Optimization of Asynchronous Communication Operations through Eager Notifications

Amir Kamil[1,2] Dan Bonachea[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 implementing the Asynchronous Partitioned Global Address Space (APGAS) model. We propose an enhancement to the completion mechanisms of UPC++ used to synchronize communication operations that is designed to reduce overhead for on-node operations. Our enhancement permits eager delivery of completion notification in cases where the data transfer semantics of an operation happen to complete synchronously, for example due to the use of shared-memory bypass. This semantic relaxation allows removing significant overhead from the critical path of the implementation in such cases. We evaluate our results on three different representative systems using a combination of microbenchmarks and five variations of the the HPCChallenge RandomAccess benchmark implemented in UPC++ and run on a single node to accentuate the impact of locality. We find that in RMA versions of the benchmark written in a straightforward manner (without manually optimizing for locality), the new eager notification mode can provide up to a 25% speedup when synchronizing with promises and up to a 13.5x speedup when synchronizing with conjoined futures. We also evaluate our results using a graph matching application written with UPC++ RMA communication, where we measure overall speedups of as much as 11% in single-node runs of the unmodified application code, due to our transparent enhancements.**

***Index Terms*—UPC++, GASNet-EX, PGAS, RMA, Atomics.**

I. Introduction

The Partitioned Global Address Space (PGAS) model provides both excellent performance and productivity. The one-sided data-movement model is a good semantic match to the capabilities provided by modern network hardware, and a single programming interface can be used for both distributed and shared memory. However, the same operation may require orders of magnitude more time to complete over distributed memory compared to shared memory, so asynchrony is often used to enable overlap of communication with computation or other communication, hiding the higher latency of network operations.

Asynchrony requires defining a progress model, specifying when an asynchronous operation may complete and the requirements on a program for ensuring that such operations make forward progress. For an operation whose cost depends on whether or not the initiating process has direct access to a target memory location (e.g. a one-sided put via a global pointer that may reference either on-node or off-node memory), a progress model may specify that completion must always be signaled asynchronously to the program. This provides uniform semantics over both shared and distributed memory. On the other hand, it may impose significant additional costs on operations that target on-node memory. For applications where most asynchronous communication operations are resolved on-node, or that happen to be run on a single node, these costs may have a nontrivial adverse effect on performance.

In this work, we examine the cost of delayed notifications in UPC++, a C++11 library that provides an Asynchronous Partitioned Global Address Space (APGAS) model. The 2021.3.0 and earlier releases of the library require deferred notification of all asynchronous operations, and we previously proposed an extension for requesting eager notifications where possible [10]. We implemented these extensions and associated optimizations for Remote-Memory Access (RMA) and atomic operations. We compare shared-memory performance of several benchmarks with this extension against the prior UPC++ implementation, demonstrating that early notifications can indeed provide better performance than universally delaying them.

II. Background

UPC++ [4, 5, 9] is a C++ library that implements the Asynchronous Partitioned Global Address Space (APGAS) programming model, providing communication operations that include Remote Procedure Call (RPC) and RMA. In this model, the global address space is partitioned among the *processes*, each of which has a private memory and a shared memory segment; the global address space is the union of all the shared segments. In UPC++, a global address is represented by a *global pointer*, implemented as a class template over the underlying object type. For example, the following[1] allocates an `int` object in the shared segment of the calling process:

```
global_ptr<int> gptr = new_<int>(3);
```

The return value of the allocation above is a global pointer, but it is actually referring to an object in on-node memory. This pointer can be *downcast* to obtain a raw C++ pointer:

```
int *lptr = gptr.local();
```

This downcast is valid on all same-node processes, since the implementation ensures they have direct access to the underlying memory. The `global_ptr` template has an `is_local`

[1]For brevity, we elide `upcxx::` namespace qualifiers in code examples.

 doi:10.25344/S42C71

method that returns whether or not the caller has direct access to the memory.

A global pointer can be sent to another process, which can use it to initiate asynchronous remote-memory operations:

```
global_ptr<int> gptr = /* obtain pointer */;
future<int> fut = rget(gptr);
future<> done = fut.then([=](int val) {
  return rput(val+1, gptr);
});
done.wait();
```

By default, an asynchronous operation returns a *future*, which encapsulates both the readiness state of the operation (whether it has completed) and any values produced by the operation. In the code above, the `rget` operation initiates a remote read of the argument, returning a future representing the read value. The code attaches a callback to that future to be executed when it is ready, and the callback initiates a remote write on the same global pointer. The `rput` call produces its own value-less future, which is also passed on as the return value of the callback. The code then does a wait on the resulting future, which is only readied after the `rget` completes, the callback runs, and the subsequent `rput` completes.

A. UPC++ Completions

While an asynchronous operation produces a future by default that represents overall completion of the operation, UPC++ actually provides a powerful *completions* mechanism that enables a program to request other forms of notifications as well as notification of different events. The latter include:

- *source completion*: for operations using a source buffer, indicates when that buffer is available for reuse or reclamation by the initiator
- *remote completion*: for RMA put, a callback can be scheduled to run on the target process after data arrival
- *operation completion*: when the overall operation has completed from the perspective of the initiator

Notification options include futures, promises, and local procedure calls for source and operation completion, and remote procedure calls for remote completion. A program may request any combination of notifications for the events that are relevant to a communication operation. The following is an example of a bulk data put, which supports all three events:

```
static bool done = false;
promise<> prom;
int *array = /* ... */;
global_ptr<int> gptr = /* ... */;
std::tuple<future<>, future<>> result =
  rput(array, gptr, size,
       source_cx::as_future() |
       remote_cx::as_rpc([](){ done=true; }) |
       operation_cx::as_future() |
       operation_cx::as_promise(prom));
```

The code above requests a custom set of completions by passing a completions object as the last argument to `rput`, constructing the object as a composition of individual event/action factory methods. Since two future notifications were requested here, the `rput` call returns a tuple of two futures representing the corresponding events. The first will be readied when the source buffer is safe to reuse, and the second when the operation as a whole is complete. In addition, the code above specifies an RPC callback to run on the target process after the data have been transferred, and it also requests notification on the promise `prom` when the operation completes.

In UPC++, a future is the consumer side of an asynchronous result, while a *promise* is the producer side. Promises are particularly efficient at keeping track of multiple asynchronous operations, essentially acting as a counter. Here is an example:

```
promise<> p;
global_ptr<int> gps[10] = /* ... */;
for (int i = 0; i < 10; ++i)
  rput(i, gps[i], operation_cx::as_promise(p));
p.finalize().wait();
```

The code creates one promise and registers ten `rput` operations on it. The `finalize` call closes registration on the promise and returns a future, and the code waits on that future. The future is readied after all ten operations have completed.

A similar outcome can be obtained by *conjoining* the individual futures from multiple `rput` operations into a single future:

```
future<> f = make_future();
for (int i = 0; i < 10; ++i)
  f = when_all(f, rput(i, gps[i]));
f.wait();
```

The `when_all` combinator takes any number of futures (or non-future values), combining them into a single future that represents the values and readiness of all the inputs. Figure 1 illustrates a portion of the dependency graph that results from the code above, where each vertex represents a future constructed at runtime. In the 2021.3.0 UPC++ release, each such future corresponds to an implicitly constructed promise cell that is dynamically allocated on the heap. In comparison, the code that uses an explicit promise incurs only a single heap allocation, corresponding to the explicitly constructed promise, and fulfilling a dependency only involves decrementing a counter rather than traversing a dependency graph[2]. As such, the promise-based code is significantly more efficient, and the results in Section IV bear this out.

B. Progress and Deferred Notification

Previous versions of UPC++ require that notifications be *deferred* until the next call into the UPC++ progress engine (e.g. a wait on a future) [9]. Consider the example code in Listing 1. In this code, even if `gptr` happens to refer to on-node memory and the data transfer is performed synchronously before `rput` returns, the 2021.3.0 UPC++ implementation is prohibited from returning a ready future from `rput`. Thus,

[2] Although the two code snippets shown here are equivalent, UPC++ is implemented as a library and only discovers the unfolding dependency graph at runtime. Automated transformation of user-specified completion/synchronization mechanisms would require supporting model-specific compiler analysis and is beyond the scope of this work.

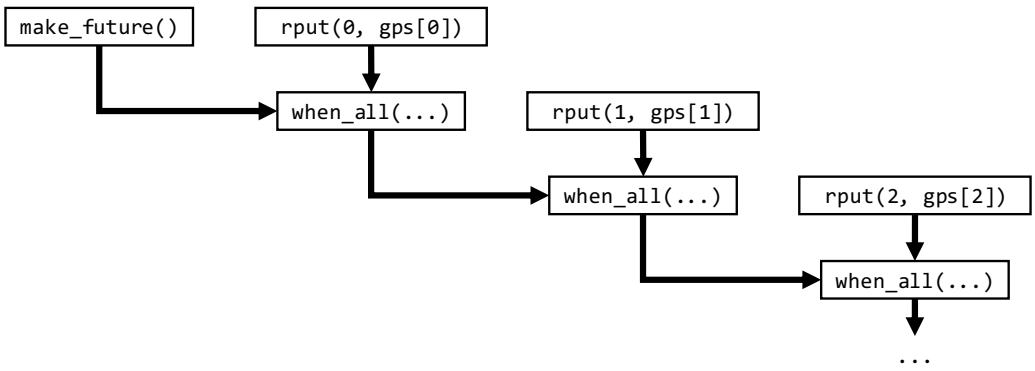

Fig. 1: Dependency graph resulting from conjoining futures.

```
global_ptr<int> gptr = producer();
// gptr may or may not be local
future<> f = rput(42, gptr);
future<> f2 =
  f.then([](){ do_something_dependent(); });
do_something_overlappable();
f2.wait();
```

Listing 1: Example Code

the programmer is assured that the subsequent callback will not run synchronously during `future<>::then()`, but will run during the `wait` call further down, regardless of whether `gptr` points to something local or remote.

While this implementation restriction leads to consistent behavior across local and remote accesses, it can incur significant overheads in the case of local access. In particular, the implementation must perform a heap allocation of an internal promise object corresponding to a non-ready future and add it to an internal queue to be readied later by the progress engine. If most accesses are local, these overheads encourage the programmer to add manual locality checks to bypass the asynchronous operation and its associated overheads when possible (as described in the next section). This practice negates one of the potential advantages of the PGAS model, that of using the same code for both local and remote memory access. Furthermore, manual bypass is not even possible in the case of atomics, which must go through `UPC++` and the underlying GASNet-EX [8] communication layer to ensure coherency correctness on systems that may offload incoming atomic operations using the network hardware.

C. Manual Localization

For the use case of local RMA operations, overheads due to deferred notification can be manually "bypassed" via manual localization, where a global pointer is downcast to a raw C++ pointer and accessed using normal C++ pointer dereference operations. This entirely bypasses `UPC++` machinery for operations that are known to be most efficiently satisfiable via synchronous load/store instructions on the cache-coherent memory system (where they are also amenable to compiler and architectural reordering optimizations).

```
global_ptr<int> gptr = producer();
// gptr may or may not be local
if (gptr.is_local()) {
  *(gptr.local()) = 42;
  do_something_dependent();
  do_something_overlappable();
} else {
  future<> f = rput(42, gptr);
  future<> f2 = f.then([] {
    do_something_dependent();
  });
  do_something_overlappable();
  f2.wait()
}
```

Listing 2: Manual Localization Example

If a global pointer is not statically known to point to directly addressable memory, a program can dynamically check the locality of a pointer before downcasting it. An example is shown in Listing 2.

A significant drawback to manual localization is that it leads to code bloat, making the code more difficult to maintain. In particular, it suffers from the following:

1) **It leads to unnecessary code duplication.** Programmers end up writing two or more versions of the code, one for the case where a global-pointer input is local at runtime and another for when it is remote, decreasing productivity and maintainability. If there are N global-pointer inputs with independent locality properties, this can expand to 2^N versions of the code, which is not scalable.
2) **The manual dynamic locality check is redundant with one that is always performed inside RMA calls.** The programmer has manually incurred the cost of a branch (or N branches) to decide to use an RMA call versus downcast, but the RMA implementation already includes a (now redundant) locality branch to choose the correct protocol (downcast versus network communication). If the majority of the accesses are dynamically remote, the cost associated with these extra, redundant branches may add up to significant performance degradation.

III. Modifications to UPC++

We made several changes to the UPC++ implementation to introduce and optimize eager notifications. These included modifications to the completions infrastructure and associated code within communication operations, introduction of new value-less atomic operations, and optimization of future conjoining.

A. Modifications to Completions

To enable eager notification of completion, we introduced new factories for explicitly requesting deferred or eager notification of promises and futures:

```
operation_cx::as_defer_future()
operation_cx::as_eager_future()
operation_cx::as_defer_promise(promise<T...> &p)
operation_cx::as_eager_promise(promise<T...> &p)
```

and similarly for `source_cx`. The `as_defer` variants guarantee deferral of notifications, matching the legacy behavior of UPC++ releases through 2021.3.0. The `as_eager` variants allow for, but do not guarantee, eager notification when the underlying data-movement operation completes synchronously during initiation.

We also added a `UPCXX_DEFER_COMPLETION` macro that controls whether the existing `as_future` and `as_promise` factories request eager or deferred completion notification. These factories result in eager notification by default in our implementation, but the macro can be defined to restore the legacy behavior of deferred notification[3].

Modifications to the UPC++ source code were largely confined to the completions logic. UPC++ makes heavy use of template metaprogramming to represent and process completions. We implemented specializations to bypass the progress queue when eager completion notification is both requested and dynamically possible, arranging for a ready future to be returned for notification via a future and eliding modifications to the given promise for notification via a promise.

Because synchronous completion of data movement is a dynamic property, some modifications to the implementation of RMA and atomic operations were required. We obtained this information through a combination of locality queries and completion status of the underlying GASNet-EX operation.

B. New Overloads of Fetching Atomics

Communication operations may either produce a value directly as part of an event notification, or they may only have side effects such as writing to a memory location. Value-containing futures and promises are ideal for chaining callbacks and automatically managing the lifetime of the underlying values. However, they are not convenient for conjoining the result of multiple operations into a single future or registering them on a single promise. In the case of futures, conjoining futures that encapsulate values produces a future with a different type. For example:

[3] At the time of this writing, we are unaware of any real application code whose correctness is sensitive to the subtle semantic difference between deferred and eager completion, only contrived/pathological examples.

```
global_ptr<int> gps[10] = /* ... */;
future<> f = make_future();
future<int> f0 = when_all(f, rget(gps[0]));
future<int,int> f1 = when_all(f0, rget(gps[1]));
```

Since the future type changes each time a new value-containing future is conjoined with an existing future, the idiom of conjoining futures in a loop shown in Section II-A does not work. Promises pose a similar problem: a promise can track any number of value-less operations, but it can only track a single operation that produces a value. Thus, the example of registering multiple communication operations on a single promise as shown in Section II-A does not work when the operations produce values.

Furthermore, we optimized the construction of ready value-less futures (`future<>`) as part of this work. Since such a future does not encapsulate a value, a common pre-allocated promise cell with its state set as ready can be used when constructing a ready value-less future. Thus, construction of such a future now elides allocation of an internal promise cell. Unfortunately, such an optimization cannot apply to ready futures that do contain a value – the value must be stored somewhere, and UPC++ uses a dynamically allocated internal promise cell to do so.

To sidestep the issues above, we introduced new variants of fetching atomic operations that write the fetched value to memory rather than producing it as part of event notification. In combination with eager notification, they enable fetching atomic operations that complete synchronously to avoid the overheads associated with allocating an internal promise cell (when notification via a future is requested) or of modifying the given promise (for notification via a promise).

C. Optimization of Future Conjoining

Additionally, we modified the implementation of `when_all` to optimize future conjoining when input futures are ready. We observed that if all the values come from a single input future, and all other input futures are already ready, then the result of conjoining them is semantically equivalent to the former single future. The following is an example:

```
future<int, double> fut1 = /* ... */;
future<> fut2 = /* ... */, fut3 = /* ... */;
auto result = when_all(fut1, fut2, fut3);
```

Here, `fut2` and `fut3` are value-less, and if they are ready before the call to `when_all`, then they do not contribute to the result in any way. Thus, the call to `when_all` can just return a copy of `fut1`. This optimization also applies when all input futures are value-less – if only one is non-ready, then that one future is the only one that contributes to the readiness of the result, so `when_all` can just return a copy of it. Similarly, if all the input futures are value-less and ready, `when_all` can return any one of them. These `when_all` optimizations are primarily relevant to ready futures that result from eager completions, but the programmer can also construct ready futures, which are useful for some UPC++ idioms (such

as the `make_future` call that forms the base case of the conjoining example shown in Section II-A).

IV. Experimental Results

To evaluate the impact of these optimizations, we compare three versions of `UPC++`:

- **2021.3.0**: the most recent official release as of this writing
- **2021.3.6 defer**: a more recent snapshot, with several new optimizations but still using deferred notifications
- **2021.3.6 eager**: the same snapshot as **2021.3.6 defer**, but using eager notifications

Experiments were run on three representative architectures:

- **Intel**: dual-socket 20-core 2.40 GHz Intel Xeon Gold 6148 (Skylake) processors with 384GiB DDR4-2666 DRAM, located in the GPU partition of NERSC Cori [20] and using the Intel 19.1.3.304 compiler
- **IBM**: dual-socket 22-core 3.07GHz IBM POWER9 processors with 512GiB DDR4-2666 DRAM, located in OLCF Summit [22] and using the GCC 10.2.0 compiler
- **Marvell**: dual-socket 32-core 2.20 GHz Marvell/Cavium ThunderX2 CN9980 (ARMv8.1) processors with 256GiB DDR4-2666 DRAM, located in OLCF Wombat [21] and using the Clang 11.0.1 compiler

Experiments were run on a single node of each system to model the case where most updates are to local memory. All of these systems feature GPU accelerators and high-performance network hardware; however our current study is focused on the CPU overheads associated with CPU-mediated on-node interprocess communication, hence these other components were idle for our experiments. For microbenchmarks and the GUPS benchmark, we used the SMP conduit on Intel. On IBM and Marvell, we used the UDP conduit for its better integration with the native job launcher; process-shared memory ensures all communication takes place via shared memory (not UDP sockets), with each process having direct access to the shared segments of all the other processes. For the graph-matching application, we used the MPI conduit to trivially satisfy that application's hybrid reliance on MPI collectives for data initialization. As with UDP, all communication in the timed region takes place using `UPC++` RMA operations targeting on-node shared memory. Each experimental result was obtained by running twenty samples, taking the average of the top ten. The exception is GUPS on IBM with 16 processes; due to higher noise in this experiment, we ran 60 samples and took the average of the top ten.

A. Microbenchmarks

To understand the effects of our optimizations in isolation, we measured the performance of several communication operations, consisting of either RMA or atomic transfers of individual 64-bit pieces of data. We collected data using notifications via futures; performance of promises is dependent on how many operations are aggregated on a single promise, obscuring the cost of a single operation.

Each experiment timed ten million operations, initiating and then immediately waiting on an operation as in the following:

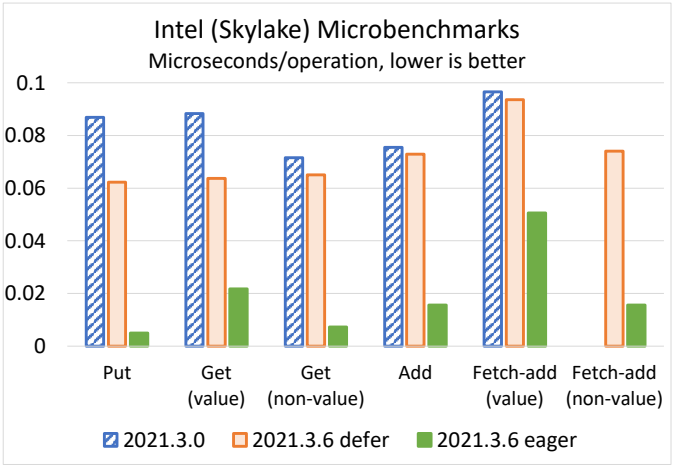

Fig. 2: Microbenchmark results on Intel.

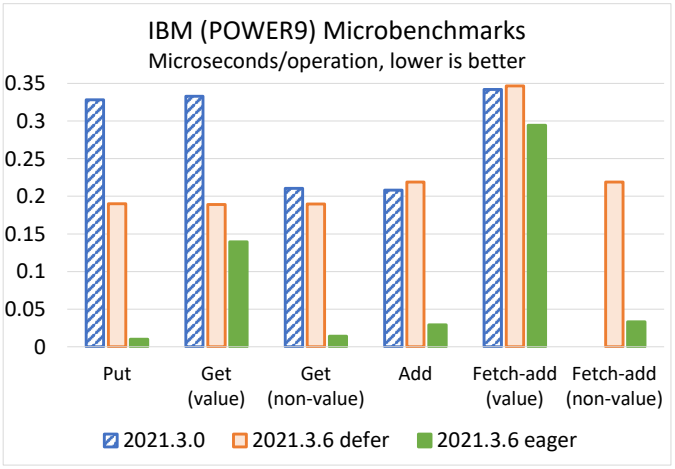

Fig. 3: Microbenchmark results on IBM.

```
global_ptr<double> gp = /* ... */;
for (int i = 0; i < 10000000; ++i)
  rput(0., gp, operation_cx::as_future())
    .wait());
```

The total time over this loop was divided by the number of operations to compute the average time per operation. This was further averaged over the top ten samples, as described above. Figures 2, 3, and 4 show the results on all three systems. Note that since our work introduced non-value fetching atomics, there is no measurement for that operation on the 2021.3.0 version as it did not exist.

The differences between the 2021.3.0 and 2021.3.6 defer results are due to an optimization we made orthogonal to whether notifications are deferred or eager, which was the elimination of an additional heap allocation for an RMA operation on directly addressable global pointers. Eager notification provides significant further benefits on top of this optimization. On Intel, improvements range from 46% speedup for value-producing atomic fetch-and-add to 92% for puts. IBM similarly shows speedups from 15% for value-producing

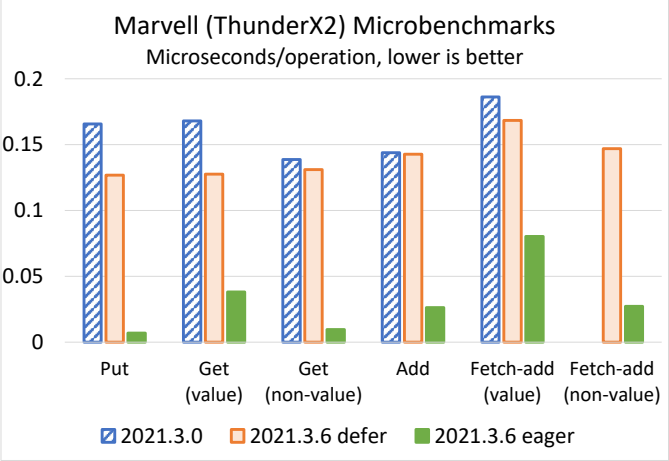

Fig. 4: Microbenchmark results on Marvell.

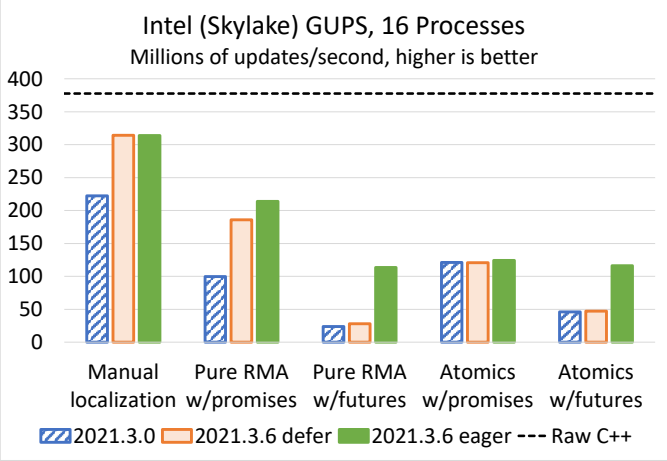

Fig. 5: GUPS results on Intel with 16 processes.

atomics to 95% for puts, and Marvell from 52% for value-producing atomics to 95% for puts.

In addition, our experiments showed that with eager completion, non-value-producing operations that complete synchronously perform significantly better than their value-producing counterparts. Improvements range from 66% for atomic fetch-and-add on Marvell to about 90% for both atomics and gets on IBM.

These results demonstrate the significantly reduced overhead of eager notification for operations that complete synchronously. These performance improvements for on-node operations do not come at the cost of degraded performance for off-node operations (which never complete synchronously and thus always exhibit deferred completion). The code path taken for off-node RMA operations has lengthened by exactly one branch (a dynamic locality check) for deploying eager completion of on-node operations, and the code path taken for off-node atomic memory operations has not changed. A microbenchmark study of off-node RMA performance (omitted due to space limitations) on Intel using two nodes communicating over an EDR InfiniBand network validates that the cost of the additional branch does not have a statistically significant impact on the latency of off-node RMA operations.

B. GUPS

GUPS is an implementation of the HPC Challenge RandomAccess benchmark [1], and it performs randomized fine-grained updates on a distributed table. There are several `UPC++` versions [6], but we focus on the RMA version that uses unsynchronized one-sided operations (some lost updates are permitted), and the AMO version that uses remote atomics.

The RMA version of the benchmark has two locality optimizations:

- When the benchmark is run with processes that all share physical memory on one node, it bypasses `UPC++` entirely, using pure C++ to do the updates. We refer to this as the **raw C++** version. It represents an upper bound on single-node performance of the benchmark.
- Otherwise, the benchmark checks each target global pointer to determine whether it can be dereferenced directly (without an RMA), downcasting it to a local pointer if that is the case. We refer to this as **manual localization**.

We tested both of these optimizations independently. In addition, we examined four versions of the benchmark without these manual optimizations:

- **Pure RMA w/promises**: directly invokes `UPC++` RMA on all global pointers, ignoring locality, using a promise to track overall completion
- **Pure RMA w/futures**: directly invokes `UPC++` RMA on all global pointers, conjoining futures from each RMA together to track overall completion
- **Atomics w/promises**: issues atomic update operations on global pointers, with a promise to track completion
- **Atomics w/futures**: issues atomic update operations on global pointers, conjoining futures to track completion

We ran experiments using 1, 2, 4, 8, and 16 processes. Due to space constraints, we only report the results for 16 processes in Figures 5, 6, and 7; results for other process counts show the same trends as those illustrated here.

The differences between the 2021.3.0 and 2021.3.6 defer results are due to two optimizations that are independent of whether notifications are deferred or eager. The first is that on the SMP conduit, it is always the case that a global pointer is directly addressable, so the `is_local` check was optimized to be `constexpr` and therefore compiled away. This effect can be seen in the manual-localization variant on Intel, where we used the SMP conduit. The second is the same allocation-elimination optimization mentioned above in relation to the microbenchmarks. This explains the difference between the two library versions on the pure RMA w/promises variant.

When it comes to the difference between deferred and eager completions, there is none for the manual-localization variant, as it does not make any calls to RMA. The pure RMA w/promises variant, however, shows a speedup of 15% on Intel, 9% on IBM, and 25% on Marvell when using eager

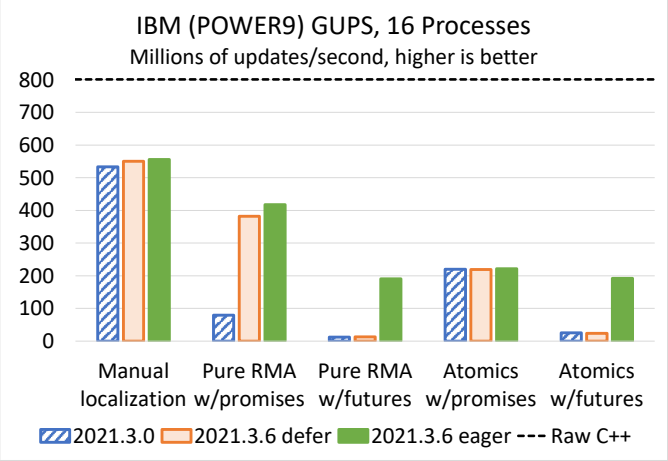

Fig. 6: GUPS results on IBM with 16 processes.

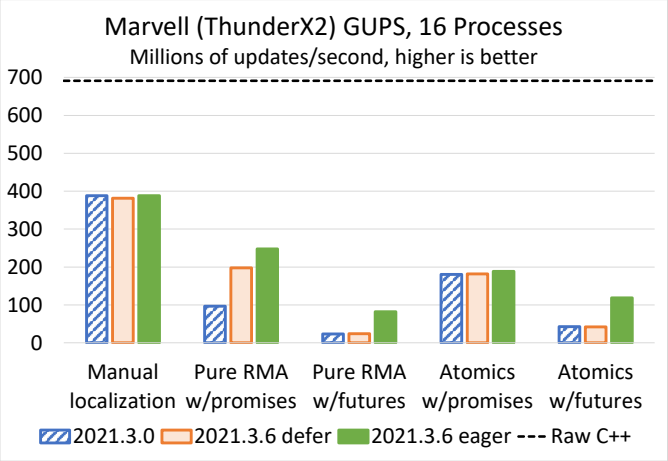

Fig. 7: GUPS results on Marvell with 16 processes.

completion. On the other hand, atomics w/promises show only a negligible speedup of 1-4%; the relatively higher cost of atomic operations overshadow the overheads of deferring completion notifications. The future-conjoining variants show very large speedups for eager completion, as they eliminate both allocation of underlying promises as well as constructing and resolving large dependency graphs. For RMA, the improvements range from 2.4x on Marvell to 13.5x on IBM, and for atomics, from 1.5x on Intel to 7.1x on IBM. In fact, on Intel and IBM, atomics with futures get very close to the performance of atomics with promises under eager notification. This large improvement in overheads for dynamically local operations using future-based completion is the motivation for this work and validates our approach.

Even with eager completions, we do not expect pure RMA to match the performance of the manual localization or raw C++ versions for this benchmark. One important reason relates to the overheads associated with locality checks and global pointer downcasting. The raw C++ version factors locality checks, downcasting and all other `UPC++` calls out of the update loop, amortizing their cost over many updates; however this version only supports single-node runs and does not generalize to distributed-memory execution. The manual localization version supports both local and remote updates, and it performs the local updates in one step that checks for locality and then conditionally downcasts and updates the table:

```
if (dest.is_local()) *dest.local() ^= value;
```

Conversely, the pure RMA versions launch a batch of RMA gets, waits for their completion and then launch RMA puts of the updated values. This multistep process notably incurs twice as many global pointer downcast operations and sacrifices the temporal locality between the read and write of each memory location. Indeed, we see that pure RMA with promises and eager completions does not match the performance of manual localization on any system, but it comes within 25% on IBM, 32% on Intel, and 36% on Marvell. We find that eager completions enable one to maintain a single version of the code that works on distributed memory but still delivers reasonable performance for on-node accesses.

C. Graph Matching

A final set of experiments compares performance of a graph-matching application across the three `UPC++` versions. The application is developed by members of the ExaGraph Co-Design Center at Pacific Northwest National Lab and computes a half-approximate maximum-weight matching over a weighted graph. Their original MPI version of this application is described in this paper [15]. The application authors have also modified the code to use `UPC++` communication for the solve step, and they have previously reported that the `UPC++` RMA version performs comparably to the best MPI version (also using RMA) on up to 2048 processes of a Cray XC (unpublished result). The `UPC++` application code is available online [12]. The application manually optimizes for target memory locations on the same process; however, it does not optimize targets on co-located processes, using `UPC++` RMA operations as it does for processes that are remote.

We collected results with 16 processes on Intel, using four undirected input graphs from the SuiteSparse Matrix Collection [11], and further information about these graphs is available online [2]:

- **Channel**: The *channel-500x100x100-b050* data set, with approximately 4.8 million vertices and 43 million edges.
- **Delaunay**: The *delaunay_n21* data set, with approximately 2.1 million vertices and 6.3 million edges.
- **Venturi**: The *venturiLevel3* data set, with approximately 4.0 million vertices and 8.1 million edges.
- **Youtube**: The *com-Youtube* data set, with approximately 1.1 million vertices and 3.0 million edges.

In addition, we tested with a randomly generated graph (labeled **random**) with 2.0 million vertices and approximately 12 million edges. The graph was generated by running the application with 16 processes and passing `-n 2000000 -p 15` as command-line arguments. The generated graph has edges between vertices within a cutoff Euclidean distance. For each

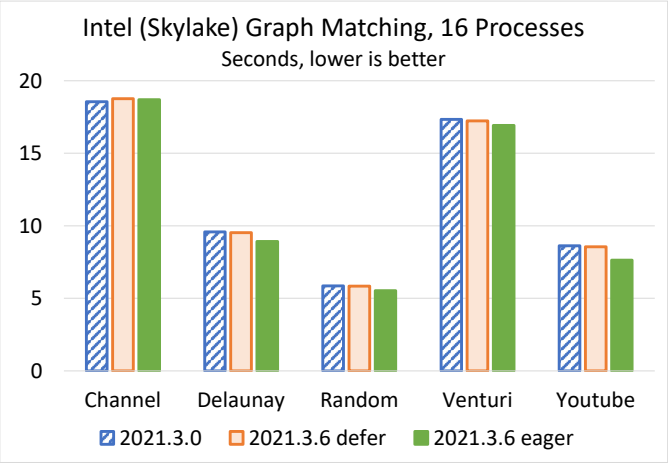

Fig. 8: Graph-matching results on Intel with 16 processes.

100 such edges, the graph contains 15 additional edges between random vertices that are not close together. We modified the code to save the graph to a file and used the same graph across all runs.

Figure 8 shows the running time of the graph-matching solve step for each input. The **channel** graph shows minimal difference between the `UPC++` variants, within experimental noise. The **venturi** graph results in a small 2% improvement in running time for eager completion compared to deferred, while the **random**, **delaunay**, and **youtube** inputs demonstrated more significant decreases of 5%, 6%, and 11%, respectively.

We attribute the differences in these results to the structure of the input graphs. Most edges in the **channel** graph are between nearby vertices, so that most updates are to memory owned by the same process. Such updates are manually optimized by the application, so there is little room for improvement. The **venturi**, **random**, and **delaunay** graphs have somewhat less locality, while the **youtube** input has a highly non-local structure. As a result, significantly more updates are to co-located processes rather than to the same process, and eager notification reduces the overhead of these updates. The overall speedup to the graph-matching application imparted by eager completion notification is of course limited, because the eligible RMA operations we are optimizing comprise only a (graph-dependent) fraction of the overall solve time.

V. Related Work

Many HPC programming models offer explicitly asynchronous communication operations, where the programmer is responsible for issuing a non-blocking operation and later taking explicit action to synchronize its completion before the result is guaranteed to be ready for consumption in subsequent operations. GASNet [7] subdivides nonblocking operations into "explicit-handle" operations (e.g., `gasnet_put_nb()`) whose initiation produces a per-operation non-blocking event handle that is passed to a later call to synchronize completion, and "implicit-handle" operations (e.g., `gasnet_put_nbi()`) whose initiation augments an implicit set of operations that are synchronized as a group during a later call. Unified Parallel C (UPC) [24] similarly provides non-blocking RMA with explicit and implicit handles (e.g., `upc_memcpy_nb` and `upc_memcpy_nbi`), as does Titanium [16, 25]. Other systems whose non-blocking APIs were influenced by GASNet such as Cray DMAPP [3] similarly provide both implicit- and explicit-handle non-blocking operations. DASH [13] provides explicitly asynchronous array copies returning a `dash::Future` that serves as an explicit handle. Similarly, Coarray C++ [17] provides non-blocking coarray read/writes that return a `cofuture` serving as an explicit handle. In all of these cases, explicit-handle initiation calls are permitted to return a *ready handle* (e.g., `GASNET_INVALID_HANDLE`, `UPC_COMPLETE_HANDLE`) to indicate the operation completed synchronously during initiation.

Systems like OpenSHMEM [23], Global Arrays [18], GASPI [14], and MPI one-sided RMA [19] offer implicit-handle non-blocking RMA operations with fence-based synchronization, where prior asynchronous operations (possibly specific to a context, queue or window) are synchronized by issuing a fence or flush call to ensure global completion. Some MPI calls also offer explicit-handle non-blocking events via `MPI_Request` objects, and initiation operations are permitted to synchronously mark the returned `MPI_Request` as complete.

Future-based completion in `UPC++` is a generalization of explicit handles, because `UPC++` allows futures to be chained and conjoined into DAG's expressing asynchronous dependencies between operations and tasks that are processed dynamically as dependencies become satisfied. None of the systems mentioned above currently offer the ability to schedule programmer-provided callbacks for automatic execution upon explicit-handle completion. As such, a semantic that eagerly returns ready handles does not pose a comparable semantic risk that a later callback-scheduling operation might unexpectedly execute the callback synchronously. We are currently unaware of any other programming model that combines explicit-handle nonblocking operations with completion callback scheduling in a manner analogous to `UPC++`.

VI. Conclusions

The PGAS model provides productivity and maintainability benefits by enabling the same code to operate on both local and remote memory. However, asynchronous PGAS systems need to ensure that the mechanisms for asynchrony have minimal impact on the performance of local operations. For `UPC++`, we demonstrated that permitting asynchronous operations to notify completion in an eager manner results in significantly better performance for RMA operations using promises. Combined with optimizations to the `when_all` future combinator, eager notification produces a large improvement in runtime for on-node RMA and atomic operations using futures, up to a 95% speedup in microbenchmarks. These enhancements provide up to a 13.5x speedup in single-node runs of the HPC-Challenge RandomAccess benchmark, and an 11% speedup

in overall solve time for a graph matching application using UPC++ RMA.

Based on these results, we expect the next UPC++ release will adopt eager notification as the default completion mode, with the as_defer factories described in Section III-A as fallbacks for rare cases when deferred notification semantics are preferred. Ongoing future work involves additional optimizations inside the implementation that should transparently further reduce overheads associated with UPC++ operations that can be satisfied on-node.

Acknowledgments

The authors would like to kindly acknowledge Sayan Ghosh for his implementation of the graph-matching application in UPC++ and for providing assistance in selecting inputs and running the application. We'd also like to acknowledge Paul H. Hargrove for assistance in reviewing early drafts of the paper and maintaining the UPC++ installations used at each center.

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.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] "HPC Challenge RandomAccess Benchmark," https://icl.utk.edu/projectsfiles/hpcc/RandomAccess/.

[2] "SuiteSparse Matrix Collection," https://sparse.tamu.edu/.

[3] "Using the GNI and DMAPP APIs," Cray Inc., Technical Report S-2446-5202, October 2014.

[4] J. Bachan, S. B. Baden, D. Bonachea, M. Grossman, P. H. Hargrove, S. Hofmeyr, M. Jacquelin, A. Kamil, and B. van Straalen, "UPC++ Programmer's Guide, Revision 2020.10.0," Lawrence Berkeley Natl. Lab, Tech. Rep. LBNL-2001368, October 2020, doi:10.25344/S4HG6Q.

[5] J. Bachan, S. B. Baden, S. Hofmeyer, M. Jacquelin, A. Kamil, D. Bonachea, P. H. Hargrove, and H. Ahmed, "UPC++: A high-performance communication framework for asynchronous computation," in *33rd IEEE International Parallel & Distributed Processing Symposium (IPDPS'19)*, 2019, doi:10.25344/S4V88H.

[6] S. B. Baden and D. Bonachea, "HPC Challenge RandomAccess in UPC++," https://upcxx.lbl.gov/extras/src/master/examples/gups/upcxx/.

[7] D. Bonachea, "GASNet specification, v1.1," University of California, Berkeley, Tech. Rep. UCB/CSD-02-1207, October 2002, doi:10.25344/S4MW28.

[8] D. Bonachea and P. H. Hargrove, "GASNet-EX: A High-Performance, Portable Communication Library for Exascale," in *Proceedings of Languages and Compilers for Parallel Computing (LCPC'18)*, ser. Lecture Notes in Computer Science, vol. 11882. Springer International Publishing, October 2018, doi:10.25344/S4QP4W.

[9] D. Bonachea and A. Kamil, "UPC++ v1.0 Specification, Revision 2021.3.0," Lawrence Berkeley Natl. Lab, Tech. Rep. LBNL-2001388, March 2021, doi:10.25344/S4K881.

[10] D. Bonachea, "UPC++ as_eager Working Group Draft, Revision 2020.6.2," Lawrence Berkeley Natl. Lab, Tech. Rep. LBNL-2001416, August 2021, doi:10.25344/S4FK5R.

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

[12] ExaGraph Co-Design Center of the Department of Energy Exascale Computing Project (ECP), "Half-approximate Graph Matching in UPC++." [Online]. Available: https://github.com/Exa-Graph/mel-upx

[13] K. Fürlinger, C. Glass, A. Knüpfer, J. Tao, D. Hünich, K. Idrees, M. Maiterth, Y. Mhedheb, and H. Zhou, "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.

[14] GASPI Forum, "GASPI: Global Address Space Programming Interface, Version 17.1," February 2017, http://www.gaspi.de/.

[15] S. Ghosh, M. Halappanavar, A. Kalyanaraman, A. Khan, and A. H. Gebremedhin, "Exploring MPI communication models for graph applications using graph matching as a case study," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 761–770, doi:10.1109/IPDPS.2019.00085.

[16] P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, A. Kamil, B. Liblit, G. Pike, J. Su, and K. Yelick, "Titanium language reference manual, version 2.20," University of California, Berkeley, Tech Report UCB/EECS-2005-15.1, August 2006, doi:10.25344/S4H59R.

[17] T. A. Johnson, "Coarray C++," in *Proceedings of the 7th International Conference on PGAS Programming Models*, ser. PGAS'13, 2013, pp. 54–66. [Online]. Available: https://www.research.ed.ac.uk/portal/files/19680805/pgas2013proceedings.pdf

[18] M. Krishnan, B. Palmer, A. Vishnu, S. Krishnamoorthy, J. Daily, and D. Chavarria, "Global Arrays User Manual," Pacific Northwest National Laboratory, Tech. Rep. PNNL-13130, February 2012.

[19] MPI Forum, "MPI: A message-passing interface standard, version 3.0," University of Tennessee, Knoxville,

Technical Report, September 21, 2012, https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[20] National Energy Research Scientific Computing Center (NERSC). Cori GPU Nodes. Accessed 2021-07-22. [Online]. Available: https://docs-dev.nersc.gov/cgpu/

[21] Oak Ridge National Laboratory Leadership Computing Facility (ORNL/OLCF) . Wombat. Accessed 2021-07-22. [Online]. Available: https://olcf.ornl.gov/olcf-resources/compute-systems/wombat/

[22] Oak Ridge National Laboratory Leadership Computing Facility (ORNL/OLCF). Summit. Accessed 2021-07-22. [Online]. Available: https://olcf.ornl.gov/olcf-resources/compute-systems/summit/

[23] Open Source Software Solutions, Inc. (OSSS), "OpenSHMEM Application Programming Interface, Version 1.5," June 2020, http://openshmem.org/.

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

[25] K. Yelick, P. Hilfinger, S. Graham, D. Bonachea, J. Su, A. Kamil, K. Datta, P. Colella, and T. Wen, "Parallel languages and compilers: Perspective from the Titanium experience," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 266–290, 2007, doi:10.1177/1094342007078449.