Title

Demonstrating UPC++/Kokkos Interoperability in a Heat Conduction Simulation (Extended Abstract)

Authors

Waters, Daniel
MacLean, Colin A
Bonachea, Dan
et al.

Publication Date

2021-11-19

DOI

10.25344/S4630V

Peer reviewed

Demonstrating UPC++/Kokkos Interoperability in a Heat Conduction Simulation

Daniel Waters, Colin A. MacLean, Dan Bonachea, Paul H. Hargrove
Computational Research Division
Lawrence Berkeley National Laboratory
{DWaters, ColinMacLean, DOBonachea, PHHargrove}@lbl.gov

***Abstract*—We describe the replacement of MPI with UPC++ in an existing Kokkos code that simulates heat conduction within a rectangular 3D object, as well as an analysis of the new code's performance on CUDA accelerators. The key challenges were packing the halos in Kokkos data structures in a way that allowed for UPC++ remote memory access, and streamlining synchronization costs. Additional UPC++ abstractions used included global pointers, distributed objects, remote procedure calls, and futures. We also make use of the device allocator concept to facilitate data management in memory with unique properties, such as GPUs. Our results demonstrate that despite the algorithm's good semantic match to message passing abstractions, straightforward modifications to use UPC++ communication deliver vastly improved performance and scalability in the common case. We find the one-sided UPC++ version written in a natural way exhibits good performance, whereas the message-passing version written in a straightforward way exhibits performance anomalies. We argue this represents a productivity benefit for one-sided communication models.**

***Index Terms*—PGAS, RMA, CUDA, Exascale Computing, Performance Portability, Productivity**

I. Introduction

The Kokkos [1] programming model is designed to allow for parallelism abstraction within a heterogeneous node, but on its own does not extend that parallelism to multiple nodes across a network. There are multiple frameworks available for accomplishing that. In their tutorial repository [2], the Kokkos developers assume that MPI is used for internode communication.

Traditionally, MPI uses a message-passing programming model, where data is explicitly sent from the private address space of one processor to that of another in a two-sided communication. This contrasts with the Partitioned Global Address Space (PGAS) programming model, where the processors involved in a program have both local memory and a segment of globally-shared memory. Data in this global memory is then transferred using one-sided communication. UPC++ [3–5] is a library that implements asynchronous PGAS programming using the GASNet-EX communication layer [6], which aides performance by streamlining communication operations, for example leveraging Remote Direct Memory Access (RDMA) network hardware. To exemplify its interoperability with Kokkos, the UPC++ team sought to demonstrate that our library is both easy to substitute for MPI in situations where the latter is commonly used, and delivers at least comparable performance across a variety of node counts.

II. UPC++ Background

Although UPC++ has many conceptual differences from MPI, they are both libraries for developing SPMD programs on distributed-memory platforms. Programs are typically run with a fixed number of processes. Regarding the memory model, UPC++ allows processes to access both their own local memory and global memory. The latter is distributed across the machine in segments, where each segment has affinity to a unique process. While conventional C++ pointers are suitable for accessing local data, working with data in a remote shared segment requires use of a *global pointer*. Although global pointers have limitations like the inability to be downcast (unless they are local to the calling rank), they are still useful for pointer arithmetic.

UPC++ programs most commonly work with global pointers via Remote Memory Access (RMA) or Remote Procedure Call (RPC) operations. The former is one-sided, and often takes the form of either a *remote get* or *remote put* operation. While an RMA moves data to computation, an RPC moves computation (including any arguments and lambda captures) so it can operate on data at the target process. Remote operations such as these are asynchronous by default in UPC++, meaning that there must be a way to query their completion status. Upon injection into the network, a *future* object (templated on the return type of the remote operation) is returned by default, allowing the programmer to not only check data readiness, but also chain additional asynchronous work dependent on the completion of the communication.

In addition to encouraging performant code through explicit global memory accesses and asynchronous remote operations, UPC++ also makes it transparent to the programmer how the application is using its available computation resources. It does this by avoiding the use of hidden threads inside the runtime. As such, work that arrives via the network (such as RPCs) or depends on the completion of asynchronous operations (such as future callbacks) is executed at well-defined points when the application threads make calls into the UPC++ library. Programmers have the flexibility to choose when this happens by using the `upcxx::progress()` routine.

III. Code Overview

The topic of our study is a heat conduction simulation code contained within the Kokkos team's tutorial example repository. They designed it to demonstrate their software's

 doi:10.25344/S4630V

use alongside MPI. The code is available online [2], and was used as our baseline for comparisons with only minimal modification to instrument performance. Our UPC++/Kokkos version of this example is also available online [7].

The main routines called within each timestep are shown in Listing 1. First, the processes pack each of their boundary cells from the encapsulating `Kokkos::View` (a multidimensional array abstraction) into separate contiguous buffers, using a distinct `cudaStream` per halo boundary. The code does not have periodic boundary conditions, so domains on the object's surface have fewer than six halo boundaries. In an additional stream, temperature calculations for interior cells are performed. Processes wait for their pack operations to complete before exchanging halo elements with their neighbors. Using MPI, this exchange takes the form of posting an `MPI_IRecv` immediately followed by an `MPI_ISend`. Each of these functions takes an `MPI_Request` object, which the program stores in arrays whose length equals the number of neighbors. The `MPI_Requests` are then waited upon for completion in `compute_surface_dT()` before updating temperature values for boundary cells. The `fence` operation ensures that all calculations are done before temperatures across the program's domain are accumulated using a parallel sum reduction to determine the global average temperature.

```
pack_T_halo();
compute_inner_dT();
exchange_T_halo();
compute_surface_dT();
Kokkos::fence();
double T_ave = compute_T();
```

Listing 1: Routines performed in each timestep

IV. Modifications

The main challenge when porting the heat conduction example to UPC++ was redesigning the halo exchanges, which included altering how the corresponding data is stored. While the original example used Kokkos managed `Views` that allocate their memory during construction, an alternative constructor can take a pointer to preallocated memory. This form of the constructor was necessary for our use case since we needed to allocate halo buffers in UPC++ global device memory, something Kokkos is incapable of doing. Downcasting to a local pointer allowed our buffers to be compatible with construction of unmanaged `Views`.

UPC++ has a specialized way of interacting with memory having unique properties (such as that found on CUDA-enabled GPUs), which we refer to as the *memory kinds* interface. UPC++'s abstractions for dealing with device memory center around a `upcxx::device_allocator` object. The entire memory segment that a process will use on a device is allocated during the object's construction. Partitions from this segment are assigned to each halo boundary region using the `device_allocator::allocate()` method.

To properly facilitate remote memory access, neighboring processes need to not only allocate buffers for incoming and outgoing halo elements in global memory, but must also exchange the global pointers to these buffers with neighboring processes during program startup. The UPC++ version facilitates this by encapsulating the buffer for a specific surface and direction in a `upcxx::dist_object`, a class template providing a global name for a location on each process that holds a value (in this case, the corresponding global pointer). The `upcxx::dist_object::fetch` member function is called with the desired process rank as an argument to obtain the buffer's location during program startup.

Pseudocode for the stage of the algorithm at which halo buffers are exchanged is shown in Listing 2. We designed it in a way that preserves the nonblocking communication of the original code so that data movement may be overlapped with computation of the internal cells. As such, after fencing on the output buffer at line 3 to ensure all halo data has been packed, matching calls to `MPI_IRecv` and `MPI_ISend` were replaced with a one-sided call to `upcxx::copy`, which is used for asynchronous data transmission between two global memory buffers, potentially of different memory kinds. In this call, both `outbuf` and `inbuf` arguments represent global pointers to device memory, with the former being local and the latter being remote.

```
1  for (n in neighbors) {
2      // sync Kokkos data packing:
3      n.outbuf.fence();
4      // RMA put halo to remote GPU:
5      upcxx::copy(n.outbuf, n.inbuf,
6          n.outbuf.size(),
7          remote_cx::as_rpc([](){
8              count++;}));
9  }
10 // await incoming copies:
11 while(count < neighbors.size())
12     upcxx::progress();
13 count = 0; // reset for next timestep
```

Listing 2: `exchange_T_halo` pseudocode for UPC++

Instead of tracking completion with an object like `MPI_Request`, our algorithm uses a remote procedure call: a function that is invoked by one process but executes on another using the target process's data. The `upcxx::copy` operation at line 5 specifies a `remote_cx::as_rpc` completion, which instructs UPC++ to enlist an RPC for execution at the target process after arrival of the data payload. In this case, the RPC callback at line 8 increments an arrival counter in the memory of the target process. Just before the surface cell temperature calculations are done, the **`while`** loop at line 11 calls `upcxx::progress`, which invokes the UPC++ runtime progress engine so that asynchronous user-provided operations (like RPCs) can be performed. Once the counter variable reaches the number of neighboring processes, all of

the incoming halo data has arrived in GPU memory at the target and is ready to be consumed by subsequent computation.

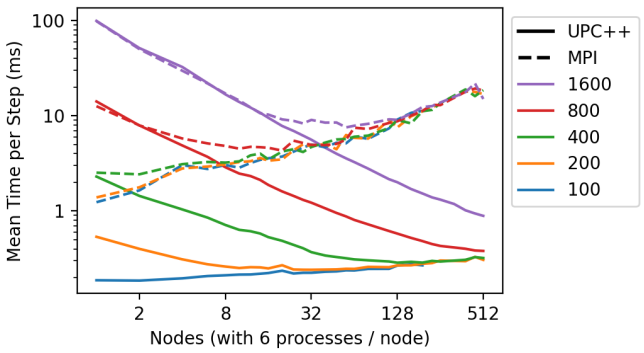
V. Results

Benchmarking data was collected on the supercomputer OLCF Summit [8]. Each node was assigned six processes, mapped 1:1 to each of a node's six NVIDIA V100 GPUs, which were used via Kokkos to perform all the floating-point computations. Runs of the MPI version use the vendor-supplied IBM Spectrum MPI library version 10.3.1.2-20200121 and were configured with CUDA-awareness enabled to maximize performance of transfers to and from GPU memory. Similarly, the UPC++ version was compiled with support for GPUDirect RDMA (GDR), enabling the GPUs and network hardware to perform `upcxx::copy` to and from GPU memory without staging through host memory. This GDR capability [9] of the GASNet-EX communications library has been available in mainline UPC++ releases since 2021.3.0 (the library version measured here). All experiments were compiled with GNU g++ v8.1.1 and utilized Kokkos library v3.4.0 and CUDA v10.1.243.

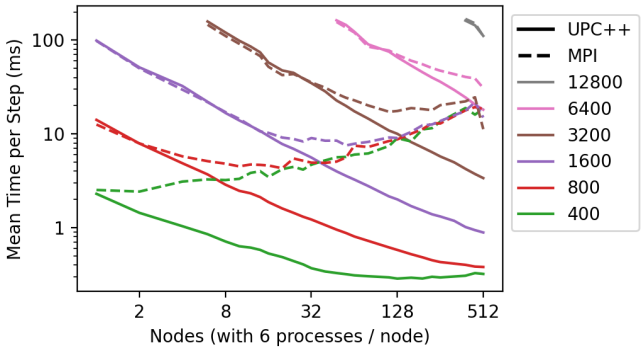
All benchmarks at each node count were executed within the same job to minimize variability of job placement and other effects on network performance. The heat conduction example was executed for 500 timesteps after discarding the first three as warm-up iterations. Problem sizes were chosen as power of two increases from a starting size of 100^3 grid cells until the GPU memory resource limit was reached. Timing data was collected using the Kokkos infrastructure and saved in a `std::map` until program termination to reduce the impact of benchmarking instrumentation in the time-sensitive portion of the program.

Figure 1 shows the measured performance for the UPC++ and MPI versions of the heat conduction example for a range of cube dimensions denoted by the the length of each side. For clarity, the data was plotted separately for small and large problem sizes. In the graph of small problem sizes, fig. 1a, it can be seen that the UPC++ version achieved improved absolute performance at 100^3-200^3 problem sizes for all levels of parallelism. This graph shows the UPC++ and MPI versions approaching different performance limits, with UPC++ both outperforming MPI by a large margin on absolute time and maximum scalability. For instance, it can be seen that with a problem size of a cube with lengths of 800 units, MPI reaches maximum performance at approximately 16 nodes, whereas UPC++ scales to between 256 and 512 nodes for the same problem size. Past a problem size of 400^3, as shown in fig. 1b, the UPC++ and MPI performances are comparable at low levels of parallelism but begin to diverge as UPC++ exhibits improved scalability. The trend of converging towards disparate performance limits continues at these larger problem sizes.

The MPI version of the heat conduction example exhibited a far larger variance in its performance than the UPC++ version, as can be seen in fig. 2. Due to the overlapping of communication and computation and loose synchronization,



(a) Cubes of sizes $100^3 - 1600^3$



(b) Cubes of sizes $400^3 - 12800^3$

Fig. 1: Comparison of UPC++ and MPI performance in the heat conduction example. Colors indicate problem size by the side length of the computed cube while solid and dashed line styles indicate UPC++ and MPI versions respectively.

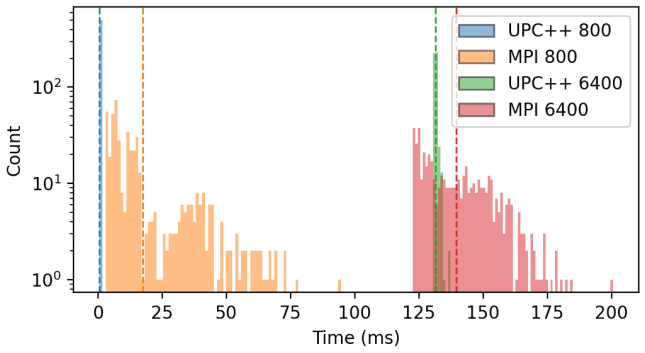


Fig. 2: Histogram of execution time for two time steps (sliding window) on 128 nodes with problem sizes 800^3 and 6400^3. This selection is a representative sample of the variance in MPI vs UPC++ timings. The dotted vertical lines indicate the median of each histogram.

load imbalance may result in early arrival of incoming data in local memory. A sliding window of two time steps is reported to ensure that each interval reflects every portion of one global timestep, with any associated delays. The UPC++ version uses one-sided RMA with GDR acceleration, meaning

the receiving process is not an active participant in receiving the halo data. In the MPI message-passing version, the high communication latency exhibited by some time steps raises the median execution time, leading to degraded performance and scalability.

We speculate that our findings using UPC++ RMA may generalize to other RMA libraries offering similar capabilities, for example MPI "one-sided" RMA using active-target synchronization to enforce the necessary dependencies. However at the time of this writing, IBM Spectrum MPI does not support RMA communication using GPU memory [10]. This limitation prevents leveraging zero-copy GDR hardware support for the HALO exchange as we did in UPC++, and would necessitate an additional host-device copy on each side of every boundary exchange that seems likely to degrade performance. Similar limitations have been reported or observed in some other MPI implementations.

VI. Conclusion

At this time, the authors are not prepared to conclusively prove why the MPI version suffers from a large number of costly outliers in performance. However, one possible explanation is missed rendezvous in the communication. The MPI example is written "naturally," without excessive effort applied towards performance tuning and ensuring that matching `MPI_Irecv`s are posted before the corresponding sends. This could lead to unexpected messages and additional protocol overheads.

The primary advantage of the UPC++ implementation is programmer productivity. As long as both communications libraries are able to effectively leverage the performance of the underlying network hardware and the programmer utilizes each optimally, there should not be vast differences in performance. The key to performance thus becomes a matter of how difficult it is to achieve optimal communications. The one-sided programming model of UPC++ has a significant advantage in this regard, as no effort is needed to ensure that a receive operation is preposted to accept an incoming message. Writing a program to make such guarantees can sometimes require complicated code restructuring and benchmarking to identify regions where such optimizations are necessary. In UPC++, the initiating process simply injects the data transfer operation, and there is no requirement for coordination with the target process. For a well-studied problem such as halo exchange on a regular grid [11], the burden of optimizing the invocation of MPI calls relative to each other and the computation is not a significant obstacle for a programmer with sufficient knowledge and insight. However as the algorithmic complexity and irregularity of the network communication increases, achieving optimal placement of message-passing handshakes becomes an increasing burden to productivity.

Future work of improving the MPI implementation of the heat conduction example will allow confirmation of the reason behind the MPI performance variability.

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 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] Main repository for Kokkos software. [Online]. Available: https://github.com/kokkos/kokkos

[2] "Kokkos MPI heat conduction example," https://go.lbl.gov/paw21-kokkos-mpi-heat-conduction.

[3] J. Bachan, S. B. Baden, S. Hofmeyr, M. Jacquelin, A. Kamil, D. Bonachea, P. H. Hargrove, and H. Ahmed, "UPC++: A High-Performance Communication Framework for Asynchronous Computation," in *Proceedings of the 33rd IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS. IEEE, 2019, doi:10.25344/S4V88H.

[4] 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.

[5] 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.

[6] 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.

[7] "Kokkos UPC++ heat conduction example," https://go.lbl.gov/paw21-kokkos-upcxx-heat-conduction.

[8] 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/

[9] D. Bonachea, "GASNet-EX: A High-Performance, Portable Communication Library for Exascale," Berkeley Lab CS Seminar, March 10, 2021. [Online]. Available: https://gasnet.lbl.gov/pubs/GASNet-2021-LBL-seminar-slides.pdf

[10] IBM, "IBM Spectrum MPI version 10.3 Documentation." [Online]. Available: https://www.ibm.com/docs/en/SSZTET_10.3/doc.pdf

[11] W. Gropp, "Lecture 25: Strategies for parallelism and halo exchange," CS598: Designing and Building Applications for Extreme Scale Systems, 2015, https://wgropp.cs.illinois.edu/courses/cs598-s15/lectures/lecture25.pdf.

REPRODUCIBILITY APPENDIX

A. *Artifact Description*

- Instrumented UPC++ and MPI Heat Conduction Example source code:
  https://bitbucket.org/camaclean/upcxx-extras/src/paw21-kokkos/examples/kokkos_3dhalo/
  - No external data required to initialize
  - There are unresolved bugs with concurrent database writing when saving results. Running multiple benchmarks simultaneously may result in missing data.
  - Harded coded variables to adjust in `upcxx_heat_conduction.cpp`:
    - The location of the sqlite3 database `results.db`
    - The number of GPUs per node recorded for the database entry
    - The network type description recorded for the database entry
- Software Dependencies:
  - GCC v8.1.1
  - Kokkos v3.4.0
  - CUDA v10.1.243
  - IBM Spectrum MPI v10.3.1.2-20200121 with CUDA-awareness enabled
  - UPC++ v2021.3.0
  - Red Hat Enterprise Linux Server release 7.6 (4.14.0-115.21.2.el7a.ppc64le)
- Hardware in each IBM Power System AC922 node of OLCF Summit:
  - Dual-socket 22-core 3.07GHz POWER9
  - Six NVIDIA Tesla V100 CUDA GPUs each with 16 GB HBM2
  - Dual-rail Mellanox EDR InfiniBand with GPUDirect RDMA support
  - 512 GB DDR4-2666 DRAM