Graph Algorithms in PGAS: Chapel and UPC++

Louis Jenkins
louis.jenkins@rochester.edu
University of Rochester
Rochester, NY, USA

Jesun Sahariar Firoz, Marcin Zalewski, Cliff Joslyn, Mark Raugas
{jesun.firoz,marcin.zalewski,cliff.joslyn,mark.raugas}@pnnl.gov
Pacific Northwest National Laboratory
Seattle, Washington, USA

***Abstract*—The Partitioned Global Address Space (PGAS) programming model can be implemented either with programming language features or with runtime library APIs, each implementation favoring different aspects (e.g., productivity, abstraction, flexibility, or performance). Certain language and runtime features, such as collectives, explicit and asynchronous communication primitives, and constructs facilitating overlap of communication and computation (such as futures and conjoined futures) can enable better performance and scaling for irregular applications, in particular for distributed graph analytics. We compare graph algorithms in one of each of these environments: the Chapel PGAS programming language and the the UPC++ PGAS runtime library. We implement algorithms for breadth-first search and triangle counting graph kernels in both environments. We discuss the code in each of the environments, and compile performance data on a Cray Aries and an Infiniband platform. Our results show that the library-based approach of UPC++ currently provides strong performance while Chapel provides a high-level abstraction that, harder to optimize, still provides comparable performance.**

I. Introduction

The Partitioned Global Address Space (PGAS) distributed memory programming model provides a strong shared-memory abstraction. PGAS can be implemented at the language-level, as in the Chapel programming language [1], or as a third-party library and runtime systems like Unified Parallel C++ (UPC++) [2], Legion [3], or HPX [4]. Language-based approaches provide greater productivity via language constructs and high-level abstractions. Libraries and runtime systems provide greater flexibility through low-level and explicit communication primitives scheduling policies. These can maximize both performance and scalability, while also making it easier to adapt new runtime systems into existing code.

In this paper, we compare the Chapel programming language and the UPC++ runtime library for implementing graph algorithms. Graph algorithms are mostly communication-bound, having a low computation-to-communication ratio. They perform irregular remote memory accesses and demonstrate fine-grained computation. Graph algorithms thus require strong runtime and language support for efficient implementation. For example, a good runtime can provide message aggregation and caching, helping to utilize bandwidth at the cost of latency.

From the runtime perspective, for UPC++, we are interested in how the one-sided communication primitives and the completion notification mechanism for asynchronous remote-memory access (RMA) can be used in graph computation. Conjoining and chaining of such completion events, together with one-sided communication primitives, results in better interleaving of communication and computation for graph algorithms. Such mechanisms can reduce straggler effects and can help achieve better scalability for graph algorithms. In Chapel, we strive to explore its high-level, data-parallel, task-parallel, and locality constructs, its implicit one-sided communication, its first-class distributed global-view arrays and domains, and its built-in reductions, with focus on the tradeoff between high productivity and performance.

The contributions of the paper are as follows:

- We leverage the one-sided asynchronous non-blocking communication primitives of UPC++ in conjunction with *future conjoining* to expose overlapping communication and computation in graph algorithms. Such formulation of algorithms naturally leads to better performance.
- We exploit Chapel's first-class arrays, domains, and distributions to handle partitioning of data, discuss ways to overcome the lack of explicit asynchronous non-blocking communication primitives to enable overlap of communication and computation, and discuss patterns to reduce the amount of fine-grained communication to a reasonable amount.
- We compare and contrast both approaches, and evaluate both the raw performance and the amount and quality of code required to model graph computations in both Chapel and UPC++.

II. Graph Algorithms in UPC++

UPC++ [2] is a modern C++-based PGAS library. In UPC++, a part of the distributed memory (shared heap) is considered as the global address space and all processes have access to this shared memory segment via *global pointers*. UPC++ supports one-sided communication through the `rget` and `rput` operations to interact with remote process memory, avoiding any coordination with the remote process. All communication in UPC++ is explicit to make the programmer aware of the cost of communication. UPC++ leverages GASNet-EX [5] networking middleware for communication. Since most UPC++ communication primitives are asynchronous and non-blocking, they facilitate interleaving of communication and computation that are essential for efficient execution of graph algorithms.

The completion of a non-blocking operations in UPC++ (such as an `rget`/`rput`) can be queried on an object called a *future*, which contains the status of an operation as well as the result of the operation. When many asynchronous operations are launched in succession, waiting on individual completion events (such as futures) can be cumbersome.

UPC++ provides a mechanism, called *future conjoining*, that aggregates all such futures and waits on only one future for completion notification. For this, the `upcxx::make_future` mechanism is used to construct an empty future (say `fut_all`). Each asynchronous operation (for example an `rget/rput`) also creates a future, which is then passed to the `upcxx::when_all` function to combine if with `fut_all`. In this way, only a single future handle is obtained to wait on the completion of all the asynchronous operations (rgets/rputs) that have been issued. Future conjoining also minimizes the number of calls to the `wait()` function, further reducing overhead. The closest analogue of UPC++ futures in C++ 11 is std::future, which is a mechanism to access the result of an asynchronous operation.

A. Triangle Counting

We implement a basic **triangle counting** (TC) algorithm to demonstrate the UPC++ asynchronous communication primitives (`rget`) and future conjoining (Listing **??**). Each vertex in the graph obtains its two-hop neighbors in two steps: first an `rget` is issued to get the global pointer to the 2-hop neighbor list (Line 20) from the neighbor's owner rank. Once the `rget` operation finishes, a callback is executed with the `.then` construct (Line 21) to issue the second `rget` operation to fetch the list of two-hop neighbors (Line 27), after a buffer with the right size is allocated. After the two-hop neighbors are obtained, a set intersection is performed between the vertex's immediate neighbors and the fetched two-hop neighbors to determine the closing wedges of the triangles with the common neighbors (Line 31). Finally, a global sum `reduction` of all local triangle counts provides the total triangle counts (Line 46).

There are two important aspects of the implementation of the algorithm. First an empty future is created (Line 5) to conjoin all the futures (Line 39) that will be returned as the completion objects of the outer `rget` operations (Line 20). Once all the `rgets` are issued, the algorithm waits on the conjoining future (Line 44) to signal completion of all the outer `rgets`. Secondly, the inner `rget` operation returns a future to the callback function of the outer `rget` operation (Line 27 and Line 22). Chaining futures as return types to the callback ensure that all inner `rget` operations are completed when the conjoining future returns completion of all outer `rget` operations (Line 44). Through this combination of one-sided communication primitive (`rget`) and `future` conjoining, UPC++ achieves better communication (fetching two-hop neighbors for different vertices) and computation (set intersections) overlap.

B. Breadth-First Search

We also implement a level-synchronous **breadth-first search** (BFS) algorithm in UPC++ (Listing **??**). The algorithm maintains the current and next frontiers of vertices, targeted for each rank, in two distributed queues (Line 12). The distributed queues are allocated on the global heap and each rank has a global pointer to its part. At the beginning of the algorithm, each rank broadcasts the global pointer pointing to the start of its part of the distributed queues (Line 24). These global

```
1  // local triangle count iterator
2  size_t                    local_triangle_count = 0;
3  counting_output_iterator counter(local_triangle_count);
4  // the start of the conjoined future
5  upcxx::future<> fut_all = upcxx::make_future();
6  // For each vertex
7  for (uint64_t i = 0; i < num_vertices_per_rank; i++) {
8   auto vtx_ptr = bases[upcxx::rank_me()].local()[i];
9   auto adj_list_start = vtx_ptr.p.local();
10  auto adj_list_len = vtx_ptr.n;
11  auto current_vertex_id = index_to_vertex_id(i);
12  // For each neighbor of the vertex, first get the
13  // global pointer to the adjacency list and size
14  for (auto j = 0; j < vtx_ptr.n; j++) {
15   auto neighbor = adj_list_start[j];
16   if (current_vertex_id < neighbor) {
17    auto rank   = vertex_id_to_rank(neighbor);
18    auto offset = vertex_id_to_offset(neighbor);
19    upcxx::future<> fut = upcxx::rget(bases[rank] + offset)
20    .then([=] (gptr_and_len pn) {
21     // Allocate a buffer of the same size
22     std::vector<uint64_t> two_hop_neighbors(pn.n);
23     // rget the actual list
24     return upcxx::rget(pn.p,
25       two_hop_neighbors.data(), pn.n)
26     .then([=, two_hop_neighbors =
27       std::move(two_hop_neighbors](){
28       // set intersection
29       std::set_intersection(adj_list_start,
30         adj_list_start + adj_list_len,
31         two_hop_neighbors.begin(),
32         two_hop_neighbors.end(), counter);
33     }); });
34     // conjoin the futures
35     fut_all = upcxx::when_all(fut_all, fut);
36 } } }
37 // wait for all the conjoined futures to complete
38 fut_all.wait();
39 ...
40 auto done_reduction = upcxx::reduce_one(
41    &local_triangle_count, &total_triangle_count, 1,
42    [](size_t a, size_t b) { return a + b; }, 0);
43 done_reduction.wait();
```

Listing 1: Triangle counting in UPC++

pointers will be leveraged by each rank to fetch the next active frontier list destined for the rank.

In each iteration, the algorithm retrieves a vertex from the current active frontier, checks whether it has already been visited (Line 97), and if not, puts all its neighbors in the next frontier queue (Line 107). Neighbors are put into their respective owner rank's buffer in the distributed queue (Line 111). Once the current frontier list traversal is done, each rank initiates two `rget` operations to fetch the next frontier list from other ranks, targeted for the current rank. The first `rget` (Line 66) fetches the corresponding global pointer to remote memory and the length of the frontier list. A buffer of the same length is allocated and a second `rget` operation is issued to obtain the frontier list (Line 69).

These `rget` operations essentially mimic the functionality of an `all_to_all` collective. Here we also utilize future conjoining (Line 78) and chaining of the completion future of the inner rget to the callback function to maximize interleaving of communication and computation. Once each rank finishes receiving the next frontier list, the algorithm proceeds to the next step, a global sum `reduction` is performed on the size of

```
1 // per-destination queues for current and next iteration
2 std::vector<std::vector<std::pair<uint64_t, uint64_t>,
3   upcxxc::allocator<std::pair<uint64_t, uint64_t>>>>
4 frontiersQarr[2];
5 std::vector<std::pair<uint64_t, uint64_t>>
6 received_buf;
7 struct gptr_and_len_pair {
8  // pointer to first element in destination buffer
9  upcxx::global_ptr<std::pair<uint64_t, uint64_t>> p;
10  int n;    // number of elements
11 };
12 using BaseQType = std::vector<upcxx::global_ptr<
13                              gptr_and_len_pair>>;
14 // Two frontiers, alternating current and next
15 BaseQType gpFrontsQarr[2];
16 // access current queue
17 auto level                 = 0;
18 auto current_queue_index = [&]() { return level % 2; };
19 auto gpCurFrontQ = [&]() -> auto& {
20   return gpFrontsQarr[current_queue_index()];
21 };
22 // initialize and exchange
23 for (auto i = 0; i < 2; ++i) {
24  gpFrontsQarr[i].resize(upcxx::rank_n());
25  gpFrontsQarr[i][upcxx::rank_me()] =
26  upcxx::new_array<gptr_and_len_pair>(upcxx::rank_n());
27  for (int r = 0; r < upcxx::rank_n(); r++) {
28   gpFrontsQarr[i][r] =
29   upcxx::broadcast(gpFrontsQarr[i][r], r).wait();
30  }
31 }
32 boost::dynamic_bitset<> color_map(num_vertices_per_rank);
33 std::vector<uint64_t>   parent_map(num_vertices_per_rank);
34 auto curFrontQ       = [&]() -> auto& {
35   return frontiersQarr[current_queue_index()];
36 };
37 while (true) {
38  auto localFrontSize = 0;
39  for(upcxx::intrank_t r = 0; r < upcxx::rank_n(); r++) {
40   // sort and remove duplicates
41   std::sort(curFrontQ()[r].begin(), curFrontQ()[r].end()
42    ,[](auto& a, auto& b) { return a.first < b.first; });
43   std::unique(curFrontQ()[r].begin(), curFrontQ()[r].end()
44    ,[](auto& a, auto& b) { return a.first == b.first; });
45   // Copy the neighbor list per rank to the dist obj
46   gptr_and_len_pair pn;
47   auto destination_buffer_size = curFrontQ()[r].size();
48   pn.n = destination_buffer_size;
49   pn.p = upcxx::try_global_ptr(curFrontQ()[r].data());
50   gpCurFrontQ()[upcxx::rank_me()].local()[r] = pn;
51   // reduce local frontier sizes
52   localFrontSize += curFrontQ()[r].size();
53  }
54  // Reduce to check whether we reached the end
55  auto totalFrontSize = 0;
56  auto done_reduction =
57  upcxx::reduce_all(&localFrontSize, &totalFrontSize, 1,
58  [](size_t a, size_t b) { return a + b; });
59  done_reduction.wait();
60  received_buf.resize(0);
61  // the start of the conjoined future
62  upcxx::future<> fut_all = upcxx::make_future();
63  if (totalFrontSize == 0) break;
64  // Get vertices targeted for me from each rank
65  for (upcxx::intrank_t r = 0; r < upcxx::rank_n(); r++) {
66   upcxx::future<> fut =
67   upcxx::rget(gpcurFrontQ()[r] + upcxx::rank_me())
68    .then([=](gptr_and_len_pair pn) {
69     return upcxx::rget(pn.p,
70       target_neighbor_list.data(), pn.n)
71     .then([=, target_neighbor_list =
72       std::move(target_neighbor_list)]() {
73       received_buf.insert(received_buf.end(),
74       target_neighbor_list.begin(),
75       target_neighbor_list.end());
76   }); });
77   // conjoin the futures
78   fut_all = upcxx::when_all(fut_all, fut);
79  }
80  // wait for all the conjoined futures to complete
81  fut_all.wait();  level += 1;
82  for (upcxx::intrank_t r = 0; r < upcxx::rank_n(); r++) {
83   curFrontQ()[r].resize(0);
84  }
85  // next frontier: sort and remove duplicates
86  std::sort(received_buf.begin(), received_buf.end(),
87  [](auto& a, auto& b) { return a.first < b.first; });
88  std::unique(received_buf.begin(), received_buf.end(),
89  [](auto& a, auto& b) { return a.first == b.first; });
90  for (auto vertex_p : received_buf) {
91   auto vtx = vertex_p.first;
92   auto parent = vertex_p.second;
93   // Check whether the vertex has already been visited.
94   auto v_index = vertex_id_to_index(vtx);
95   if (color_map[v_index]) {
96    continue; //Already visited
97   } else {    // Not visited yet
98    color_map.set(v_index);
99    parent_map[v_index] = parent;
100    // Put all its neighbors into the frontier
101    auto vtx_ptr        =
102      bases[upcxx::rank_me()].local()[v_index];
103    auto adj_list_start = vtx_ptr.p.local();
104    auto adj_list_len   = vtx_ptr.n;
105    for (auto j = 0; j < vtx_ptr.n; j++) {
106     auto neighbor     = adj_list_start[j];
107     auto neighborRank = vertex_id_to_rank(neighbor);
108     curFrontQ()[neighborRank].push_back(
109     std::make_pair(neighbor, vtx));
110 } } } } // end while
```

Listing 2: Breadth-first search in UPC++

the next frontier queue to detect termination (Line 56).

III. Graph Algorithms in Chapel

Chapel [1], [6] is Cray's exascale programming language that emerged from DARPA's High Productivity Computing Systems (HPCS) challenge. Chapel takes PGAS philosophy to its very limit, focusing heavily on enabling greater productivity by abstracting the sense of locality and the need to think from the perspective of individual processors. A processing element in Chapel is known as a *locale*. Chapel also provides *locale models* that model the underlying topology of the runtime devices. We utilize Chapel's default flat locale model in which locales provide uniform access to all system resources. Also available are the NUMA locale model, which maps *sublocales* to NUMA domains, and the KNL locale model, which support accessing high-bandwidth memory on Knights Landing and Xeon Phi processors. Furthermore, in contrast to UPC++, we use one process per compute node in Chapel where data structures are shared between worker threads.

In Chapel a *task* is a set of coroutines that are managed by the *tasking layer*, which handles scheduling tasks to threads. Chapel's execution begins with an initial task, and all threads (local or remote), except the one the initial task is multiplexed on top of, wait for instructions from the initial task in a thread

```
iter roundRobin(D : domain, param tag : iterKind)
  where tag == iterKind.standalone {
 // Spawn one task per locale
 coforall loc in Locales do on loc {
  // Spawn on task per core
  coforall tid in 0..#here.maxTaskPar {
   // Compute this task's slice
   const localD = D.localSubdomain();
   const localStride = here.maxTaskPar;
   const localAlignment = localD.stride * tid
     + localD.alignment;
   const localRange = localD by localStride
     align localAlignment;
   for v in localRange do yield v;
} } }
var numTriangles : int;
forall v in roundRobin(A.domain)
  with (+ reduce numTriangles) {
 forall u in A[v].these(ignoreRunning=true) do if v > u {
  numTriangles += intersectionSize(A[v], A[u]);
} }
numTriangles /= 3;
```

Listing 3: Triangle Counting in Chapel

pool managed by the tasking layer. This has the benefit of greatly simplifying writing distributed memory programs, as it resembles most shared-memory programming languages.

Chapel takes emulating shared memory a step further by enabling *global-view programming*. This is a programming paradigm that, unlike UPC++, enables (but does not require) users to think about computations from a level above each individual task. The compiler enables remote memory access (RMA) via lexical scoping and task migration: migrating a task from one locale to another, and then accessing local variables in the parent scope, results in implicit PUTs and GETs. All communication is handled by a *communication layer* (currently limited to GASNet and uGNI for Cray-specific hardware).

Where communication is explicit and asynchronous in UPC++, in Chapel it is both implicit and synchronous, with the compiler and runtime transforming accesses to remote memory into PUTs and GETs, enabling algorithms to be written that function and perform well in both shared and distributed memory contexts. Chapel enables very rich data-parallel constructs, such as: seamless parallel and distributed iteration, reductions and scans, and native built-in support for serial/parallel and local/distributed computation (locality and parallelism are two disentangled concepts in Chapel).

A. Triangle Counting

In our UPC++ TC implementation, the vertices are cyclically distributed across all ranks, with one rank per processor. To emulate this in Chapel (Listing **??**), we utilize one of Chapel's *distributions*, which handle mapping indices in the domain to locales. Of the distributions available, such as *Block*, *BlockCyclic*, and *Cyclic*, we choose a Cyclic distribution so that the vertices are mapped cyclically across all locales. Where Chapel's *for* loops are the standard serial iterators, *forall* loops are implemented using the *leader-follower* framework [7], where the *leader* task creates *follower* tasks and divides the work among them. As the leader-follower iterator for Chapel's distributed arrays and domains are implemented by statically

```
// Replicates pairs of an array and a lock on each locale.
var globalWorkDom = {0..1} dmapped Replicated();
var globalLocks : [globalWorkDom] Lock;
var globalWork : [globalWorkDom] Array(int);
var globalWorkIdx : int;
// Insert root into work queue.
on A[0] do globalWork[globalWorkIdx].append(0);
var visited : [A.domain] atomic bool;
while true {
 var pendingWork : bool;
 // Spawn one task per locale (SPMD)
 coforall loc in Locales with (|| reduce pendingWork)
   do on loc {
  // Local work queues to be reduced into
  var localeLock : [LocaleSpace] Lock;
  var localeWork : [LocaleSpace] Array(int);
  ref workQueue = globalWork[globalWorkIdx];
  removeDuplicates(workQueue);
  // One task per core, even chunks of work
  coforall chunk in chunks(0..#workQueue.size,
                           numChunks=here.maxTaskPar)
   with (|| reduce pendingWork) {
   // Task-local work queue
   var localWork : [LocaleSpace] Array(int);
   for v in workQueue[chunk] {
    // Check if vertex has been visited
    if visited[v].testAndSet() == false {
     pendingWork = true;
     for neighbor in A[v] {
      localWork[A[neighbor].locale.id].append(neighbor);
   } } }
   // Reduce into local work queue
   for (lock, _localeWork, _localWork)
     in zip(localeLock, localeWork, localWork) {
    lock.acquire();
    _localeWork.append(_localWork);
    lock.release();
  } }
  // Perform a global reduction into other work queues
  coforall loc in Locales do on loc {
   globalLocks[globalWorkIdx].acquire();
   globalWork[(globalWorkIdx + 1) % 2]
     .append(localeWork[here.id]);
   globalLocks[globalWorkIdx].acquire();
  }
  globalWork[globalWorkIdx].clear();
 }
 // Swap work queue and check if empty
 globalWorkIdx = (globalWorkIdx + 1) % 2;
 if !pendingWork then break;
}
```

Listing 4: Breadth-First Search in Chapel

dividing the iteration space of the local subdomain of each locale into evenly sized chunks, we cyclically distribute the vertices among tasks via the *roundRobin* iterator to match UPC++. Chapel's *coforall* spawns one task per iteration, and its *on* construct migrates the current task to the requested locale. By application of Chapel's built-in *reduction intent*, which performs a global reduction across each task and locale in the user program, we can quickly obtain the number of triangles via the size of intersection of the neighbor list of each vertex and its neighbor's neighbor list.

To emulate UPC++'s asynchronous communication primitives, we intentionally oversubscribe the system to allow Chapel's tasking layer to handle load balancing, and to allow overlap of communication with computation where possible. The uGNI communication layer will yield the current task

Graph	$\|V\|(M)$	$\|E\|(M)$	$\mathbb{E}(\cdot)$	$\sigma(\cdot)$	Source
Friendster	65.0	1,800.0	55.0	138.0	[9]
com-LiveJournal	4.0	34.6	17.0	43.0	[8]
as-skitter	1.6	11.0	6.0	15.0	[9]
com-Youtube	1.1	3.0	5.0	51.0	[8]
Graph500-scale21	2.0	67.0	51.0	389.0	[8]
Graph500-scale22	4.0	130.0	54.0	460.0	[8]
Graph500-scale23	8.0	260.0	56.0	543.0	[8]
Graph500-scale24	16.0	540.0	59.0	639.0	[8]
Graph500-scale25	32.0	1,000.0	61.0	750.0	[8]

TABLE I: Datasets used for experiments: # vertices and edges in millions; $\mathbb{E}(\cdot) = \mathbb{E}(\deg(v))$ is the mean vertex degree; $\sigma(\cdot) = \sigma(\deg(v))$ is its standard deviation; source is the citation for the data set. Top group: strong scaling. Bottom group: weak scaling.

while waiting for communication to complete, allowing for the additional tasks from the oversubscription to run. The GASNet communication layer is not capable of this, as it will make blocking calls to GASNet such as *gasnet_get* and *gasnet_put* which will block not just the task, but the thread it is multiplexed on. This prevents overlap, while still allowing automatic load balancing.

B. Breadth-First Search

For BFS (Listing **??**), we replicate two pairs of *Arrays*, a simple dynamic array similar to C++'s Vector that serves as our work queue, and *Lock*, a simple test-and-test-and-set spinlock. To determine whether or not a vertex has been visited, we maintain an array of atomic Booleans that is distributed over the same domain as the array of vertices, ensuring that the atomics are always local to it's corresponding vertex. After inserting the root into the work queue, we enter our work loop where we spawn a task per locale and create an array mapped over *LocaleSpace*. This creates an Array and corresponding Lock for each destination locale, essentially performing our own aggregation.

At each level, we remove duplicates from the current work queue in a similar way to UPC++, where we sort the array via a parallel radix sort (using the Chapel's *Sort* module) and mutate the array to eliminate all duplicates. We then evenly distribute work across tasks on the current locale, where each task maintains its own aggregation buffer of outgoing work. After determining that we have not visited a node, we not only denote that we have *pendingWork* (so the loop is not terminated), but we also push the neighbor of the current vertex to a destination buffer. These task-specific destination buffers are then reduced into the locale-specific destination buffer, which in turn gets reduced into the global work queue to be used in the next phase.

The pendingWork flag is propagated in a reduction across all tasks and locales, so if any single task pendingWork, it will be known globally. Then based on whether or not there is pendingWork, the loop either continues or terminates. Manual reduction is performed for non-scalar types for efficiency reasons, although it is possible to implement a custom reduction operation for it, it is unfortunately not well-optimized.

IV. Experimental Results

Our benchmarks were run on two separate systems with different hardware and configurations: 1) up to 64 nodes of a Cray-XC50 cluster with Cray Aries network and Intel Broadwell 44-core compute nodes, and 2) up to 16 nodes of an Infiniband cluster with Intel Xeon 20-core compute nodes. In Chapel, programs are compiled with *–fast* flag, which ensures that all optimizations are enabled in both the Chapel compiler as well as the backend used (GCC).

A. Strong Scaling

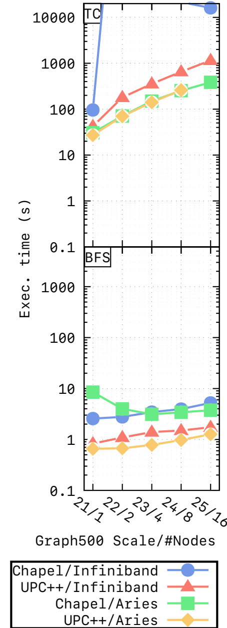


Fig. 2: Weak scaling results with Graph500.

The data sets used for strong scaling measurements are listed in the top portion of Table I. These were selected to highlight different data properties, which can be seen in the results for both TC and BFS (Figure 1a and Figure 1b, where missing points are for runs which didn't complete). For BFS, Chapel is within 25% to near-equivalent with UPC++ on both Aries and Infiniband, thanks to message aggregation via destination buffering.

B. Weak Scaling

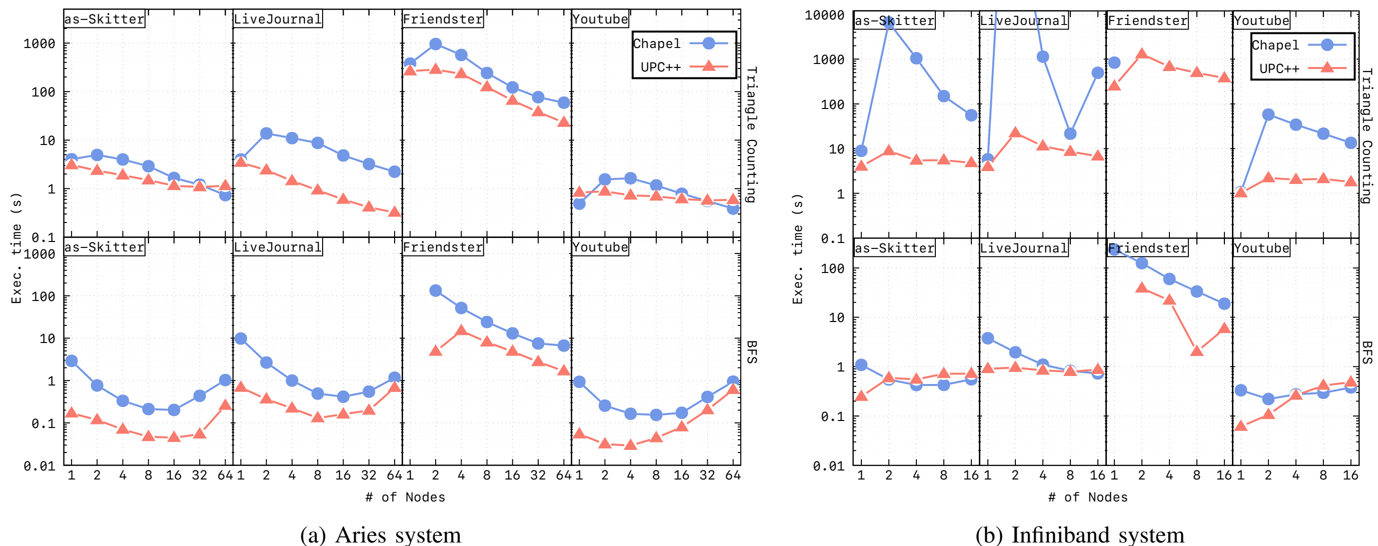
Weak scaling was tested on synthetic graphs from Graph500 (bottom of Table I), where scale 21 runs with one compute node, and scale 22 with two. From Figure 2, we can see that for BFS, both Chapel and UPC++ are comparable, both on Aries and Infiniband. Due to the coarser granularity of the communication involved in this benchmark compared to TC, we see that Chapel has the potential to match bare-metal hand-optimized algorithms, while maintaining high-level semantics.

For TC, Chapel and UPC++ are comparable, we believe due to extremely high variance in the synthetic dataset causing load-imbalance. Chapel, when run using an Infiniband conduit via GASNet, is at least two orders of magnitude slower than with the native Aries network, but its performance does seem to improve as the node count increases.

V. Discussion

UPC++ enables programmers to have fine-grained control by providing access to low-level messaging and communication primitives. More importantly, flexible completion notifications of the execution of non-blocking primitives (i.e., with future conjoining) streamlines interleaving communication and computation that is needed by graph algorithms.

We can identify a number of opportunities for potential performance improvements in UPC++. For example, currently UPC++ does not provide an *all_to_all* collective. In this work, we manually implement its functionality for exchanging frontiers in the BFS algorithm. The UPC++ development team



(a) Aries system

(b) Infiniband system

Fig. 1: Strong scaling results with real-world graph dataset.

indicated that a library implementation of *all_to_all* is in their future road map for 2021. Such a built-in primitive will ease future implementation of graph algorithms. Asynchronous graph algorithms in UPC++ may also prove fruitful. Semantically these algorithms are label-correcting (in contrast to the label-setting algorithms in our present discussion) and will eliminate the requirement for global synchronization barriers. UPC++ provides a construct, `rpc_ff` (RPC fire and forget), which can be a promising way to implement such algorithms. However, this will require implementation of a termination detection algorithm (e.g., Sinha-Kale-Ramkumar) to detect global quiescence, either at the library or application level.

On Chapel's side, there is a lack of portability of performance between Cray Aries and Infiniband systems, which is due to the bias towards Cray systems by Cray's Chapel development team. Currently, support for the GASNet communication layer is being tapered in favor of the development of a new libfabric [**?**] communication layer that will support both GNI and Infiniband interconnects. As well, the lack of non-blocking communication primitives does make it extremely difficult to match UPC++, but using oversubscription of tasks as a work-around can suffice.

More benchmarks should be conducted to study precisely *why* UPC++ and Chapel vary so much in performance, especially since Chapel performance varies based on the choice of network provider. Such studies would benefit from beginning with a single-node SMP conduit with controlled number of threads to isolate language feature and compiler maturity versus network-based influences on performance. Then network effects might be more systematically taken into account.

There are both performance and productivity differences between UPC++ and Chapel for irregular algorithms. UPC++ codes in general are more verbose and mastering the use of asynchronous models of computation, especially the powerful conjoined futures feature, is non-trivial. However, C++ developers can come up to speed in the core library relatively quickly. Chapel has a more succinct syntax, but a learning curve of its own, and the core language is still evolving. Optimizing Chapel codes for irregular applications requires greater knowledge of the language and runtime internals than may be practical to expect of new users of the language.

Acknowledgement

We thank John Bachan, Scott Baden, Dan Bonachea for their valuable help with UPC++. We thank Brad Chamberlain, the technical lead of the Chapel, for his assistance in closing the performance gap between UPC++ and Chapel. We thank Cray for providing access to a Cray-XC50 computer. This work was supported by the High Performance Data Analytics (HPDA) program at Pacific Northwest National Laboratory.

References

[1] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[2] UPC++: a PGAS library for c++. https://bitbucket.org/berkeleylab/upcxx/src/master/. Accessed: 2019-05-20.

[3] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[4] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[5] Dan Bonachea and Paul H Hargrove. Gasnet-ex: A high-performance, portable communication library for exascale. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2018.

[6] et. al Chamberlain, Bradford L. Chapel comes of age: Making scalable programming productive. 2007.

[7] Bradford L Chamberlain, Sung-Eun Choi, Steven J Deitz, and Angeles Navarro. User-defined parallel zippered iterators in chapel. In *Proceedings of Fifth Conference on Partitioned Global Address Space Programming Models*, volume 2011, pages 1–11, 2011.

[8] DARPA HIVE graphchallenge dataset. http://graphchallenge.mit.edu/data-sets. Accessed: 2019-05-20.

[9] Suitesparse matrix collection. https://sparse.tamu.edu/. Accessed: 2019-05-20.