ZERO COPY SERIALIZATION USING RMA IN THE HPX DISTRIBUTED TASK-BASED RUNTIME

Author 1
Affiliation
Europe

Author 2
Affiliation
Europe

Author 3
Affiliation
Europe

Author 4
Affiliation
USA

ABSTRACT

Increasing layers of abstraction between user code and the hardware on which it runs can lead to reduced performance unless careful attention is paid to how data is transferred between the layers of the software stack. For distributed HPC applications, those layers include the network interface where data must be passed from the user's code and explicitly copied from one node to another. Message passing incurs relatively high latencies (compared to local copies) from the transmission of data across the network and also from the injection and retrieval of messages into and out of the network drivers which may be exacerbated by unwanted copies of data being made at different levels of the stack. As memory bandwidth is becoming one of the limiting factors in scalability of codes within a node, and latencies of messaging between nodes, it is important to reduce both memory transfers and latencies wherever possible. In this paper we show how the distributed asynchronous task-based runtime, HPX, has been developed to allow zero-copy transfers of data between arguments in user defined remote function invocations and demonstrate the performance of our network layer in a state-of-the-art astrophysics code.

KEYWORDS

Distributed, Task-based, Asynchronous, Runtime, Network, Serialization

1 INTRODUCTION

The HPX runtime system for parallelism and concurrency (Kaiser et al. 2014) is a C++ library that implements asynchronous execution of task (graphs) using futures as synchronization primitives and extends the C++ API with distributed operations using an Active Global Address Space (Kaiser et al. 2015). Asynchronous task launching is performed using an `async` function that is templated over the function type and over a variadic arguments list; this powerful construct allows the user to define any function - with *any number of arguments* - and invoke it asynchronously, returning a future. The C++ standard introduced this function in C++11 as a means of introducing concurrency into the language; when this feature is extended to allow arbitrary function invocations on remote compute nodes as well as the local node, it introduces the need to serialize the arguments into a buffer (marshalling) for transmission and then deserialize them on reception and pass the function arguments onwards to the call being made. When the arguments are small (in terms of bytes of memory required to represent them), then the arguments can be copied into a memory buffer and transmitted as one using an eager protocol, but when the arguments are larger, performance improves when using remote memory access (RMA) operations (using a rendezvous protocol) between nodes to avoid copies and reduce latency – this technique is employed by nearly all HPC message passing systems such as MPI/MVAPICH (W. Huang 2007), PGAS models like UPC (El-Ghazawi et al. 2003), Legion (Bauer et al. 2012).

In addition to implementing the serialization of arguments and RMA transfer between nodes, HPX is a multithreaded task based runtime that makes use of lightweight threads for fast context switching when suspending (or ending) one task and resuming (or starting) another, this means that our implementation must be thread safe (any thread may invoke a remote function at any time) and in order to be used in HPC applications our solution must give high performance.

2 RELATED WORK

There are three areas of research that this work overlaps with, serialization, message passing, and runtime systems – these areas are too large to cover fully in the space provided and we must therefore highlight only those aspects that differ significantly from existing work.

There exist a large number of serialization libraries that are used for RPC purposes (as well as for persisting the state of objects to the filesystem or a database), they can be separated broadly into categories as follows

- Auto generated data requiring an intermediate description and/or pre-processor
- Auto generated but not requiring additional description/compiler
- Manually generated and possibly not strongly typed

Where auto generated means that code necessary to transfer parameters (either streamed/copied/placed) for functions can be generated by using either the native compiler for the system – or the compiler accompanied by an additional preprocessing/compilation step using a tool to transform a user supplied description of structures/data to be transmitted. The principal advantage of using an intermediate description of structures is that serialization between different languages (Java/Python/C++/&etc.) can be handled by the preprocessing step since it can generate different import/export code for each language. A secondary advantage is that it can produce very fast serialization code as the user has supplied type and size information to the preprocessor that makes it easier for the final compilation step to do the right thing. Serializers that fall into this category include Google's protobuf (Google & Varda 2017) and Flatbuffers, Apache thrift (Agarwal et al. 2007) and the charm++ Pack/Unpack (PUP) framework (Kale & Krishnan 1993) and Cap'n Proto (Varda 2015). The principal disadvantage of these libraries is that they require the user to instrument any datatypes that they need to send/receive and/or run a preprocessing step over them before use. In projects that have a limited number of fixed messages/structures/records/types (or vocabulary) this is not a significant workload, but in a runtime system where the user may invoke arbitrary functions with arbitrary parameters, this places an unacceptable burden on the developer, particularly so for projects in their development phase where messages and types can be changing rapidly.

Notable libraries that do not require an intermediate description include the boost serialization library (Boost 1998-2017), the boost MPI library, Cereal. These libraries have the advantage of not requiring additional preprocessing steps and instead require the user to provide a (frequently trivial) serialization function for custom types that usually follows the pattern of Listing 1. (Note that built in types usually have serialization functions provided either as part of the language distribution or the serialization library provides them).

Listing 1: Structure of a typical serialization function, an archive object is given a size and binary data, the templated type of the item being serialized allows the compiler to instantiate the correct specialization

```
template <typename Char>
void serialize(output_archive & ar, const std::string<Char> & s, unsigned) {
    std::uint64_t size = s.size();
    ar << size;
    save_binary(ar, s.data(), s.size() * sizeof(Char));
}
```

The third category that requires manually generating serialization functions includes the MPI library itself and the HPC RPC framework Mercury (Soumagne et al. 2013). With MPI, one can build custom datatypes to represent aggregates of other types and then pass these to the network for transmission, with Mercury one can do the same, but in addition each element may be designated either as a bulk data type or a normal argument. The user must register the function signature and provide a registered memory handle to each bulk data item so that the serialization of arguments can be done using an eager protocol to transfer bulk data handles, followed by a rendezvous phase where each of the bulk arguments is retrieved using an RMA fetch from the remote node. In this respect, Mercury performs essentially the same operation as the zero copy infrastructure in HPX, however, HPX being based on a C++ solution (rather than C) automates a large part of the function and argument registration to greatly simplify the process and place the burden of work on the compiler instead of the user.

When transferring data via RMA, a memory registration process known as pinning is required on both source and destination buffers. The reason for this is to ensure that when the RMA hardware driver initiates a

copy from/into user memory and onto/off the wire, the memory must not have been paged out by the operating system. Registration can be an expensive operation (requiring a kernel level call) and so it is commonplace for networking libraries such as MPI, GASNet (Bonachea 2002), Libfabric (Choi et al. 2015) to provide a pool of registered memory or a registration cache so that repeated requests for registration of memory blocks that are in pages of pre-pinned memory do not incur large costs. Memory registration caching can be a cause of problems as it depends on parameters that are system dependent, generally hidden from the user and can cause system instability or poor performance when incorrectly set. In the HPX messaging layer we expose an `allocator` via a `pool` that provides registered memory and a custom `vector` that makes use of it and may in turn be used for variables that are frequently transmitted between nodes – this places the memory registration in the user's hands rather than leaving it to the system to make decisions - we will further discuss the serialization process in the following sections. PGAS programming models partition distributed memory such the address space spans all nodes and R/W operations to/from nodes are mapped from those addresses to the relevant node, this allows blocks of memory to be 'assigned' to communication on a per node basis – HPX uses an Id type for objects and localities to map addresses in the same way, but there is no block memory reservation made on any given node to represent objects or data on another node.

3 SERIALIZATION

A large number of serialization libraries exist already and the need to re-implement it in HPX was driven by the desire to reduce unwanted memory copies and enable zero copy transfers – to illustrate what can happen in an extreme case Figure 1 shows that 5 copies of data can be created when a transfer is made. We wish to replace this with a single RMA operation between user variables at each end of a connection (when appropriate), to make this possible, HPX adopts a technique referred to as chunking.

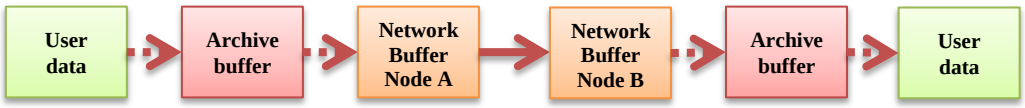

Figure 1: Memory copies that can occur when transferring data from user variables on one node to another.

3.1 Serialization with chunking

To solve the problem of serialization and zero-copy of arguments, HPX uses a chunk-based archive format that differs from 'flat' archives used elsewhere. If we consider the following function invocation

Listing 2: Example of remote action invocation that benefits from a zero copy parameter

```
char x = '5'; double y = 3.1415;
std::vector<float> data(1000000, 2.718);
hpx::future<thing> = hpx::async(action, locality, x, y, data, "string", ...);
```

where `action` represents a remote method, `locality` the Id of a remote node, `thing` an arbitrary return type (whatever the action function returns), and the parameters are typical function arguments, then we would like the small objects to be serialized as usual into a buffer, but the large data vector to be left untouched and instead pass a pointer to the network layer so that it can transfer the object directly without copying. Since the `async` implementation is a variadic template, the compiler can generate the serialization code for us, providing the type of each parameter is known. User defined types must provide a function of the kind shown in Listing 1, built in types and those provided by the STL are supplied by the HPX library, so `std::vector<float>` is automatically handled. The serialization layer creates an archive object that holds a special `chunker` object responsible for tracking blocks of data inserted into the archive - scalar parameters are inserted directly into the archive, however the vector is specialized to call `save_binary` on its data, which terminates the current chunk and writes a pointer chunk containing the vector's data pointer. The next argument may be another large object or a smaller one and depending on the serialization threshold may generate another pointer chunk or start a new index chunk (where the index tracks the size of data being

incrementally written). The process continues until all arguments are written. The archive therefore contains two objects, a raw buffer and a chunk list, both of which are transmitted across the network together.

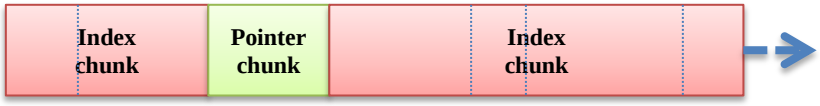

Figure 2: Structure of the chunk based archive generated for a function call similar to that shown in Listing 2, dotted lines delimit how individual sub-elements might be aligned within the index chunks. Pointer chunks always contain a single {pointer, size} entry.

3.2 serialize_buffer

There is unfortunately a problem with Listing 2 that prevents us using the code exactly as shown - when a function is executed asynchronously, one must not allow parameters to be passed by reference, in case they go out of scope and are destroyed before the function is actually executed (which might happen at some arbitrary time later), so parameters may be *moved* or *copied* into an `async` function. We do not wish to copy the vector as this defeats the purpose of the exercise and we may not wish to move the vector as its data might still be required later on the local node. HPX therefore provides a `serialize_buffer` object that wraps the data supplied into a shared pointer and passes it through the `async` API to the parcelport layer in HPX where it is then converted into a pointer chunk for transmission. Listing 3 shows the modified form of the function call. Since the vector is now being passed by reference, it remains the responsibility of the user to ensure that it is not destroyed until transmission is complete (when the returned future becomes ready, it is certain that the message has been delivered).

Listing 3: A serialize buffer is used to wrap data into a form that can be passed by reference using zero-copy

```
hpx::serialize_buffer buff(data.data(), data.size());
hpx::future<thing> = hpx::async(action, locality, x, y, buff, "string", ...);
```

Note that if the remote function does a significant amount of work, the future returned may not become ready until long after the message has been delivered, if the user wishes to reuse buffers immediately after they have been sent, HPX provides an alternative version `async_cb` that allows the user to attach a callback that will be triggered as soon as the parcelport layer has transmitted the data and buffers may be reused.

In the event that the user wishes to send a vector with a smaller size (or other type that has the HPX traits type `is_bitwise_serializable`), the serialization layer will copy data into the index chunk instead of creating a pointer chunk when the size is lower than the zero copy serialization threshold (by default set to 4096 bytes).

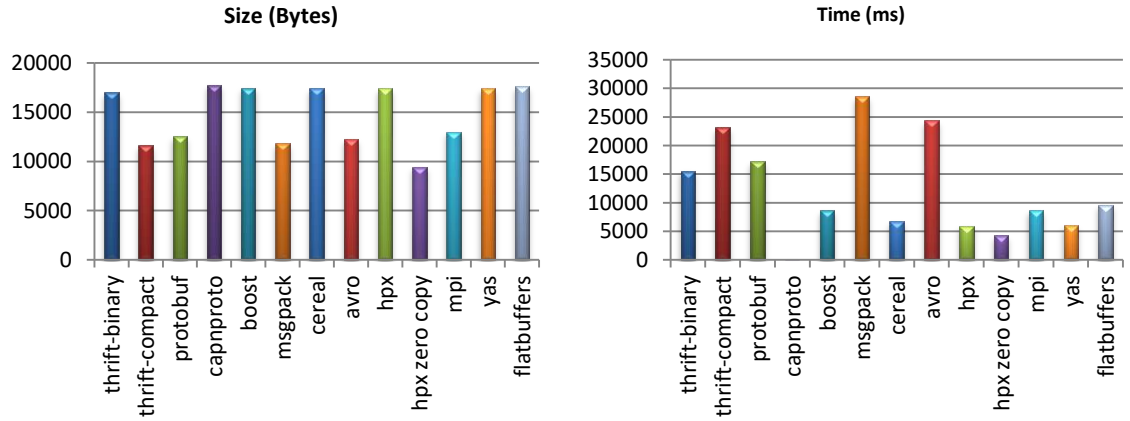

Figure 3: Comparison of serialization libraries. In general, the larger the size, the faster the time, HPX (with zero copy) produces small archives (because pointer chunks are skipped) and achieves good speed for the same reason.

In Figure 3 we show the performance of the HPX serialization layer (with and without zero copy enabled) compared to a number of other libraries using an extended version of the '*C++ serialization comparison tool*' (Sorokin 2017). In the example used, the parameters are not particularly large (20KB) and so the advantages of the pointer chunks is not very significant, but it is clear that HPX performs extremely well. The Cap'nProto library achieves the best performance because it does not actually do any serialization – the archive is simply a pointer to the raw record structure that is to be serialized. Unfortunately this approach cannot be used since it would require all functions to be invoked with a single parameter (a struct) and no zero copy of individual elements would be possible. (If and when static reflection capabilities are added to C++ (Chochlik et al. 2017), then serialization should become both faster and simpler for the user as the compiler will be able to do almost everything).

3.3 Parcelport

The serialized archive (data + chunk list) is passed to the parcelport (network layer) in HPX for transmission. The term 'parcel' is used to refer to the abstraction that represents the serialized form of the archive data, plus chunk list, and also a serialized descriptor of the remote function that must be invoked which is inserted at the start of the archive data (first index chunk); the function type is necessary because HPX uses active messages rather than simple data sends to waiting receivers.

The parcelport is responsible for converting the index and pointer chunks into actual messages that are sent. Multiple parcelport implementations exist in HPX, the default one uses MPI to send the parcel and chunks using non-blocking send + receive calls; the header block is transmitted using the eager protocol and if all data fits inside, then nothing more needs to be done. When pointer chunks are present (or the message exceeds the default (but configurable) 4096 byte limit, then an eager message is sent containing as much of the data as possible, and the remaining chunks must be received in a second round of rendezvous receives. MPI might zero copy pointer chunks or not depending upon how it internally decides to perform the send/receive.

Our new parcelport implementation is based on Libfabric (an implementation also exists using infiniband ibverbs) and follows the same basic procedure except that pointer chunks may be registered on the fly and the memory keys passed in the eager header block so that during the second rendezvous round, data is RMA copied from the source into memory registered on the destination side. The implementation itself is thread-safe and lock-free and integrates cleanly into the HPX tasking runtime.

3.4 Extension to RMA chunks

The ability to copy pointer chunks directly from the user's variables is a significant advancement over the serialization process used in other libraries and gives a significant performance boost to HPX, but there is still a penalty incurred at both ends – registration of user variables for send and registration of memory blocks to receive them into – we can avoid the registration of a blocks on the receive end, by using pre-registered memory, but this means that we must then copy the data from the registered block into the user's variable – otherwise when the memory is freed, it will not be reclaimed correctly unless we intercept the system malloc and maintain a registration cache for example (this is used by some MPI implementations to improve performance and Libfabric also supports this behaviour). The drawbacks to transparently intercepting memory requests in this way are that when data is being frequently allocated and freed by the user and used in messages, the random fragmentation of blocks allocated to the user can lead to excessive pinned memory being reserved by the system since it can only pin whole pages at a time – only by forcing the user to allocate from a particular pool can this be kept under control. We seek a solution that explicitly places control of memory in user's hands and therefore provide an `rma_vector<T>` class that takes its memory from an `rma_allocator` that in turn uses a configurable memory pool provided by the parcelport being used (we aim to support many possible network layers and all have different memory registration handles and APIs) so the pool abstraction must be specialized by each parcelport implementation.

Any datatype that is `is_bitwise_serializable` may be stored in an `rma_vector` and will be serialized by the HPX runtime automatically to produce an RMA chunk that stores the memory registration information needed by the network layer. We also provide a templated `rma_object<T>` for individual objects of significant size that might benefit from zero copy.

Figure 4: An archive may contain RMA chunks: pointer chunks with additional registered memory handles,

The advantages of using an `rma_vector` are that

- Memory is taken from an allocator that is aware of the memory registration API of the network and pinned when the user creates the data structure (or resizes it). It is therefore not pinned/unpinned on the fly during transfer, the RMA chunk constructed during serialization contains the memory handle already.
- The vector is a drop in replacement for `std::vector` with the difference that the copy/assignment constructor makes a shared pointer copy of the internal data so it may be passed as a function parameter without the need to wrap it in a `serialize_buffer`.
- On the receive end of the message, the types of arguments are matched and during deserialization the HPX runtime will construct an `rma_vector` to receive the data into; the received buffer can then be moved into the user variable and when freed will be given to the correct allocator for destruction. The user may declare as many or as few RMA capable objects as needed and does not rely on hidden functions intercepting memory requests to manage resources.

Figure 5 shows the performance of our Libfabric parcelport compared to the existing MPI parcelport, and also the native OSU benchmark using Cray MPI running on Piz Daint, a Cray XC50 with 12 core Intel Xeon nodes and Aries dragonfly interconnect. The performance improvement over the previous HPX implementation is dramatic and we are able to achieve results close to the native MPI benchmark. It is important to note that the MPI OSU benchmark (Ohio-State-University & Panda 2017) implements a ping-pong of messages between two nodes; the native MPI implementation uses a blocking send and matching receive at each end in alternation and therefore does nothing more than poll the network, receive and return the message. As HPX is a task-based runtime using active messages, a ping-pong operation requires a message from one node to another with the action being to invoke a function that creates a message to send the arguments back. The HPX version of the OSU test therefore includes, parcel creation, serialization, deserialization and task creation/management which is why it does not achieve as high a performance as the native MPI implementation.

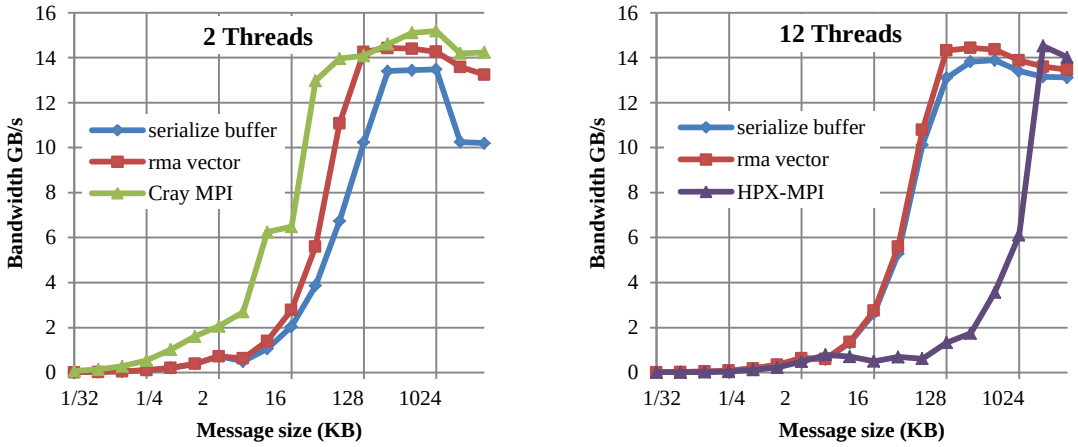

Figure 5: Performance of an HPX version of the equivalent OSU bi-directional bandwidth test between two nodes using the Libfabrics parcelport in HPX on different thread counts and with/without RMA chunking. Also included on (left) is the OSU micro-benchmark using Cray MPI and on (right), the HPX OSU test with the HPX parcelport implemented using MPI asynchronous calls.

Figure 5 shows two important features, the speed improvement of the Libfabric parcelport over the MPI parcelport is evident, however the effects of the RMA chunking can be seen when comparing the `serialize_buffer` and `rma_vector` versions of the test. When many threads are used, the cost of

pinning and unpinning the memory on send and receive is hidden by the fact that many threads may take part in messaging – with 12 threads active, the two versions produce almost identical results. When only 2 threads are used the pre-pinned version with RMA chunks is on average 40% faster than the pointer chunk version for the larger messages sizes (below 4096 bytes, both use the eager protocol with data copied into the network). The difference between the two versions (lower latency, higher bandwidth) translates into a reduction in the idle or busy wait times of the thread pools in HPX which can in turn be used to for other more productive work.

4 APPLICATION RESULTS

The performance of the RMA enabled parcelport is evident in the micro-benchmark test, but we wish to also demonstrate that the reduction in idle time translates into improved application performance. We have therefore tested OctoTiger, a 3D octree based, finite-volume AMR hydrodynamics code with Newtonian gravity; it is a successor to previous hydrodynamics codes described in (Kadam et al. 2017) written using the HPX runtime as the parallelism framework for both on node and distributed operation.

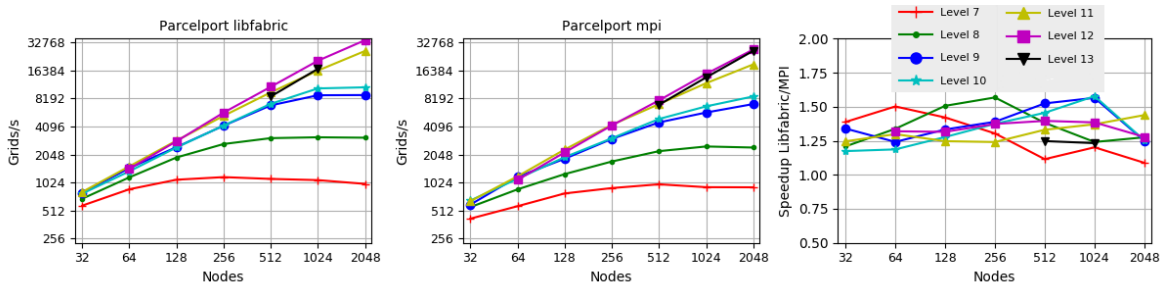

Figure 6: Comparison of the number of AMR grids processed per second for different levels of refinement when using the libfabrics (left) and MPI (middle) parcelports with OctoTiger. The speedup (right) achieved when using libfabrics compared to MPI. All tests were run using 12 cores per node on the Piz Daint supercomputer at CSCS.

Figure 6 shows the number of AMR grids processed per second using different levels of refinement (LoR) with the Libfabrics and MPI parcelports in HPX – (higher levels of refinement cannot be computed on smaller node counts, and lower levels of refinement show reduced performance on higher node counts); a pattern is clear from the results when the speedup is shown. With the exception of the level 7 LoR on high node counts, we see a performance improvement of between 25% and 50% across the board - at 24,576 cores on 2048 nodes, an improvement of 25% is highly significant, and it should be noted that this improvement is not just the messaging parts of the application, but the entire solve step which includes communication via re-gridding as the mesh is adapted/refined.

5 CONCLUSION

We have demonstrated that the serialization layer in HPX is as efficient (or better) than other libraries available and have extended it to support RMA features that make it possible to perform zero-copy RPC calls in distributed HPX task-based applications. In micro-benchmarks it performs well and this performance extends to improved application performance in large scale runs on state of the art supercomputers. Our implementation relies heavily on the strongly typed features of the C++ language and removes most of the burden of message optimization from the user, placing it instead on the compiler and runtime. This improvement to the HPX parcelport layer opens new opportunities for applications built on the HPX runtime and opens the door to exascale development for them.

ACKNOWLEDGEMENT

This work has been partially funded by the European Union Seventh Framework Program (FP7/2007-2013) under grant agreement no. 604102 (HBP). The authors would like to thank the Libfabric developers for their patience and assistance with development.

REFERENCES

Agarwal, A., Slee, M. & Kwiatkowski, M. (2007), Thrift: Scalable cross-language services implementation, Technical report, Facebook. http://thrift.apache.org/static/files/thrift-20070401.pdf

Bauer, M., Treichler, S., Slaughter, E. & Aiken, A. (2012), Legion: Expressing locality and independence with logical regions, *in* 'Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis', SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 66:1–66:11. http://dl.acm.org/citation.cfm?id=2388996.2389086

Bonachea, D. (2002), Gasnet specification, v1.1, Technical report, Berkeley, CA, USA.

Boost (1998-2017), 'Boost: a collection of free peer-reviewed portable C++ source libraries'. http://www.boost.org/. http://www.boost.org/

Chochlik, M., Naumann, A. & Sankel, D. (2017), 'P0350R0: Static reflection', ISO/IEC C++ Standards Committee Paper. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0350r0.pdf

Choi, S.-E., Pritchard, H., Shimek, J., Swaro, J., Tiffany, Z. & Turrubiates, B. (2015), An implementation of ofi libfabric in support of multithreaded pgas solutions, *in* 'Proceedings of the 2015 9th International Conference on Partitioned Global Address Space Programming Models', PGAS '15, IEEE Computer Society, Washington, DC, USA, pp. 59–69. http://dx.doi.org/10.1109/PGAS.2015.14

El-Ghazawi, T., Carlson, W., Sterling, T. & Yelick, K. (2003), *UPC: Distributed Shared-Memory Programming*, Wiley-Interscience.

Google & Varda, K. (2017), 'Protocol buffers', http://code.google.com/apis/protocolbuffers/.

Kadam, K., Clayton, G. C., Motl, P. M., Marcello, D. & Frank, J. (2017), Numerical Simulations of Close and Contact Binary Systems Having Bipolytropic Equation of State, *in* 'American Astronomical Society Meeting Abstracts', Vol. 229 of *American Astronomical Society Meeting Abstracts*, p. 433.14.

Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A. & Fey, D. (2014), HPX: A Task Based Programming Model in a Global Address Space, *in* 'Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models', PGAS '14, ACM, New York, NY, USA, pp. 6:1–6:11. http://doi.acm.org/10.1145/2676870.2676883

Kaiser, H., Heller, T., Bourgeois, D. & Fey, D. (2015), Higher-level parallelization for local and distributed asynchronous task-based programming, *in* 'Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware', ESPM '15, ACM, New York, NY, USA, pp. 29–37. http://doi.acm.org/10.1145/2832241.2832244

Kale, L. V. & Krishnan, S. (1993), Charm++: A portable concurrent object oriented system based on c++, *in* 'Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications', OOPSLA '93, ACM, New York, NY, USA, pp. 91–108. http://doi.acm.org/10.1145/165854.165874

Ohio-State-University & Panda, D. K. (2017), 'MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE'. http://-mvapich.cse.ohio-state.edu/benchmarks/. http://mvapich.cse.ohio-state.edu/benchmarks/

Sorokin, K. (2017), 'Compare various data serialization libraries for c++'. https://github.com/thekvs/cpp-serializers

Soumagne, J., Kimpe, D., Zounmevo, J. A., Chaarawi, M., Koziol, Q., Afsahi, A. & Ross, R. B. (2013), Mercury: Enabling remote procedure call for high-performance computing., *in* 'CLUSTER', IEEE Computer Society, pp. 1–8. http://dblp.uni-trier.de/db/conf/cluster/cluster2013.html#SoumagneKZCKAR13

Varda, K. (2015), 'Cap'n proto'. https://capnproto.org/

W. Huang, G. Santhanaraman, H. J. Q. G. D. P. (2007), Design and implementation of high performance mvapich2: Mpi2 over infiniband.