The UPC++ PGAS library for Exascale Computing

Extended Abstract

John Bachan, Dan Bonachea, Paul H. Hargrove, Steve Hofmeyr,
Mathias Jacquelin, Amir Kamil, Brian van Straalen, Scott B. Baden
pagoda@lbl.gov
Computational Research Division, Lawrence Berkeley National Laboratory
Berkeley, CA

ABSTRACT

We describe UPC++ V1.0, a C++11 library that supports APGAS programming. UPC++ targets distributed data structures where communication is irregular or fine-grained. The key abstractions are global pointers, asynchronous programming via RPC, and futures. Global pointers incorporate ownership information useful in optimizing for locality. Futures capture data readiness state, are useful for scheduling and also enable the programmer to chain operations to execute asynchronously as high-latency dependencies become satisfied, via continuations. The interfaces for moving non-contiguous data and handling memories with different optimal access methods are composable and closely resemble those used in modern C++. Communication in UPC++ runs at close to hardware speeds by utilizing the low-overhead GASNet-EX communication library.

CCS CONCEPTS

• **Software and its engineering** → **Parallel programming languages**; **Distributed programming languages**; **Concurrent programming structures**; **Software libraries and repositories**;

KEYWORDS

PGAS, Global Address Space, UPC++, Exascale computing

ACM Reference Format:
John Bachan, Dan Bonachea, Paul H. Hargrove, Steve Hofmeyr, Mathias Jacquelin, Amir Kamil, Brian van Straalen, Scott B. Baden. 2017. The UPC++ PGAS library for Exascale Computing: Extended Abstract. In *PAW17: Second Annual PGAS Applications Workshop, November 12–17, 2017, Denver, CO, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3144779.3169108

1 INTRODUCTION

UPC++ [2, 3] is a C++11 library that supports Asynchronous Partitioned Global Address Space (APGAS) programming. UPC++ is well-suited for implementing elaborate distributed data structures where communication is irregular or fine-grained. The UPC++ interfaces for moving non-contiguous data and handling memories with different optimal access methods are composable and closely resemble those used in modern C++.

The key abstractions in UPC++ are: (1) global pointers, that enable the programmer to express ownership information for improving locality, (2) asynchronous remote procedure call (RPC) also known as function shipping, and (3) futures. Futures enable the programmer to capture data readiness state, which is useful in making scheduling decisions, or to chain together high-latency operations, via continuations, to execute asynchronously as dependencies become satisfied.

The UPC++ programmer can expect communication to run at close to hardware speeds. To this end, UPC++ runs atop the GASNet [4] communication library and takes advantage of GASNet's low-overhead communication as well as access to any special hardware support, e.g. RDMA.

The UPC++ project began in 2012 with a prototype designated V0.1, described in [10]. We are revising the library under the auspices of the United States Department of Energy's Exascale Computing Project, to meet the needs of exascale applications requiring PGAS support. This paper describes this new production version, V1.0, which adds several features including a new asynchronous API.

2 UPC++ FEATURES AND DESIGN

UPC++ presents a SPMD programming model, wherein a distributed-memory parallel computer is viewed as an abstract collection of *processing elements* each with a *local memory* (see Fig. 1). Each processing element is called a *rank*, and the number of UPC++ ranks is fixed during program execution.

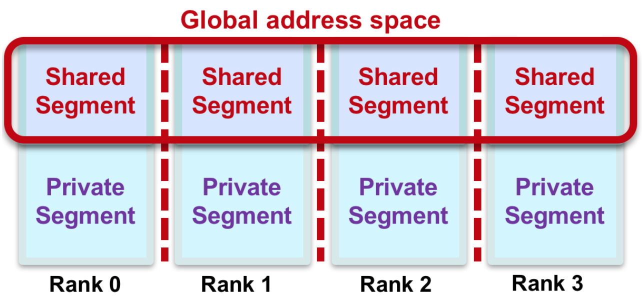

Figure 1: PGAS logical memory model

```
 1 // C++ "global" variables become rank-local state
 2 std::unordered_map<int64_t , upcxx::global_ptr<double> > my_dht_local;
 3
 4 upcxx::future<> insert_data(int64_t key, const double *data_p, size_t data_cnt) {
 5   return upcxx::rpc( key % upcxx::rank_n(),         // send the lambda to this rank
 6     [=]() {                                        // this code runs at the remote rank
 7       // allocate space in shared memory to hold the data (data_cnt doubles)
 8       upcxx::global_ptr<double> gpbuf = upcxx::new_array<double>(data_cnt);
 9       my_dht_local[key] = gpbuf;                   // insert global pointer into hash table
10       return gpbuf;                                // respond by returning buffer location
11     }
12   ).then([=](upcxx::global_ptr<double> gpbuf) {    // process the response at initiator
13     return upcxx::rput(data_p, gpbuf, data_cnt);   // RDMA: put data to the remote buffer
14   });
15 } // return value is a future representing entire asynchronous operation
```

Figure 2: A distributed hash table for variable-sized data in UPC++: insertion operation

As with other PGAS models, ranks can access their respective local memories via conventional C++ pointers. In addition, ranks have access to a global memory, which is allocated in *shared segments* distributed over the ranks. Ranks use *global pointers* to reference objects in any shared segment and move data between them. As with threads programming, references made via global pointers may be subject to race conditions, and appropriate synchronization must generally be employed.

UPC++ global pointers are fundamentally different from conventional C-style pointers. A global pointer refers to a location in a shared segment. It cannot be dereferenced using the `*` operator, nor does it support conversions between pointers to base and derived types. It also cannot be constructed by the address-of operator. On the other hand, UPC++ global pointers *do* support some properties of a regular C pointer, such as pointer arithmetic and passing a pointer by value.

Global pointers are notably used in *one-sided* Remote Memory Access (RMA) communication operations (similar to `memcpy` but across ranks) and in Remote Procedure Calls (RPC). RPC enables the programmer to ship code to other ranks, which is useful in managing irregular distributed data structures. These ranks can push or pull data via global pointers, which can refer to objects in any shared segment.

UPC++'s design philosophy is to provide "close to the metal performance." To meet this requirement, wherever possible, UPC++ will engage low-level hardware support for communication and this capability is crucial to UPC++'s support of *lightweight communication*. In addition, UPC++ imposes certain restrictions. In particular, non-blocking communication is the default for nearly all operations defined in the API, and all communication is explicit. These two restrictions encourage the programmer to write code that is performant and make it more difficult to write code that is not. 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.

Fig. 2 demonstrates several of the core UPC++ features used to implement an asynchronous insertion operation for a distributed hash table storing variable-sized data. Each rank uses a C++ `std::unordered_map` to manage its local partition of the hash table (line 2). The insertion operation begins at line 5 by hashing the provided key to a unique rank id in the parallel job whose partition will hold the inserted value (`upcxx::rank_n()` returns the number of ranks in the job). An RPC is injected to that rank, where upon arrival the code provided by the C++ lambda body on lines 8-10 will execute. Line 8 allocates space in the shared segment of the target rank sufficient to hold the value array (whose element count `data_cnt` was captured by the lambda expression and traveled as part of the RPC). The resulting `global_ptr<double> gpbuf` is a global pointer that can be used by any rank to access the shared buffer. This global pointer is inserted into the local partition of the hash table on line 9, to allow later retrieval by a hash table lookup operation (omitted due to space constraints). The global pointer is also returned from the lambda on line 10, where it will travel back to the initiating rank as a response to the RPC.

The `rpc()` expression is non-blocking and yields a UPC++ future of type `future<global_ptr<double>>` to the initiator. Line 12 uses that future's `.then()` method to schedule a continuation for processing the RPC's eventual response. This continuation is a C++ lambda expression that will execute at the initiating rank and receive the RPC response value `gpbuf`. The lambda body at line 13 uses `upcxx::rput()` to inject a one-sided RMA put operation that transfers the data payload from the local input buffer to its final location (specified by `gpbuf`) in the shared segment of the target rank. This put operation is itself non-blocking and yields a UPC++

future that is returned from the lambda continuation, and propagates outwards to the return of the overall insertion function. Written in this manner, the insertion operation is fully asynchronous, and will return a `future<>` to the caller immediately after initial injection of the RPC, without blocking. The caller uses that future to later synchronize completion of the entire insertion operation, potentially overlapping unrelated work during the communication latencies. This demonstrates the highly-asynchronous coding style that `UPC++` enables and encourages.

`UPC++` avoids non-scalable constructs found in other PGAS systems such as UPC. For example, it does not provide distributed shared arrays nor shared scalars. Instead, it provides *distributed objects,* which can be used to similar ends. Distributed objects are useful in scalably solving the *bootstrapping problem*, whereby ranks need to distribute their local copies of global pointers to other ranks. Though `UPC++` does not directly provide multidimensional arrays, applications that use `UPC++` may define them. To this end, `UPC++` supports non-contiguous one-sided RMA for vector, indexed, and strided data.

The design of `UPC++` avoids introducing hidden threads inside the runtime. The strengths of this approach are improved user-visibility into the resource requirements of `UPC++` and better interoperability with software packages and their possibly restrictive threading requirements. The consequence, however, is that the user must be conscientious to balance the need for making progress (via explicit library calls) against the application's need for CPU cycles.

Ranks may be grouped into teams. A team can participate in collective operations. Teams are also the interface used by `UPC++` to expose the shared-memory capabilities of the underlying system and can let a programmer reason about hierarchical processor-memory organization, enabling various locality optimizations. `UPC++` supports remote atomic operations, currently on 32-bit and 64-bit integers. Atomics are useful in managing distributed lock-free data structures. Unlike C++11 atomics, `UPC++` atomic operations are split-phased, to encourage overlap of unrelated work during the communication latency.

3 UPC++ PERFORMANCE BENEFITS

To demonstrate the benefits of `UPC++`, we present early results with a direct linear solver for sparse symmetric matrices, symPACK [7]. Sparse solvers are key to the solution of numerous science problems, and are well-known for their high communication-to-computation ratio. This makes it challenging to achieve strong scalability on distributed-memory platforms. SymPACK employs `UPC++` one-sided communication to implement a pull strategy. In a direct solver, after a column (or more precisely a supernode) is factored on a given rank, a number of columns (or supernodes) need to be updated. These columns may reside on remote ranks which we refer to as recipients. In symPACK, the recipients are responsible for fetching the data when they are ready to handle it. `UPC++` allows global pointers to dynamically allocated data to be transferred between ranks, which makes this asynchronous pull strategy efficient and easy to implement.

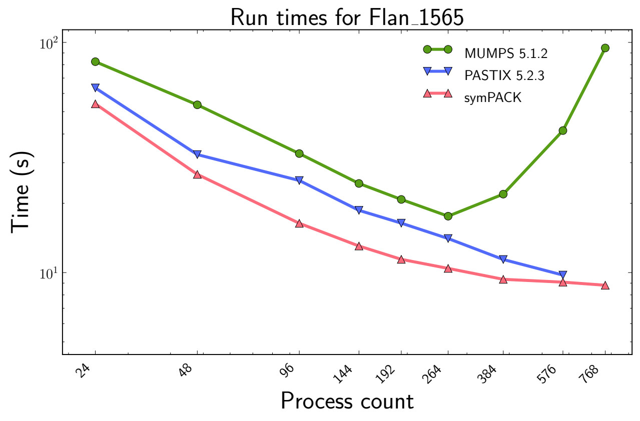

Figure 3: SymPACK/UPC++ vs. competing solvers

We have conducted a preliminary strong scaling experiment on the Flan_1565 input from the SuiteSparse matrix collection [5] on NERSC Edison [8]. Each node is equipped with two 12-core Intel Xeon Sandy-Bridge processors and 96 GB of RAM. We compare the performance of symPACK to that of two state-of-the-art direct linear solvers for sparse symmetric matrices: MUMPS 5.1.2 [1] and PaSTiX 5.2.3 [6]. Nested-dissection computed using SCOTCH [9] was used to reorder the matrices to reduce the amount of fill. No pivoting was used in the experiments, which were done using complex arithmetic. Finally, multithreading was disabled in all solvers as it provided no benefit for this particular sparse matrix.

Figure 3 depicts the strong scalability for the numerical factorization of each solver. As can be seen, symPACK consistently displays lower execution times than the other solvers up to 768 cores (32 nodes), demonstrating the low overhead of `UPC++` and the efficiency of the one-sided pull strategy it enables. The difference can be attributed to the use of one-sided communication in combination with RPC. The other two solvers communicate using non-blocking, two-sided MPI message passing. A pull strategy would be awkward to implement in a message-passing model. MPI RMA would not remedy the expressibility gap due to its lack of support for RPC.

4 CONCLUSIONS

`UPC++` is intended for challenging applications that employ fine-grained or irregular communication. `UPC++`'s support for communication is lightweight, and will deliver hardware offload performance when available, through use of the GASNet-EX communication layer.

`UPC++` provides an APGAS programming model designed for extreme scalability. Its benefits are primarily due to the following attributes:

- Concise and efficient support for distributed irregular data structures in algorithms that move data at a fine granularity, and that have a low computational intensity.
- The ability to move data to code or code to data as best suits the problem, without incurring high programming overheads.
- The ability to execute all communication asynchronously and schedule it to preserve dependencies.
- It leverages modern C++ features and interoperates with MPI, OpenMP and other parallel and distributed programming frameworks.

UPC++ is undergoing further development on additional features, including memory kinds, subset teams and collectives. We shall report on these in more detail at a later time.

5 ACKNOWLEDGMENTS

The UPC++ software is available at http://upcxx.lbl.gov. GASNet is available at http://gasnet.lbl.gov.

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. Early development of UPC++ was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This manuscript has been authored by an author at Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy.

REFERENCES

[1] P. Amestoy, I. Duff, J.-Y. L'Excellent, and J. Koster. 2001. A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling. *SIAM J. Matrix Anal. and Appl.* 23 (2001), 15–41.

[2] John Bachan, Scott B. Baden, Dan Bonachea, Paul H. Hargrove, Steven Hofmeyr, Khaled Ibrahim, Mathias Jacquelin, Amir Kamil, Bryce Lelbach, and Brian van Straalen. 2017. *UPC++ Specification, v1.0 Draft 4.* Technical Report LBNL-2001066. Lawrence Berkeley National Laboratory. http://escholarship.org/uc/item/2nm9n3jm

[3] John Bachan, Scott B. Baden, Dan Bonachea, Paul H. Hargrove, Steven Hofmeyr, Khaled Ibrahim, Mathias Jacquelin, Amir Kamil, and Brian van Straalen. 2017. *UPC++ Programmer's Guide, v1.0-2017.9.* Technical Report LBNL-2001065. Lawrence Berkeley National Laboratory. http://escholarship.org/uc/item/0nq2k8sx

[4] Dan Bonachea and Paul Hargrove. 2017. *GASNet Specification, v1.8.1.* Technical Report LBNL-2001064. Lawrence Berkeley National Laboratory. http://escholarship.org/uc/item/03b5g0q4

[5] T. A. Davis and Y. Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Software* 38 (2011), 1.

[6] Pascal Hénon, Pierre Ramet, and Jean Roman. 2002. PASTIX: a high-performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Comput.* 28, 2 (2002), 301–321.

[7] Mathias Jacquelin, Yili Zheng, Esmond Ng, and Katherine A. Yelick. 2016. An Asynchronous Task-based Fan-Both Sparse Cholesky Solver. *CoRR* abs/1608.00044 (2016). arXiv:1608.00044 http://arxiv.org/abs/1608.00044

[8] NERSC Edison system 2017. (2017). National Energy Research Scientific Computing Center, http://www.nersc.gov/users/computational-systems/edison/configuration.

[9] François Pellegrini and Jean Roman. 1996. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking.* Springer, 493–498.

[10] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. 2014. UPC++: A PGAS Extension for C++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium.* 1105–1114. https://doi.org/10.1109/IPDPS.2014.115