Lawrence Berkeley National Laboratory
Recent Work

Title
UPC++: A high-performance communication framework for asynchronous computation

Permalink
https://escholarship.org/uc/item/1gd059hj

ISBN
9781728112466

Authors
Bachan, J
Baden, SB
Hofmeyr, S
et al.

Publication Date
2019-05-01

DOI
10.1109/IPDPS.2019.00104

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

UPC++: A High-Performance Communication Framework for Asynchronous Computation

John Bachan[1] Scott B. Baden[1] Steven Hofmeyr[1] Mathias Jacquelin[1]
Amir Kamil[1,2] Dan Bonachea[1] Paul H. Hargrove[1] Hadia Ahmed[1]
[1] *Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA*
[2] *University of Michigan, Ann Arbor, MI 48109, USA*
pagoda@lbl.gov

***Abstract*—UPC++ is a C++ library that supports high-performance computation via an asynchronous communication framework. This paper describes a new incarnation that differs substantially from its predecessor, and we discuss the reasons for our design decisions. We present new design features, including future-based asynchrony management, distributed objects, and generalized Remote Procedure Call (RPC).**

We show microbenchmark performance results demonstrating that one-sided Remote Memory Access (RMA) in UPC++ is competitive with MPI-3 RMA; on a Cray XC40 UPC++ delivers up to a 25% improvement in the latency of blocking RMA put, and up to a 33% bandwidth improvement in an RMA throughput test. We showcase the benefits of UPC++ with irregular applications through a pair of application motifs, a distributed hash table and a sparse solver component. Our distributed hash table in UPC++ delivers near-linear weak scaling up to 34816 cores of a Cray XC40. Our UPC++ implementation of the sparse solver component shows robust strong scaling up to 2048 cores, where it outperforms variants communicating using MPI by up to 3.1x.

UPC++ encourages the use of aggressive asynchrony in low-overhead RMA and RPC, improving programmer productivity and delivering high performance in irregular applications.

***Index Terms*—Asynchronous, PGAS, RMA, RPC, Exascale.**

I. Introduction

UPC++ [1, 2, 3] is a C++ library that supports Partitioned Global Address Space (PGAS) programming. An early predecessor of UPC++ was released in 2012 as v0.1 [4]. This paper describes a new incarnation of UPC++ (v1.0) with a very different design, which is tailored to meet the needs of exascale applications that require PGAS support[1]. There are three main principles behind the redesign of UPC++. First, wherever possible, *operations are asynchronous* by default, to allow the overlap of computation and communication, and to encourage programmers to avoid global synchronization. Second, all *data motion is explicit*, to encourage programmers to consider the costs of communication. Third, UPC++ encourages the use of *scalable data-structures* and avoids non-scalable library features. All of these principles are intended to provide a programming model that can scale efficiently to potentially millions of processors.

Like other PGAS models, UPC++ supports physically distributed global memory, which can be referenced via special global pointers. Using global pointers, processes can copy data between their local memory and remote global memory using one-sided Remote Memory Access (RMA) operations. Unlike pointer-to-shared in UPC [5], UPC++ global pointers cannot be directly dereferenced, as this would violate our principle of making all communication syntactically explicit. In addition to RMA, UPC++ also supports Remote Procedure Calls (RPCs), whereby the caller can induce a remote process to invoke a user function, including any arguments and generating an optional result for return to the sender.

In this paper, we provide an overview of the UPC++ programming model and describe its implementation over the GASNet-EX communication library [6], which delivers a low-overhead, high-performance runtime. Using microbenchmarks, we demonstrate that the bandwidth and latency achieved by UPC++ RMA operations are competitive with MPI one-sided transfers. Finally, we present results for two application motifs that exemplify the advantages of UPC++, and more generally, PGAS programming. The first motif is a distributed hash table, where latency is a limiting factor; it benefits from the synergistic interaction of RPC with RMA. The second is an extend-add operation that is a crucial building block of sparse solvers; it relies heavily on RPC and benefits from UPC++ mechanisms for asynchrony.

II. UPC++ Programming Model

A UPC++ program running on a distributed-memory parallel computer can be viewed as a collection of *processes*, each with *local memory* (see Fig. 1). UPC++ implements SPMD parallelism, with a fixed number of processes during program execution. Like other PGAS models, UPC++ provides *global memory* that the user program allocates in *shared segments* distributed over the processes. Processes have access to remotely allocated global memory via a *global pointer* and can access their respective local memories via conventional C++ pointer. As with threads programming, references made via global pointers may be subject to race conditions, and appropriate synchronization must generally be employed.

A UPC++ global pointer differs from a conventional C-style pointer in several ways. Most importantly, it cannot be dereferenced with the $\star$ operator, as this would violate the explicit data-motion principle. However like conventional C-style pointers, UPC++ supports arithmetic on global pointers and passing global pointers by value. A global pointer can also

[1] UPC++ is portable beyond HPC systems—it runs on 64-bit Linux and macOS, and can be used on laptops as well as servers.

©2019 IEEE doi:10.25344/S4V88H

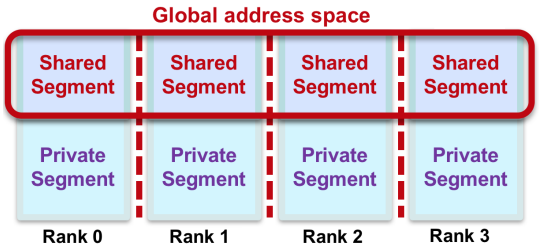


Fig. 1: PGAS logical memory model

be converted to and from a conventional pointer to the same object in the shared segment by the owning process.

Global pointers are used to refer to memory in shared segments when transferring data between processes via *one-sided* RMA communication operations. Global pointers are also used to refer to shared-segment memory locations in atomic operations.

All operations that involve communication are non-blocking and are managed through an API that includes *futures* and *promises*. This is a departure from the previous version of UPC++, which used an event-based model. While both allow the expression of asynchronous operations, futures are a much more powerful and composable abstraction, enabling arbitrary dependency graphs to be constructed. Futures and promises are also more likely to be familiar to C++ programmers, since they were introduced to the C++ standard library in C++11 [7].

Unlike standard C++ futures, UPC++ futures and promises are used to manage asynchronous dependencies within a thread and not for direct communication between threads or processes. A future is the interface through which the status of the operation can be queried and the results retrieved, and multiple future objects may be associated with the same promise. A future thus represents the consumer side of a non-blocking operation. Each non-blocking operation has an associated promise object, which is created either explicitly by the user or implicitly by the runtime when the non-blocking operation is invoked. A promise represents the producer side of the operation, and it is through the promise that the results of the operation are supplied and its dependencies fulfilled. A user can pass a promise to a UPC++ communication operation, which registers a dependency on the promise and subsequently fulfills the dependency when the operation completes. The same promise can be passed to multiple communication operations, and through a single wait call on the associated future, the user can be notified when all the operations have completed.

Futures are also a mechanism for composing asynchronous operations, allowing the construction of elaborate dependence-driven graphs of asynchronously executed operations. A user can *chain* a *callback* to a future via the `.then()` method, and the callback will be invoked on the values encapsulated by the future when they are available. The callback can be a function or a lambda, and it can initiate asynchronous operations of its own. The `.then()` method itself produces a new future, representing the results of the callback, and the new future can then have further callbacks chained on to it. Multiple futures can be *conjoined* by the `when_all()` function template, which produces a single future that represents readiness of all the input futures and their values. A callback can then be chained to the resulting future, allowing the callback to depend on the completion of and the values produced by multiple asynchronous operations.

UPC++ provides several types of communication operations: RMA, RPC, remote atomics and collectives; all are asynchronous and support a flexible set of synchronization mechanisms. RMA implements one-sided communication, and RPC ships a user-provided function or lambda with arguments to a remote processor for execution. On most high-performance networks, RMA operations use network hardware offload support to deliver low latency and high bandwidth. Remote atomics are useful in implementing lock-free data structures, and on network hardware with appropriate capabilities (such as available in Cray Aries) remote atomic updates can also be offloaded, improving latency and scalability [8].

UPC++ has several other powerful features that can enhance programmer productivity and improve performance. For example, UPC++ also supports non-contiguous RMA transfers (vector, indexed and strided), enabling programmers to conveniently express more complex patterns of data movement, such as those required with the use of multidimensional arrays[2].

Unlike UPC, UPC++ does not support distributed shared arrays, partially because these are difficult to implement scalably in a portable way—in general, every process must store at least one base pointer for every other process's shared segment. The other reason, a semantic reason, is that symmetric heaps (the implementation approach to distributed arrays used in UPC, OpenSHMEM [9] and other models) require globally collective allocation that does not compose well with subset processor teams. Instead, UPC++ provides *distributed objects*, which can be used to conveniently represent an object (such as an array) distributed across any subset of processes, without implicitly requiring non-scalable storage anywhere in the software stack to track remote instances of the object. UPC++ RPCs include support to automatically and efficiently translate distributed object arguments between global identifiers and the local representative in each process. Obtaining a global pointer from a remote instance of a distributed object requires explicit communication, in keeping with the principle that there should be no implicit communication in UPC++.

The remainder of this paper will focus on a subset of the UPC++ features: futures, global pointers, RMA and RPC. For a discussion of other features, the reader is referred to the UPC++ *Programmer's Guide* [3].

III. Implementation of the UPC++ Runtime

In this section, we give a simplified overview of how the UPC++ runtime handles asynchronous operations. UPC++ uses GASNet-EX as the communication layer, which handles the

[2]UPC++ does not directly provide a multidimensional-array abstraction, but is compatible with those provided by other packages.

transfer of data over the network and within shared memory. For the purposes of this discussion, we will focus on two categories of data-movement operations provided by GASNet-EX: RMA (get and put), and Active Messages (AM). RMA copies data between local memory and global memory, and an AM invokes a function with an optional payload on a remote process. These operations execute asynchronously, and GASNet-EX provides interfaces to obtain notification when operations complete.

The `UPC++` runtime is responsible for translating `UPC++` communication operations into lower-level GASNet-EX operations, synchronizing their completion, and progressing `UPC++`-level operations. The asynchronous aspects of the operations are managed by the *Progress Engine*, which advances the progress of pending or ongoing communication requests and processes completed requests. `UPC++` does not use special runtime threads to implement progress. Instead, progress is made either during calls into the library (*internal progress*), or explicitly when a user program invokes the `progress` or `wait`[3] calls (*user-level progress*).

The design of `UPC++` avoids introducing hidden threads inside the runtime in order to improve user-visibility into the resource requirements of `UPC++` and to enhance interoperability with software packages that have restrictive threading requirements. The consequence, however, is that the user must be aware of the balance between the need to make progress (via explicit library calls) and the application's need for CPU cycles. This is related to the matter of *attentiveness*, which we will discuss shortly.

The Progress Engine employs three unordered queues for user operations:

defQ	Operations in the *deferred* state, i.e. not yet handed off to GASNet-EX.
actQ	Operations in the *active* state, i.e. already handed off to GASNet-EX.
compQ	Operations in the *complete* state, i.e. they have finished.

When a `UPC++` RMA injection call is made, the runtime creates a promise for the operation and stores the promise in *defQ*, indicating the operation is in the *deferred* state. This promise is used to create the future that is returned by `rget/rput`. The Progress Engine polls *defQ* and initiates the queued operations. When the RMA is handed off to GASNet-EX, the operation enters the *active* state, and its associated promise is placed in *actQ*. The RMA will now complete without initiator *attentiveness*[4], since GASNet-EX has taken over responsibility. When GASNet-EX has finished the RMA, it indicates completion to the `UPC++` runtime, and the next call into internal progress (any `UPC++` communication call) promotes the operation to the *complete* state, at which point it will move the promise to the *compQ* queue. The *compQ* queue is drained only by explicit user-progress. Hence, this queue can be thought of as the list of "futures to satisfy."

[3]The `wait` call is simply a spin loop around `progress`.

[4]No further local CPU resources are required to complete the data transfer.

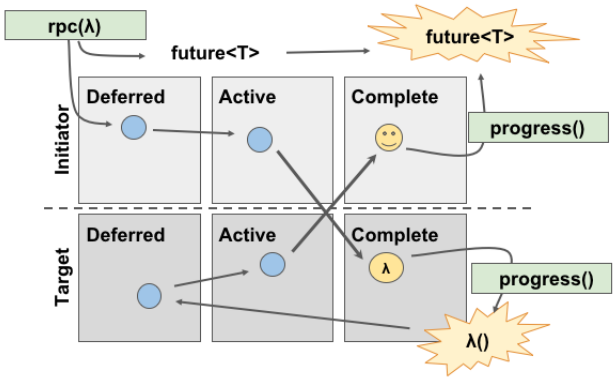


Fig. 2: Progression of RPC through the `UPC++` runtime. Blue and yellow circles are promises.

RPC operations (Fig. 2) progress through both the initiator's and the target's queues. Once the RPC is dispatched from the sender, the `UPC++` runtime places a promise in the initiator's *actQ*, and GASNet-EX moves the payload to the target using AM. On the target's side, the incoming RPC (the lambda or function pointer, plus any arguments) is inserted into the target's *compQ*. The target must be attentive to user-level progress to execute this RPC (e.g., if the target enters intensive, protracted computation without calls to `progress`, incoming RPCs will stall). Once the RPC executes, its return value will be sent through *defQ* and *actQ* back to the initiator side using AM, which updates the corresponding promise and moves it to the initiator's *compQ*[5].

If a program attaches a callback action to a future (via the `.then()` method), that callback is invoked at the initiator after the corresponding future-returning operation completes (this progression is not shown in Fig. 2). Each future maintains a queue of callbacks that are waiting on it, and when the promise associated with an operation is moved to the *compQ*, the callbacks attached to the corresponding future are also placed there. When user progress executes a callback in *compQ*, the promise behind the associated future is satisfied.

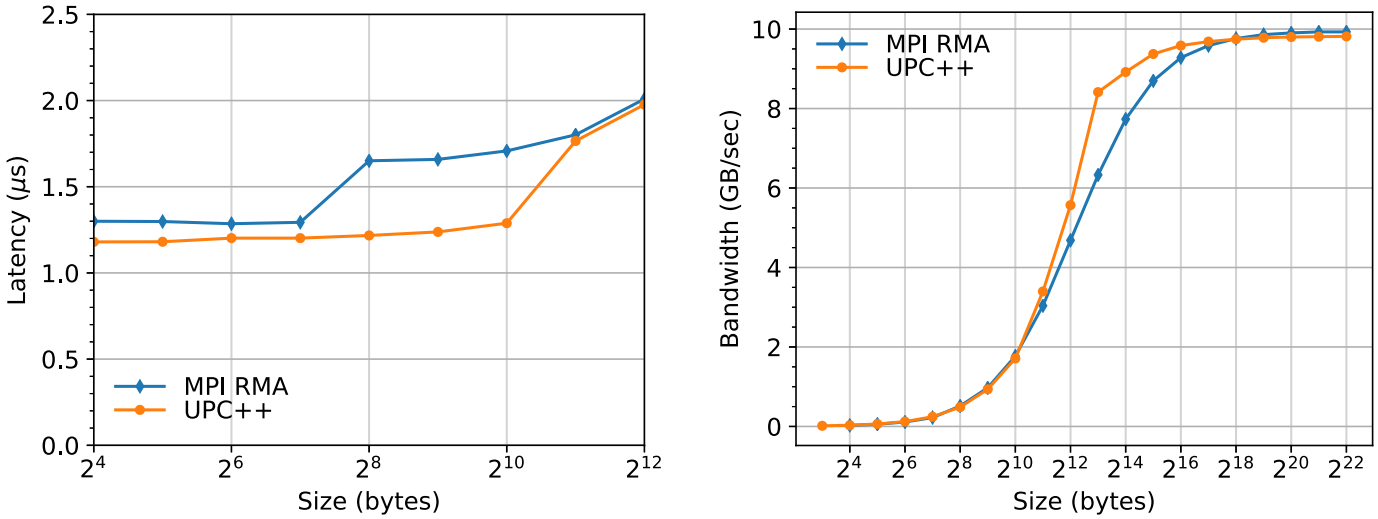
IV. Experimental Results

In this section, we present microbenchmark results showing that the performance of `UPC++` RMA operations is competitive with MPI one-sided RMA. We then describe the code for two application motifs and present performance results.

A. Testbed

All the experiments presented in this paper were run on the Cori Cray XC40 system [10] at NERSC [11]. This system has two disjoint partitions, each interconnected using a Cray Aries network in a Dragonfly topology. *Cori Haswell* has 2,388 compute nodes, each equipped with two 2.3GHz 16-core Intel Xeon E5-2698v3 "Haswell" processors and 128GB of DDR4

[5]`UPC++` provides several variants of RPCs, one of which (`rpc_ff`) does not return anything to the initiator, and hence its progress is more like `rget/rput`.



(a) Round-trip Put Latency (lower is better)

(b) Flood Put Bandwidth (higher is better)

Fig. 3: RMA microbenchmark performance on Cori Haswell.

DRAM. *Cori KNL* has 9688 nodes, and each node has a single 1.4GHz 68-core Intel Xeon Phi 7250 "KNL" processor, 96GB of DDR4 DRAM, and 16GB of on-package high-bandwidth memory. The code used for all experiments was compiled with Cray's `PrgEnv-intel/6.0.4` environment module and version `18.0.1 20171018` of the Intel compiler suite. All UPC++ results use release version 2018.9.0 with GASNet-EX v2018.9.0 aries-conduit [8]. All MPI results presented for comparison used Cray's `cray-mpich/7.6.2` environment module. To ensure proper optimizations for the KNL CPU, executables built for Cori KNL used the `craype-mic-knl` environment module. Other than selection of the appropriate `craype-*` module, all environment modules and settings used the NERSC defaults.

B. Microbenchmarks

To measure bandwidth, we ran a "flood" microbenchmark, which initiates a large number of non-blocking RMA puts of a given size and then waits for them all to complete. The bandwidth metric is then the total volume of data transferred, divided by the total elapsed time. For `UPC++`, we issue multiple `rput` operations in a loop and use promises to track completion, as shown in the following code outline:

```
upcxx::promise<> p; // for tracking completion
while (iters--) {
  // one-sided put to remote memory location
  // dest is a global pointer
  upcxx::rput(src, dest, size,
              upcxx::operation_cx::as_promise(p));
  // occasional progress
  if (!(iters % 10)) upcxx::progress();
}
// wait for all rputs to complete
p.finalize().wait();
```

In the second experiment, we measured latency by issuing a large number of blocking transfers of a given size, i.e. the microbenchmark waits for each `upcxx::rput` operation to complete (which includes waiting for a network-level acknowledgment from the target) before issuing a new one. The latency metric is then the average time taken to transfer a given size. The code outline is simple, using a `wait` on a future:

```
while (iters--)
  // issue one rput, wait for completion
  upcxx::rput(src, dest, size).wait();
```

For comparison, we also ran semantically analogous MPI-3 one-sided benchmarks from the Intel MPI Benchmarks suite [12] (IMB), version `v2018.1`. The flood bandwidth of the `MPI_Put` function was measured using the *aggregate* timings of the `Unidir_put` microbenchmark from the `IMB-RMA` test, whereas the latency microbenchmark used the *non-aggregate* timings. Both tests use a passive-target access epoch and synchronize using `MPI_Win_flush`.

All tests were run between two compute nodes of Cori Haswell[6], using a single process per node, i.e. one initiator and one passive target. Each data point shows the best result obtained across 10 different batch jobs for that configuration, with the `UPC++` and MPI RMA tests run back-to-back within each batch job. It can be seen in Fig. 3 that for sizes less than 256 bytes the latency of `UPC++` is better than that of MPI RMA by more than 5% on average, and from 256 to 1024 bytes the improvement averages more than 25%. This latency advantage is present though at least 4MB (the largest we measured). The bandwidths are comparable for small and large sizes, but differ in the range between 1KB and 256KB. This difference is most pronounced at 8KB, where `UPC++` is delivering over 33% more bandwidth than MPI RMA.

[6]The same experiments on Cori KNL reveal a known performance anomaly in Cray MPI RMA on this platform. Since this would imply a non-representative advantage for UPC++, we have omitted those results. However, the remaining benchmarks in this paper *do not* use MPI RMA.

C. Distributed Hash Table

In our first example application motif, we show how to implement a distributed hash table that scales efficiently to large numbers of processes.[7] In the simplest implementation, each process has its own local map:

```
std::unordered_map<string, string> local_map;
```

The insert operation uses RPC to update the local map at the target[8]:

```
upcxx::future<> insert(const string &key,
                       const string &val) {
  return upcxx::rpc(get_target(key),
                    [](string key, string val) {
                      local_map.insert({key, val});
                    }, key, val);
}
```

The target process is determined by a hash function `get_target` (not shown) that maps a key to a unique process. The data is inserted using a lambda function, which takes a key and a value as parameters. The insert operation returns the result of the lambda, which in this case is an empty future. An example use of this asynchronous insert operation:

```
upcxx::future<> f = insert("Germany", "Bonn");
f.wait();
```

Although an RPC-only distributed hash table will work, we can improve the performance for larger value sizes by taking advantage of the zero-copy RMA provided by UPC++. First, we define a landing zone type, `lz_t`, comprised of a global pointer and a size for a stored value:

```
struct lz_t {
  upcxx::global_ptr<char> gptr;
  size_t len;
};
```

Then our `local_map` maps to landing zones, rather than directly to values:

```
std::unordered_map<string, lz_t> local_map;
```

We define a function, `make_lz`, which takes as input a key and value size, and creates a landing zone using `upcxx::allocate` to allocate uninitialized space in global memory. `make_lz` inserts the key and landing zone pointer into the local map, returning a global pointer suitable for use in RMA:

```
upcxx::global_ptr<char>
make_lz(string key, size_t len) {
  upcxx::global_ptr<char> dest =
                    upcxx::allocate<char>(len);
  local_map.insert({key, {dest, len}});
  return dest;
}
```

[7] To facilitate discussion, the example code presented in this section is a simplified version of our actual distributed hash table benchmark, which uses an integer key type and a value type of `std::array<uint64_t, N>` rather than `std::string`.

[8] For brevity, we omit the find operation; it can be similarly implemented using RPC.

Finally we show the RMA-enabled hash table insert operation:

```
upcxx::future<> insert(const string &key,
                       const string &val) {
  upcxx::future<upcxx::global_ptr<char>> f;
  f = upcxx::rpc(get_target(key),
                 make_lz, key, val.size()+1);
  return f.then(
     [val](upcxx::global_ptr<char> dest) {
        return upcxx::rput(val.c_str(), dest,
                           val.size()+1);
     });
}
```

This function initiates an RPC of `make_lz` to obtain the remote landing zone pointer, and schedules a callback to transfer the value data to the retrieved location using `upcxx::rput`. Note the RPC returns a future containing the global pointer, so the `rput` is injected by a `.then` callback that executes when the RPC's future completes. The `.then` operation itself returns a future representing completion of the chain, and is returned by `insert` to represent the entire asynchronous insertion operation.

We measured the distributed hash table implementation with a simple use case where each process inserts a different set of randomly generated 8-byte keys, with varying sizes of values. For each element size, the same total volume of data was inserted, e.g. a run with an element size of 2KB would execute 4x more iterations than a run with an element size of 8KB. The benchmark blocks after each insertion, so this application is limited by communication latency[9].

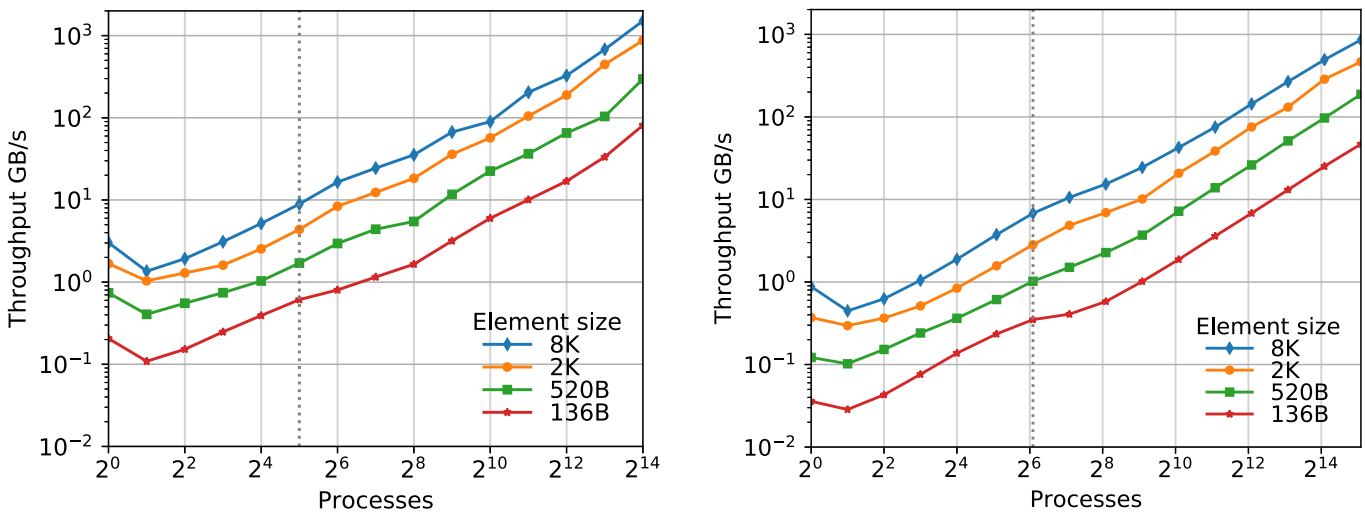
Weak scaling results are shown in Fig. 4, running from a single process up to 512 nodes (16384 processes on Cori Haswell and 34816 processes on Cori KNL). As expected, there is an initial decline in performance from one to two processes, as we move from serial to parallel operation. The serial code omits all calls to `UPC++`, and thus represents the best we can achieve with the underlying C++ standard library. Beyond two processes, the implementation scales efficiently. Although the communications are fine-grained, they are random and so the network traffic is well-distributed, which aids in the scaling.

As we have shown with this application motif, RPCs can be a useful mechanism for implementing distributed hash tables. They are particularly elegant when we need to update complex entries in the hash table. Suppose we represented a graph as a distributed hash table, where each vertex is a class containing a vector of neighbors, among other properties:

```
struct Vertex {
  ... // various properties
  vector<string> nbs;
};
```

Now, if we wish to update a vertex (with id `v`) to add a new neighbor (with id `nb`), that is easy to do with RPCs (this simplified example assumes the key is always found):

[9] Latency performance is a key consideration for many distributed hash table applications, such as genome assembly [13].



(a) Cori Haswell

(b) Cori KNL

Fig. 4: Weak scaling of distributed hash table insertion. The dotted line represents the processes in one node.

```
upcxx::rpc(get_target(v),
           [](string key, string val) {
             auto it = local_map.find(key);
             auto vertex = &it->second;
             vertex->nbs.push_back(val);
           }, v, nb);
```

By contrast, if we had no support for RPCs and had to use only one-sided puts and gets, this would be more complicated, error-prone, and likely less efficient. We'd have to first lock the entry in the hash table, then `rget` the Vertex from the remote process, modify it locally, `rput` it back to the remote process, and then unlock it. The representation of the Vertex class would also have to be modified, because the representation of STL types `std::vector` and `std::string` are not amenable to direct access via RMA.

D. Sparse Solvers

1) Background: Our second application motif is a sparse solver. Sparse matrices contain a significant number of zeros, and by taking advantage of this specific structure, sparse matrix computations employed by sparse solvers are able to drastically reduce storage and computing costs. Here we focus on an operation called *extend-add* (henceforth denoted `e_add`), a key component of *multifrontal sparse solvers* [14].

In multifrontal algorithms, computations comprise a sequence of parallel operations on dense matrices called *frontal matrices* that correspond to a part of the entire sparse matrix. Frontal matrices are organized along the *elimination tree* [15] of the sparse matrix, forming a hierarchy described in terms of a parent/child relationship representing the numerical dependencies among rows/columns during the factorization. Fig. 5 depicts two levels of such a hierarchy; the three frontal matrices represent a portion of some sparse matrix A.

The `e_add` operation entails a child updating the parent where each use different local coordinate systems, therefore indices of the child must be mapped to the parent's corresponding indices. The mapping between *local* and *global* element indices is stored in the metadata of each frontal matrix, namely I_p for the parent and I_{lC} / I_{rC} for each child. Frontal matrices are processed in parallel: they are mapped onto groups of processes using the *proportional mapping* heuristic [16], which assigns subtrees of frontal matrices to groups of processes of varying size depending on their computational cost. Frontal matrices are then distributed in a 2D block-cyclic manner with a fixed block size among processes of each group, as depicted by the colored blocks in Fig. 5.

The numerical factorization of a sparse matrix corresponds to a single bottom-up traversal of the elimination tree, more precisely the frontal matrix tree in the context of the multifrontal algorithm. At each level of the hierarchy, it is convenient to partition a frontal matrix F into four component matrices:

$$F = \left(\begin{array}{c|c} F_{11} & F_{12} \\ \hline F_{21} & F_{22} \end{array}\right)$$

F_{11}, F_{12}, and F_{21} are called *factors* and correspond to the output of the algorithm. F_{22} is called the *contribution block*, and contains the values that the child will use to update the parent frontal matrix. This matrix is a temporary, and can be discarded once the `e_add` has completed.

At each level of the traversal, the `e_add` operation applies the updates from the F_{22} matrix of each child to the four submatrices of the parent. Element indices of each child are first converted to global indices via I_{lC} or I_{rC}, and then onto local indices in the parent's coordinate system via I_p. Fig. 5 depicts these updates on a 2-by-3 process grid. The red arrows show where entries represented by blue dots in the left child are sent to the parent and accumulated. In this example, the parent frontal matrix is distributed over six processes, the left child is distributed over two processes (magenta and yellow),

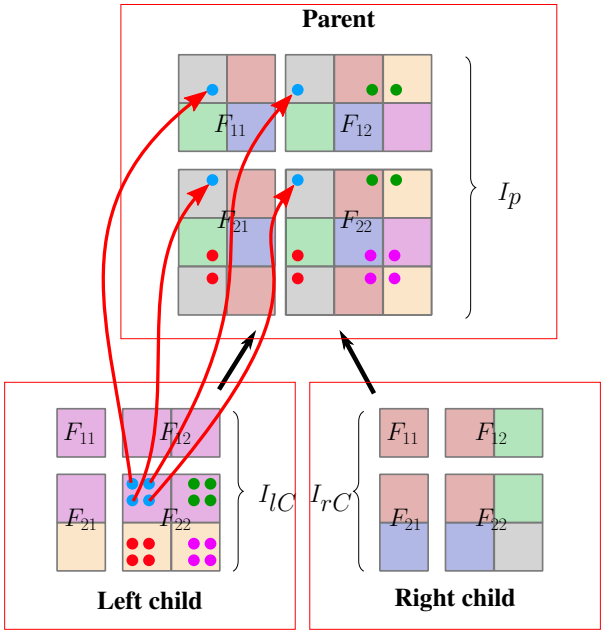


Fig. 5: The `e_add` operation on a 2-by-3 process grid.

and the right child is distributed over the remaining four. The number of processes will generally not be the same in the parent and the children.

2) Extend-add Implementation: In many multifrontal solvers, the update operation is implemented in three steps: (1) processes working on a child compute the locations, in the coordinate system of the parent, where values of their respective chunk of the contribution block need to be accumulated in the parent; (2) values are communicated between all processes assigned to the parent; and (3) the received values are accumulated by the owner of local chunks of the parent frontal matrix. The communication step (2) can be performed either using an all-to-all collective communication or a non-blocking point-to-point strategy. State-of-the-art solvers like STRUMPACK [17] implement this step using the former approach while solvers such as MUMPS [18] use the latter.

Our `UPC++` implementation is similar to the point-to-point strategy. It issues an RPC to every process in the parent, and the input data to these RPCs (i.e. numerical values to accumulate on a given process) are serialized by the `UPC++` framework and sent over the network using `UPC++` *views* [2]. A view is a mechanism that enables an RPC to serialize a sequence accessed via a user-provided iterator. After the RPC arrives at the target process, it is executed to accumulate data into the parent frontal matrix. Fig. 6 depicts the operation. The magenta process in the left child (1) packs the data going to each remote process, (2) issues three RPCs to the red, yellow, and gray processes to transfer the data, and (3) RPCs are executed on the target processes to accumulate received data into the locations indicated by the red arrows. This corresponds to finding the locations of indices i_1, i_2, i_3, and i_4 from I_{lC} in the parent index set I_p.

We now demonstrate in detail how these three steps can be carried out using `UPC++`. The implementation defines a class `FMat` that includes several fields important to this discussion:

- `lChild` and `rChild`: pointers to left and right children
- `row_indices`: a vector containing the global indices of the frontal matrix in the sparse matrix (corresponding to I_p, I_{lC} and I_{rC} in Fig 5)
- `front_team`: a `upcxx::team` object (similar in functionality to an MPI communicator) representing the processes onto which this frontal matrix is mapped
- `pack`: a utility function that compares `row_indices` of the child and the parent, determines which numerical values are to be sent to a given process in the parent, and bins them into appropriate buffers
- `e_add_prom`: a `upcxx::promise` initialized with the number of incoming RPCs expected by the current process; this promise acts as a counter and has an associated `upcxx::future` which becomes ready when the counter reaches zero

The top-level code is shown in Fig 7. The `e_add` function iterates over both children of the process and calls the `eadd_send` helper function (at lines 7-10), which packs the data to be sent and calls `upcxx::make_view` to create a serializable `upcxx::view` object, `v`, of the data destined for each process of the parent frontal matrix (line 26). The actual serialization is done by the RPC injection call, which eventually results in remote invocation of the `accum` function. The `accum` callback (not shown) traverses the data packed in the `upcxx::view` argument (a non-owning view into the incoming network buffer), accumulates each element into one of the local factor matrices ($F_{11}, F_{21}, F_{12}, F_{22}$), and calls `e_add_prom.fulfill_anonymous(1)`, signaling the `e_add()` function that the expected incoming RPC has been processed. The RPC injection returns a future, `fut`, that is used to track acknowledgment (at line 28). The returned future objects are conjoined into a single future, `f_conj`, via the `upcxx::when_all` function (line 29).

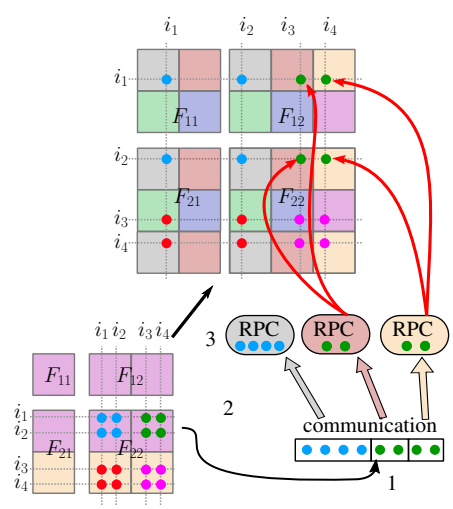


Fig. 6: `e_add` operation implemented with `UPC++` using RPCs on a 2-by-3 process grid.

```
1 void FMat::e_add() {
2   vector<vector<double> > sbufs[2];
3   // empty future, the starting point
4   // of conjoined futures
5   upcxx::future<> f_conj;
6   f_conj = upcxx::make_future();
7   eadd_send(this->lChild, sbufs[0],
8             f_conj, this->front_team);
9   eadd_send(this->rChild, sbufs[1],
10            f_conj, this->front_team);
11  // wait until all RPCs issued by this
12  // process have completed
13  upcxx::when_all(f_conj,
14          e_add_prom.finalize()).wait();
15 }
```

```
16 void FMat::eadd_send(FMat* ch, vector<vector<double> > &sbuf,
17              upcxx::future<> &f_conj, upcxx::team &front_team) {
18   int myrank = front_team.rank_me(), P = front_team.rank_n();
19   int myrank_ch = ch->front_team.rank_me();
20   pack(ch, sbuf); // bin outgoing entries into sbuf
21   // launch an RPC to every process in front_team
22   for (int lp = 0; lp < P; lp++) {
23     pdest = (myrank + 1 + lp) % P;
24     if (sbuf[pdest].size() == 0) continue;
25     // construct a serializable view of the data
26     auto v = upcxx::make_view(sbuf[pdest]);
27     // accumulate into factor matrix on the remote processor
28     auto fut = upcxx::rpc(front_team[pdest], accum, myrank_ch, v);
29     f_conj = upcxx::when_all(f_conj, fut); // conjoin
30   }
31 }
```

Fig. 7: Extend-add code sketch. `this` refers to the parent frontal matrix object.

This function is also used to conjoin `f_conj` to the future associated with the `e_add_prom` promise, obtained using `promise::finalize` (lines 13-14). This ensures that all the expected RPCs have been executed by the current process.

3) Extend-add Evaluation: We evaluated the performance of our `UPC++` implementation against two MPI variants, one using all-to-all collectives and the other using `MPI_Isend/Irecv` calls. Our use of these two different MPI variants follows the strategy generally employed by state-of-the-art solvers such as STRUMPACK and MUMPS. We use the `audikw_1` sparse matrix input from the Suite Sparse matrix collection [19]. The frontal matrix tree and data distribution information are extracted from the STRUMPACK solver. No computation other than the accumulation of numerical values is performed by the benchmark. Each data point corresponds to the mean of 10 runs, and each variant executes the exact same amount of computation and communicates the same amount of data.

The results are shown in Fig. 8 on both Cori Haswell and KNL, using 32 and 64 processes per node, respectively; the all-to-all variant is labeled `MPI Alltoallv`, whereas the point-to-point variant is labeled `MPI P2P`. As can be observed in the figure, the `UPC++` implementation maintains a consistent advantage over both MPI implementations, delivering up to a 1.63x speedup relative to `MPI Alltoallv` and a 3.11x speedup relative to `MPI P2P`. Similar results were observed for other matrices from the Suite Sparse matrix collection. This demonstrates that the RPC-based approach can be effective for implementing non-blocking asynchronous computations such as those required by the extend-add operation. Furthermore, `UPC++` RPCs offer an elegant way to implement this operation from a productivity standpoint.

4) symPACK Comparison Between UPC++ Versions: In the next experiment, we analyze the performance of a direct linear solver for sparse symmetric matrices named symPACK, which has been shown to be competitive against state-of-the-art solvers [20]. It was originally implemented using the predecessor `UPC++` and has recently been ported to `UPC++` v1.0. The previous implementation used v0.1 *asyncs* and *events* to schedule the asynchronous communication. These translated naturally to RPCs and futures, respectively, in v1.0. We compared the performance of both implementations of symPACK on NERSC Cori Haswell with 32 processes/node, using the `Flan_1565` sparse matrix from the Suite Sparse matrix

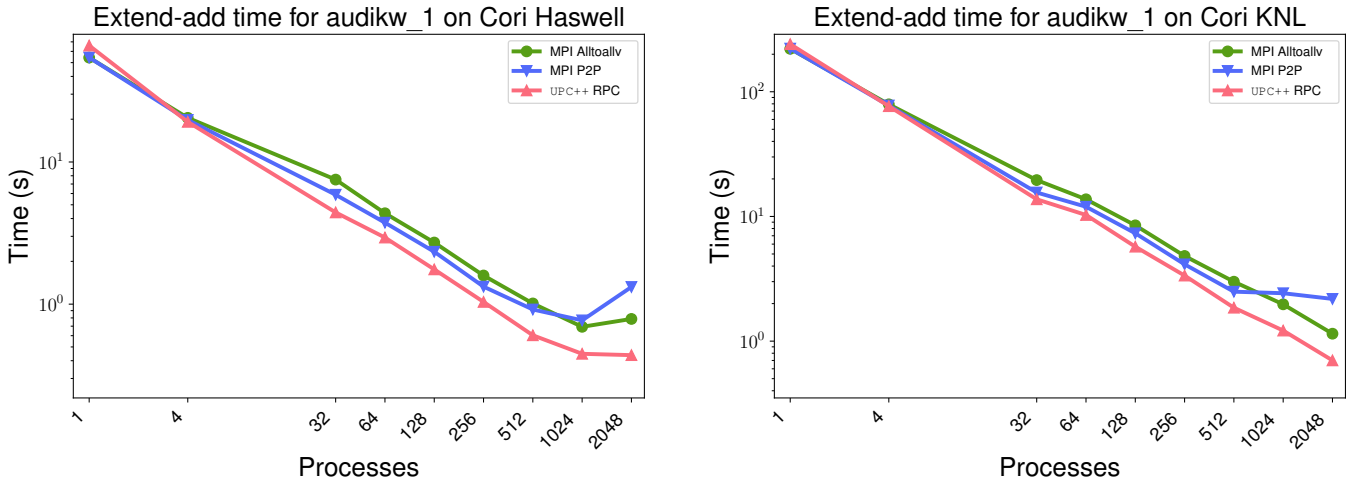


Fig. 8: Strong scaling of extend-add on Cori Haswell (left) and Cori KNL (right).

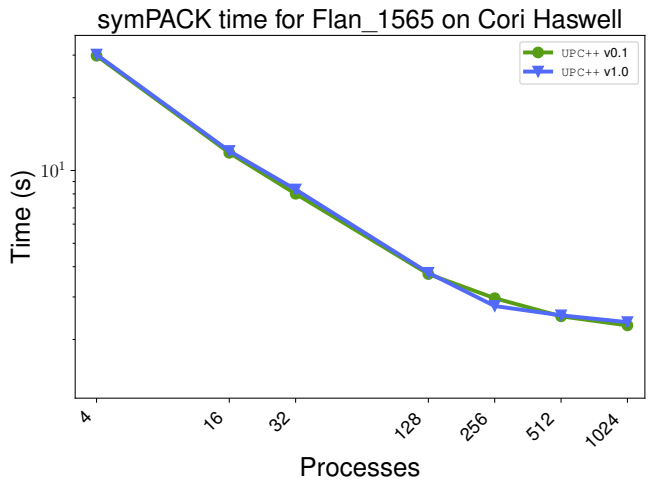


Fig. 9: Strong scaling comparison of symPACK using UPC++ v0.1 and v1.0 on Cori Haswell.

collection and reporting the mean time for 10 runs at each data point. Results depicted in Fig 9 show the performance of the two implementations to be nearly identical; the average difference in performance across all job sizes is 0.7%, with the UPC++ v1.0 variant providing up to a 7.2% advantage at 256 processes. This demonstrates that the new UPC++ framework does not incur any measurable added overheads for this application.

V. Related Work

A. Comparison to Predecessor UPC++ v0.1

The version of UPC++ presented in this paper differs considerably from the predecessor developed by Zheng et al [4]. Both are libraries supporting the PGAS model and use GASNet [21] as the underlying communication layer, but the APIs are quite different. In the current version, the principles of making communication explicit and avoiding non-scalable data structures means that we've dropped support for implicit dereference of global pointers, shared scalars and shared arrays. Furthermore, the API for expressing asynchrony in our current version is based on the abstraction of futures and promises, as introduced in C++11 [7], rather than the async/finish interface in the predecessor version and other models such as X10 [22] and Habanero-C [23]. The new model improves composability of asynchronous operations, and it enables additional flexibility in the expression of data movement and synchronization.

Our new version of UPC++ provides substantial new capabilities that are absent from its predecessor. The future abstraction encapsulates both data values as well as readiness information, as opposed to events in the old version that carry readiness information only. This semantic binding enables asynchronous operations that return values; as such, the new version's RPCs are permitted to return a value, while the old version's asyncs could not. The future abstraction also frees the programmer from the burden of explicitly managing event-object lifetime, which can be challenging in algorithms with highly asynchronous and irregular communication patterns. RMA operations in the predecessor UPC++ were also very limited – they did not support events, and there was no mechanism to attach a local or remote operation to the completion of an RMA. The ability to attach an operation which effectively serves as a completion handler is semantically elegant, and it leads to more compact code. As a result of the limitations of the predecessor, a hash-table insertion operation similar to the one presented in section IV-C requires 50% more lines of code in old UPC++, and it incurs both a blocking remote allocation and a blocking RMA, which negatively impact latency performance and overlap potential. Our improvements to asynchrony support directly enable the simpler, streamlined, and fully asynchronous implementation of distributed hash table that scales beyond thousands of cores.

In addition to the incomplete support for asynchrony, the old version of UPC++ lacked several important design features introduced by the new version, such as atomics and view-based serialization of RPC arguments. Finally, the new version of UPC++ has a more formal and rigorous specification, compared to the incomplete documentation and specification of its predecessor.

B. Other Programming Models

Several recent and older programming systems support the PGAS model, including UPC [5, 24], Fortran 2008 coarrays [25], OpenSHMEM [9], and Titanium [26, 27]. While X10 and Chapel [28] both support remote task execution, their execution model is rooted in forking and joining tasks, placing less emphasis on PGAS-style RMA operations. UPC++ supports the SPMD execution model provided by traditional PGAS systems, but augments it with remote procedure calls.

The implementation of UPC++ notably takes a template-metaprogramming approach rather than relying upon a custom compiler, resulting in a lightweight and sustainable implementation that leverages existing C++ compilers and simplifies interoperability with other C++ libraries. There are a number of programming systems that take a compiler-free, C++-library approach toward parallel programming on distributed machines. DASH [29] is a PGAS library implemented over DART [30], which has an MPI-3 RMA backend. Like UPC++, DASH provides global pointers, but unlike UPC++, it lacks support for RPCs or any form of code shipping and it includes implicit communication via dereference. STAPL [31] is another parallel programming library, based on an Adaptive Remote Method Invocation (ARMI) layer. It does not expose a true PGAS API, but instead abstracts the details of the data distribution and parallelism into elementary patterns (e.g. map, map-reduce, scan, zip). Another PGAS library is Coarray C++ [32], which focuses on distributed data structures such as coarrays. It assumes the existence of a symmetric shared heap, an implementation detail that UPC++ has deliberately avoided because it can result in non-scalable data structures and is incompatible with subset teams. Like UPC++, Coarray C++ provides asynchrony, but in the form of cofutures. There exist

some other libraries (e.g Hierarchically Tiled Arrays [33]) that focus explicitly on distributed data structures.

Other libraries, such as HPX [34], Phalanx [35], and Charm++ [36] provide a more task-based execution model rather than the core SPMD abstraction of UPC++. HPX, Charm++, and HabaneroUPC++ [37] all support task scheduling and load balancing. By contrast, UPC++ is intended to provide lightweight constructs for communication and remote execution with basic progress guarantees, while supporting interoperation with external scheduling libraries.

VI. Discussion and Conclusions

We have presented UPC++, a C++ PGAS library. UPC++ provides low-overhead, one-sided RMA communication, remote procedure call and remote atomics. UPC++'s asynchronous communication model is based on futures and promises. Futures capture data readiness state, and they enable the programmer to chain or conjoin operations to execute asynchronously as high-latency dependencies become satisfied, via completion handlers. Promises provide another mechanism for tracking completion of multiple operations.

Current work includes adding a rich set of non-blocking collective operations. Future work will enhance UPC++'s one-sided communication to express transfers to and from other memories (such as that of GPUs) with extensions to the existing abstractions.

Unlike many other PGAS models, UPC++ forbids syntactically implicit communication, and all communication is non-blocking by default. These restrictions were made as conscious design decisions in order to encourage the programmer to write code that is performant by carefully considering communication costs. The communication model closely matches the unordered delivery and RMA semantics of modern RDMA network hardware, unlike two-sided message passing. The increased semantic flexibility improves the possibility of overlapping communication and scheduling it appropriately.

UPC++'s ability to offer low-overhead communication relies on the GASNet-EX communication library, which hides incidental details of the communication network while engaging any available low-level hardware support. Future work includes benchmarking UPC++ applications on additional supercomputing networks supported by GASNet-EX.

We demonstrated the benefits of UPC++ on a Cray XC40 via microbenchmarks and two application motifs that perform fine-grained communication. UPC++ delivers up to a 25% improvement in the latency of blocking RMA put, and up to a 33% bandwidth improvement in an RMA throughput test. Our distributed hash table in UPC++ delivers near-linear weak scaling up to 34816 cores. Our UPC++ implementation of the sparse solver component shows robust strong scaling up to 2048 cores, where it outperforms variants communicating using MPI by up to 3.1x. In these proxy applications, RPC was vital in realizing high performance—RMA alone was not sufficient.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] UPC++ home page, http://upcxx.lbl.gov/

[2] J. Bachan, S. B. Baden, D. Bonachea, P. H. Hargrove, S. Hofmeyr, et al.: UPC++ Specification, v1.0 Draft 8. Tech. Rep. LBNL-2001179, Lawrence Berkeley Natl. Lab (September 2018). doi:10.25344/S45P4X

[3] J. Bachan, S. B. Baden, D. Bonachea, P. H. Hargrove, S. Hofmeyr, et al.: UPC++ Programmer's Guide, v1.0-2018.9.0. Tech. Rep. LBNL-2001180, Lawrence Berkeley Natl. Lab (September 2018). doi:10.25344/S49G6V

[4] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick: UPC++: A PGAS extension for C++. In: IEEE 28th International Parallel and Distributed Processing Symposium. pp. 1105–1114 (May 2014). doi:10.1109/IPDPS.2014.115

[5] UPC Consortium: UPC Language and Library Specifications, v1.3. Tech. Rep. LBNL-6623E, Lawrence Berkeley Natl. Lab (November 2013). doi:10.2172/1134233

[6] D. Bonachea and P. H. Hargrove: GASNet-EX: A High-Performance, Portable Communication Library for Exascale. Tech. Rep. LBNL-2001174, Lawrence Berkeley Natl. Lab (October 2018). doi:10.25344/S4QP4W, to appear: Languages and Compilers for Parallel Computing (LCPC'18)

[7] ISO: ISO/IEC 14882:2011(E) Information technology - Programming Languages - C++. Geneva, Switzerland (2012), https://www.iso.org/standard/50372.html

[8] P. H. Hargrove and D. Bonachea: GASNet-EX performance improvements due to specialization for the Cray Aries network. In: 2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM). pp. 23–33 (November 2018). doi:10.1109/PAW-ATM.2018.00008

[9] S. Pophale, R. Nanjegowda, T. Curtis, B. Chapman, H. Jin, et al.: OpenSHMEM Performance and Potential: A NPB Experimental Study. In: Proceedings of the 6th Conference on Partitioned Global Address Space Programming Models (PGAS'12). (2012), https://www.osti.gov/biblio/1055092

[10] Cray, Inc.: Cray XC Series. https://www.cray.com/products/computing/xc-series, accessed 2018-07-17

[11] NERSC: National Energy Research Scientific Computing Center. http://www.nersc.gov

[12] Intel Corp.: Introducing Intel® MPI Benchmarks. https://software.intel.com/en-us/articles/intel-mpi-benchmarks

[13] E. Georganas, A. Buluç, J. Chapman, S. Hofmeyr, C. Aluru, et al.: HipMer: An Extreme-scale De Novo Genome Assembler. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 14:1–14:11. SC '15 (2015). doi:10.1145/2807591.2807664

[14] I. S. Duff and J. K. Reid: The multifrontal solution of indefinite sparse symmetric linear. ACM Transactions on Mathematical Software (TOMS) **9**(3), 302–325 (1983). doi:10.1145/356044.356047

[15] J. W.-H. Liu: The role of elimination trees in sparse factorization. SIAM Journal on Matrix Analysis and Applications **11**, 134–172 (1990). doi:0.1137/0611010

[16] A. Pothen and C. Sun: A mapping algorithm for parallel sparse Cholesky factorization. SIAM Journal on Scientific Computing **14**(5) (1993). doi:10.1137/0914074

[17] P. Ghysels, X. Li, F. Rouet, S. Williams, and A. Napov: An Efficient Multicore Implementation of a Novel HSS-Structured Multifrontal Solver Using Randomized Sampling. SIAM Journal on Scientific Computing **38**(5), S358–S384 (2016). doi:10.1137/15M1010117

[18] P. Amestoy, I. Duff, J.-Y. L'Excellent, and J. Koster: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM Journal on Matrix Analysis and Applications **23**, 15–41 (2001). doi:10.1137/S0895479899358194

[19] T. A. Davis and Y. Hu: The University of Florida sparse matrix collection. ACM Transactions on Mathematical Software **38**, 1 (2011). doi:10.1145/2049662.2049663

[20] J. Bachan, D. Bonachea, P. H. Hargrove, S. Hofmeyr, M. Jacquelin, et al.: The UPC++ PGAS library for exascale computing. In: Proceedings of the Second Annual PGAS Applications Workshop. pp. 7:1–7:4. PAW17 (2017). doi:10.1145/3144779.3169108

[21] D. Bonachea and P. H. Hargrove: GASNet specification, v1.8.1. Tech. Rep. LBNL-2001064, Lawrence Berkeley Natl. Lab (August 2017). doi:10.2172/1398512

[22] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, et al.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. (OOPSLA'05) (2005). doi:10.1145/1103845.1094852

[23] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cave, M. Chabbi, et al.: Integrating asynchronous task parallelism with MPI. International Parallel and Distributed Processing Symposium pp. 712–725 (2013). doi:10.1109/IPDPS.2013.78

[24] UPC consortium home page, http://upc-lang.org/

[25] J. Reid: Coarrays in the Next Fortran Standard. SIGPLAN Fortran Forum **29**(2), 10–27 (July 2010). doi:10.1145/1837137.1837138

[26] P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, et al.: Titanium language reference manual. Tech Report UCB/EECS-2005-15.1, University of California, Berkeley (November 2001). doi:10.25344/S4H59R

[27] K. Yelick, P. Hilfinger, S. Graham, D. Bonachea, J. Su, et al.: Parallel Languages and Compilers: Perspective from the Titanium Experience. International Journal of High Performance Computing Applications **21**(3), 266–290 (2007). doi:10.1177/1094342007078449

[28] D. Callahan, B. L. Chamberlain, and H. P. Zima: The Cascade High Productivity Language. International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS) pp. 52–60 (2004). doi:10.1109/HIPS.2004.10002

[29] K. Fürlinger, C. Glass, A. Knüpfer, J. Tao, D. Hünich, et al.: DASH: Data Structures and Algorithms with Support for Hierarchical Locality. In: Euro-Par Parallel Processing Workshops (2014). doi:10.1007/978-3-319-14313-2_46

[30] H. Zhou, Y. Mhedheb, K. Idrees, C. W. Glass, J. Gracia, et al.: DART-MPI: An MPI-based Implementation of a PGAS Runtime System. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models. pp. 3:1–3:11. PGAS '14 (2014). doi:10.1145/2676870.2676875

[31] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, et al.: STAPL: An Adaptive, Generic Parallel C++ Library. In: Languages and Compilers for Parallel Computing (LCPC 2001). Lecture Notes in Computer Science, vol. 2624. Springer (2001). doi:10.1007/3-540-35767-X_13

[32] T. A. Johnson: Coarray C++. In: Proceedings of the 7th International Conference on PGAS Programming Models. pp. 54–66. PGAS'13 (2013), https://www.research.ed.ac.uk/portal/files/19680805/pgas2013proceedings.pdf

[33] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almási, B. B. Fraguela, et al.: Programming for parallelism and locality with hierarchically tiled arrays. In: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). pp. 48–57 (2006). doi:10.1145/1122971.1122981

[34] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. 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. pp. 6:1–6:11. PGAS '14 (2014). doi:10.1145/2676870.2676883

[35] M. Garland, M. Kudlur, and Y. Zheng: Designing a unified programming model for heterogeneous machines. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC '12 (2012). doi:10.1109/SC.2012.48

[36] L. Kalé and S. Krishnan: CHARM++: A portable concurrent object oriented system based on C++. In: A. Paepcke (ed.) Proceedings of OOPSLA. pp. 91–108 (September 1993). doi:10.1145/167962.165874

[37] V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar: HabaneroUPC++: A Compiler-free PGAS Library. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS) (2014). doi:10.1145/2676870.2676879