assumed to be single-core, but otherwise kept the same. Therefore, parallel primitives such as **FILTER** and **SUMFLAGS** are serialized, and atomic operations such as **PRIORITYUPDATE** and **CAS** are replaced with computationally cheaper read and writes that are not thread-safe due to the lack of local parallelization.

To minimize communication, we aggregate all changes during each round share results only at the end of the round. For this, we implement two additional functions, **SYNCROUNDSPARSE** and **SYNCROUNDDENSE**, to synchronize the frontiers and distances and reach consensus amongst nodes, which are only invoked at the end of each round, forming a **compute-communicate** cycle.

For **SYNCROUNDSPARSE**, since each node may update values of non-local vertices within each round, finding the true value involves considering the corresponding array values of all nodes and reducing them into one consensus value. For this, we use the UPC++ collective **reduce_all**, which takes the global array by the same name and reduces the candidate values into one by applying a binary operator repeatedly.

Specifically, for reducing the sparse frontier, we use the UPC++ built-in reduction operator **op_fast_max** to take the maximum value before applying the filter operation. Recall that, before filtering, $frontier[i] = -1$ if vertex i is in the frontier, and $frontier[i] = i$ otherwise. This reduction effectively unionizes all the frontiers in our nodes. The value array can be reduced similarly, though the specific reduction operator varies depending on the use case. For our algorithms, taking the minimum value is the desired behavior.

In reality, the **reduce_all** function returns a **future**, which we simply wait to complete. We omit these technical details here.

For **SYNCROUNDDENSE**, since each node only update values of local vertices, the true value for a vertex lies in the node that owns the vertex. Therefore, reaching consensus means that each node broadcasts the values of its local vertices. For this, we use the UPC++ collective **broadcast**, which takes a global array and an origin node, and it broadcasts the values of the array at the origin node to all nodes (and sets their corresponding values to be the origin node's values).

4 EXPERIMENT

Our evaluation of UPC++ is twofold.

First, we compare the UPC++ and OpenMP implementations side-by-side on a single-node, multi-core machine. Though UPC++ is designed for multi-node supercomputers, this experiment investigates the scalability of UPC++ implementation's internal logic without communication latency as a major factor. Then we benchmark UPC++ on the NERSC supercomputer, using a multi-node setup to evaluate our graph algorithm suite. This experiment demonstrates the production scalability of our UPC++ implementation.

For both setups, we run a strong scaling and a weak scaling test. **Strong scaling** is how the runtime varies with the number of nodes for a fixed problem size. However, since parallel speedup is ultimately limited by the serial fraction of the code, it is helpful to understand how the runtime scale as problem

sizes scale, which motivates us to measure weak scaling. **Weak scaling** refers to how the runtime varies with the number of nodes for fixed problem size per node.

4.1 Setup

Our single-node, multi-core machine is an Amazon Web Service c5.18xlarge instance. It has 72-core (with hyper-threading) with 32×3.6GHz 2-core Intel Xeon Scalable Processors (Cascade Lake), 144GB of main memory, and a 14Gbps bus. The OpenMP code is compiled using `g++ 7.3.1` with flags `-std=c++11` and `-fopenmp` set. The UPC++ code is compiled using UPC++ version 20190900L, with flags `-threadmode=seq` and `-codemode=O3` set.

Our NERSC experiments use the Haswell Compute Nodes. Each node contains two sockets, with each socket being populated with a 2.3GHz 16-core Intel Xeon Processor E5-2698 v3 and 128GB of main memory.

Input	# Vertices	# Directed Edges
YouTube	1.13×10^6	5.98×10^6
Orkut	3.07×10^6	2.34×10^8
Friendster	6.56×10^7	1.81×10^9
rMat24	3.36×10^7	3.34×10^8

Table 1: Input graphs of the experiments

The input graphs used in our strong scaling experiments are shown in Table 1. ***YouTube***, ***Orkut***, and ***Friendster*** are all undirected user community graphs retrieved from the Stanford Large Network Dataset Collection [5]. ***RMat24*** is a synthetic graph with a power-law distribution of degrees [12], generated with parameters $a = 0.5$, $b = c = 0.1$, $d = 0.3$. For our weak scaling experiments, we generate rMat graphs of exponentially increasing number of vertices using the same parameters. The smallest graph has 1.6×10^4 vertices and 1.8×10^5 directed edges; the largest graph (32 times the size) has 5.2×10^5 vertices and 6.3×10^6 directed edges. We added weights to the graphs by assigning randomly generated integers between 1 and 10 to each edge, converted the undirected graphs into directed graphs, and symmetrized the graphs.

4.2 Result

The results of our AWS experiments are shown in Table 2 and Table 4. We are unable to perform scaling tests on large graphs such as **rMat24** and **Friendster**, as each process of our UPC++ implementation attempts to create an array of size $|V|$, and all these arrays are to be stored in the main memory of our single-node machine, which leads to insufficient memory. This behavior is as expected, as, in practice, each UPC++ process will be operating on a separate node that has its own memory.

Performance. As a baseline, we notice consistently good strong scaling of all four algorithms in their OpenMP implementation, though the larger graphs scale better. In particular, **YouTube** scales the least well due to its small graph size and low density. This can be explained as small graphs amplify the effect of algorithm overheads that do not scale, and low graph density leads to sparse frontiers that are as effectively parallelized as dense ones. This claim is supported by the fact that the OpenMP BFS achieves good runtime in the weak scaling test, which mitigates the effect of serial overhead.

In single-core computations, our UPC++ implementation achieves better runtime across the board, a surprising result given that UPC++ is designed to accommodate distributed systems, while OpenMP is written specifically for shared-memory systems such as this experiment machine, and both implementations are based on the same overall architecture. This may be due to the fact that our UPC++ implementation replaced atomic operations with normal reads and writes, or this may be indicative of UPC++'s cheaper initialization overheads than OpenMP.

Our UPC++ implementation scales in the right direction on **Orkut**, but not **YouTube**, and it performs the worst in BFS (in strong scaling) and Bellman-Ford (in weak scaling). This can be explained with the same reasoning. Sparse algorithms such as BFS and Bellman-Ford do not scale well, as they frequently invoke the sparse step, which needs to be synced at round end by an expensive reduction step. The unscalable overhead from looking through the aggregate frontier each round and identifying local members in sparse rounds renders sparse steps much more expensive in systems with large core count than dense rounds, which only loop through the entire local frontier range (i.e., vertices $[i\frac{|V|}{n}, (i+1)\frac{|V|}{n}$ for process i in an n processor system) no matter how large the system. This is an inherent issue in directly porting shared-memory parallel algorithms into distributed infrastructures. Note that the dense step is able to scale well due to our algorithms' locality awareness.

The results of our NERSC experiments are shown in 3 and 5. They confirm our hypothesis that our algorithms scale with dense graphs much more effectively. As in the AWS experiments, BFS fails to scale well. Overall, the denser a graph is (from densest to least, **Orkut**, **Friendster**, **rMat24**, **YouTube**), the better its scalability. Connected components and PageRank scale much better than BFS due to their larger frontiers. Our relatively naive porting is still able to achieve a 13.6x speedup on PageRank with **Orkut**, as well as a 7.27x speedup on connected components, confirming that our dense step is efficient in a distributed setting.

Programmability. With a more careful, fine-grained distributed-memory implementation of parallel graph algorithms, such as the Gemini system [6], we can achieve much better results. As it stands, we still gain the capability to process much larger graphs and easily scale up the system. As demonstrated by the brevity of our source code, UPC++ is much easier to use than traditional distributed libraries such as OpenMPI, which require manual managing of inter-node messages. The simple, intuitive collective and the future system hide the details of communication under clear abstractions and allow for simple, readable code with the same level of control as OpenMPI.

# Cores	BFS				Bellman-Ford				Connected Components				PageRank			
OpenMP	rM	YT	Orkut	Fs	rM	YT	Orkut	Fs	rM	YT	Orkut	Fs	rM	YT	Orkut	Fs
1	10.2	0.21	2.67	67.2	121	0.97	34.3	958	89.5	0.70	30.3	2128	126	0.98	32.9	1753
2	5.88	0.13	1.45	24.8	78.6	0.74	18.2	640	56.3	0.49	15.3	1336	80.4	0.73	20.8	1418
4	3.57	0.08	1.06	16.0	57.0	0.59	9.80	159	32.8	0.34	8.45	670	48.1	0.50	12.0	901
8	1.83	0.05	0.59	16.6	23.1	0.27	7.24	312	22.0	0.24	4.62	359	28.9	0.34	7.11	463
16	0.99	0.04	0.37	2.63	16.1	0.28	4.69	92	12.9	0.16	3.59	183	18.9	0.21	4.89	238
32	0.61	0.03	0.25	4.98	11.8	0.18	2.52	54	7.29	0.12	1.90	107	10.2	0.14	2.58	136
32 SU	16.8	7.79	10.8	13.4	10.2	5.50	13.5	17.7	12.2	5.94	15.9	19.8	12.3	6.80	12.7	12.8
UPC++																
1	N/A	0.09	0.84	N/A	N/A	0.47	13.9	N/A	N/A	0.34	12.0	N/A	N/A	0.25	8.56	N/A
2	N/A	0.22	0.82	N/A	N/A	0.55	8.44	N/A	N/A	0.34	7.36	N/A	N/A	0.22	5.52	N/A
4	N/A	0.40	0.87	N/A	N/A	0.81	5.60	N/A	N/A	0.39	4.77	N/A	N/A	0.25	3.71	N/A
8	N/A	0.55	0.84	N/A	N/A	0.87	4.60	N/A	N/A	0.43	3.16	N/A	N/A	0.25	2.47	N/A
16	N/A	0.82	0.95	N/A	N/A	1.39	4.33	N/A	N/A	0.59	2.56	N/A	N/A	0.27	2.02	N/A
32	N/A	1.39	1.34	N/A	N/A	1.84	3.64	N/A	N/A	0.80	2.04	N/A	N/A	0.35	1.75	N/A
32 SU	N/A	0.07	0.62	N/A	N/A	0.25	3.82	N/A	N/A	0.42	5.91	N/A	N/A	0.71	4.87	N/A

Table 2: Strong scaling runtime (in seconds) on the single-node AWS instance. **rM** is rMat24; **YT** is **YouTube**; **Fs** is **Friendster**. **32 SU** is the speedup achieved by using 32 nodes, calculated as the single-node runtime divided by the 32-node runtime. **rMat24** and **Friendster** were unable to fit in memory during the multi-node UPC++ tests as each process creates a size $|V|$ frontier array.

# Cores	Algorithms			
OpenMP	BFS	Bellman-Ford	CC	PageRank
1	0.05	0.49	0.25	0.42
2	0.06	0.62	0.36	0.59
4	0.08	0.74	0.46	0.76
8	0.09	1.01	0.59	0.92
16	0.12	1.38	0.86	1.30
32	0.14	2.43	1.35	2.25
UPC++				
1	0.02	0.34	0.12	0.09
2	0.05	0.62	0.19	0.15
4	0.22	1.04	0.37	0.31
8	0.55	2.38	0.78	0.63
16	1.89	4.66	1.85	1.18
32	6.19	11.7	5.34	2.66

Table 4: Weak scaling runtime (in seconds) on the single-node AWS instance using scaling **rMat** graphs.

# Cores	Algorithms		
OpenMP	BFS	CC	PageRank
1	0.03	0.11	0.12
2	0.09	0.25	0.25
4	0.40	1.17	0.82
8	1.76	3.08	1.90
16	5.79	5.26	3.82
32	26.4	14.4	9.93

Table 5: Weak scaling runtime (in seconds) on the NERSC supercomputer using scaling **rMat** graphs.

# Cores	BFS				Connected Components				PageRank			
UPC++	rM	YT	Orkut	Fs	rM	YT	Orkut	Fs	rM	YT	Orkut	Fs
1	1.71	0.08	1.61	0.64	3.11	0.41	9.75	553	0.55	0.31	29.59	690
2	5.32	0.22	0.97	55.6	4.67	0.40	6.45	298	1.07	0.39	6.55	398
4	6.77	0.52	1.26	99.1	9.94	0.42	4.92	175	2.06	0.52	3.87	266
8	11.8	0.88	1.52	7.81	11.7	0.47	3.00	137	2.98	0.81	2.78	265
16	17.1	1.51	1.82	11.6	11.9	0.48	2.45	106	4.57	0.98	2.47	318
32	14.5	1.97	2.71	17.8	12.1	0.50	1.93	76.1	6.03	1.34	2.18	312
32 SU	0.12	0.04	0.59	0.24	0.26	0.82	5.05	7.27	0.09	0.23	13.6	2.21

Table 3: Strong scaling runtime (in seconds) on the NERSC supercomputer. **32 SU** is the speedup achieved by using 32 nodes, calculated as the single-node runtime divided by the 32-node runtime. **rM** is rMat24; **YT** is **YouTube**; **Fs** is **Friendster**. We are unable to run Bellman-Ford on large graphs due to technical difficulties.

5 Processing System

Motivated by the similar overall structure of all our graph algorithms and by the design of Gemini's API [6], we implement a UPC++-based distributed graph processing system that automates the graph loading, sparse-dense frontier management, and terminating conditions. The user only needs to provide the specific interaction logic within a sparse step and a dense step. The system exports a **Graph** class that supplies a constructor and a method, defined in Algorithm 11.

Algorithm 11 Distributed graph processing system in UPC++

```
1:  procedure GRAPH(path)
        ▹ Loads and partitions a graph amongst nodes from a path.
2:
3:  procedure GRAPH.COMPUTE(initD, initFrontier, sparseOp, denseOp)
        ▹ Computes a value array D in sparse and/or dense rounds until the frontier is empty
        ▹ Initializes D and frontier
        ▹ Each round, iterates the entire sparse/dense frontier and applies sparseOp/denseOp to each edge connected to the frontier
```

Armed with this API, we can easily simplify our UPC++ graph algorithms. Algorithm 12 gives an example of such a rewriting for BFS.

Algorithm 12 BREADTHFIRSTSEARCH on our processing system

```
1:  procedure BREADTHFIRSTSEARCH(G, v)
2:      procedure INITD(i)
3:          if i = v then
4:              return ∞
5:          else
6:              return 0
7:
8:      procedure INITFRONTIER(i)
9:          return i = v
10:
11:     procedure SPARSEOP(DNext, FrontierNext, src, dst, round)
12:         if DNext[src] = ∞ then
13:             DNext[dst] ← round
14:             FrontierNext[dst] ← dst
15:
16:     procedure DENSEOP(DNext, FrontierNext, src, dst, round)
17:         if DNext[src] = ∞ then
18:             DNext[src] ← round
19:             FrontierNext[dst] ← dst
20:
21:     G.COMPUTE(INITD, INITFRONTIER, SPARSEOP, DENSEOP)
```

6 Conclusion

In this work, we investigate the potential of porting shared-memory parallel algorithms into UPC++ by implementing a suite of parallel graph algorithms in OpenMP and UPC++ and extensively testing their scaling performance. Our experiments show that performance and scalability often greatly depend on a combination of input data, algorithms, and platform. Though distributed parallel computing requires large input data and dense computation to be efficient, and shared-memory parallelism is preferable when data fits in memory, UPC++ still proves itself a powerful library despite our simple porting. Its well-designed API, especially when compared to the dated counterpart in OpenMPI, and global memory model render implementing algorithms intuitive.

For future work, a closer analysis at the cost of each communication and computation step of our algorithms can yield more insight. Moreover, one can implement highly-optimized OpenMPI-based distributed graph algorithms in UPC++ and compare the two versions as an evaluation of UPC++'s potential to achieve state-of-the-art performance.

Acknowledgement

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH1.

This research is conducted through MIT PRIMES, under the close mentorship of Yan Gu. Two other researchers, Julian Shun and Changwan Hong, have provided tremendous guidance along the way. Without them or MIT PRIMES, none of this work would have been possible.

References

[1] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software.

[2] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. Upc++: a pgas extension for c++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1105–1114. IEEE, 2014.

[3] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L Chamberlain, Jonathan Cohen, Zachary Devito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, et al. Exploring traditional and emerging parallel programming models using a proxy application. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 919–932. IEEE, 2013.

[4] OpenMP Architecture Review Board. OpenMP application program interface version 5.0, 2018. URL http://www.openmp.org/mp-documents/spec30.pdf.

[5] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[6] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 301–316, 2016.

[7] J Bachan, S Baden, Dan Bonachea, P Hargrove, S Hofmeyr, M Jacquelin, A Kamil, and B Van Straalen. Upc++ v1. 0 programmer's guide, revision 2019.9. 0. 2019.

[8] Upc installation, Sep 2019. URL https://bitbucket.org/berkeleylab/upcxx/wiki/INSTALL.

[9] John Bachan, Scott B Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H Hargrove, and Hadia Ahmed. Upc++: A high-performance communication framework for asynchronous computation. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 963–973. IEEE, 2019.

[10] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[11] Julian E. Shun, Guy T. Blelloch, Jeremy B. Fineman, Phillip Vardhan Gibbons, Aapo undefined Kyrola, Harsha undefined Simhadri, and Kanat undefined Tangwongsan. Brief announcement: The problem based benchmark suite. *Proc. ACM Symposium on Parallelism in Algorithms and Architecture (SPAA)*, Jun 2012. doi: 10.1145/2312005.2312018.

[12] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. *Proceedings of the 2004 SIAM International Conference on Data Mining*, 2004. doi: 10.1137/1.9781611972740.43.