Zero copy serialization using RMA in the Hpx distributed task-based runtime

Author 1

Affiliation

Europe

Author 3

Affiliation

Europe

Author 2

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.

1. 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 (Grant & Voorhies 2017). 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.

1. 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 (the TCP parcelport in HPX being one example) 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.

**User**

**data**

**Archive**

**buffer**

**Network**

**buffer**

**Network**

**buffer**

**Archive**

**buffer**

**User**

**data**

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

* 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 directy 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.

**Index**

**chunk**

**Pointer**

**chunk**

**Index**

**chunk**

Figure 2: Structure of the chunk based archive similar to that of Listing 2, dotted lines indicate how multiple individual elements might be aligned within the index chunks. Pointer chunks always contain a single item.

Figure 3: The archive is split into chunks, index chunks hold streamed binary data which may come from numerous small items (dotted lines), pointer chunks only hold a pointer and size of the out-of-band data. RMA chunks are pointer chunks with additional memory registration information.

The HPX runtime

In standard C++, a std::async function call is spawned on a new thread of execution (or via a thread pool, depending upon implementation),

allowing concurrent execution of tasks in different threads and also enabling parallelism of algorithms by providing a mechanism whereby they may be broken into sub-tasks and executed on different threads.

The HPX library provides an implementation of hpx::async that conforms to the std::async API but uses thread pools and a lightweight threading model for tasks that does not require a kernel level context switch when changing the thread of executing from one task to another. HPX supports distributed execution of tasks via async calls by allowing them to be executed on remote localities (nodes) which, in turn, requires that the user supplied arguments to remotely invoked functions (or RPCs - Remote Procedure Calls) must be transmitted across the network, a process usually known as serialization

Qthreads (Wheeler et al. 2008)

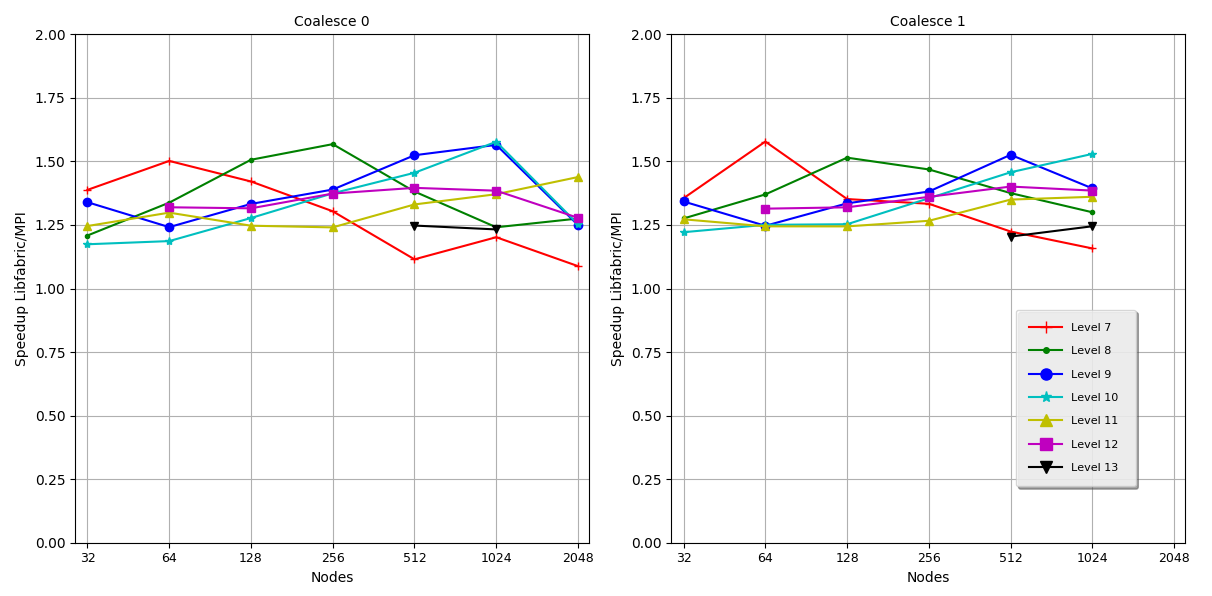
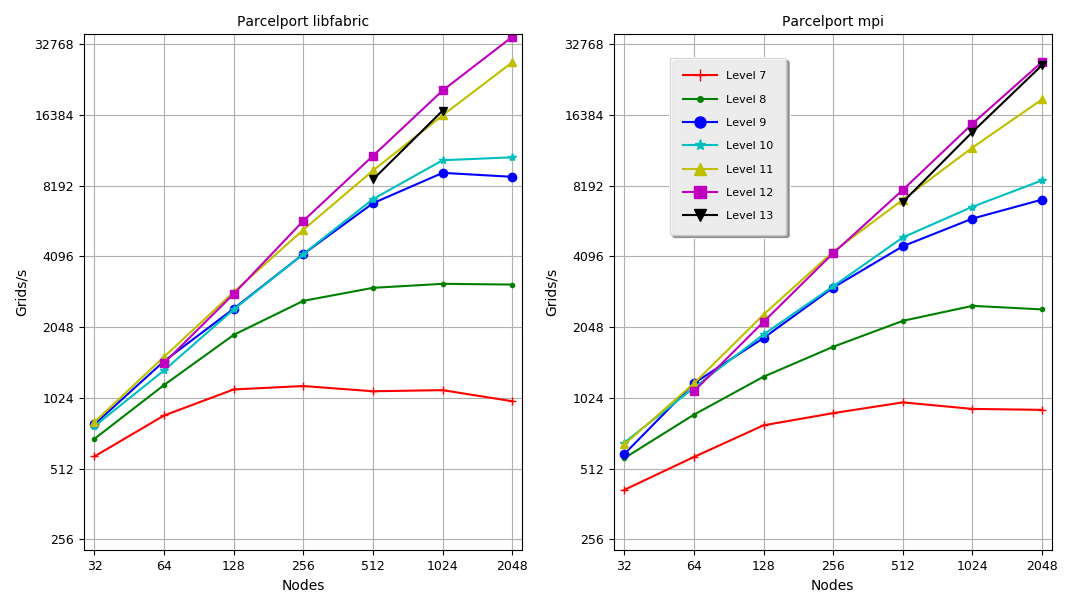
Charm++ (Kale & Krishnan 1993)

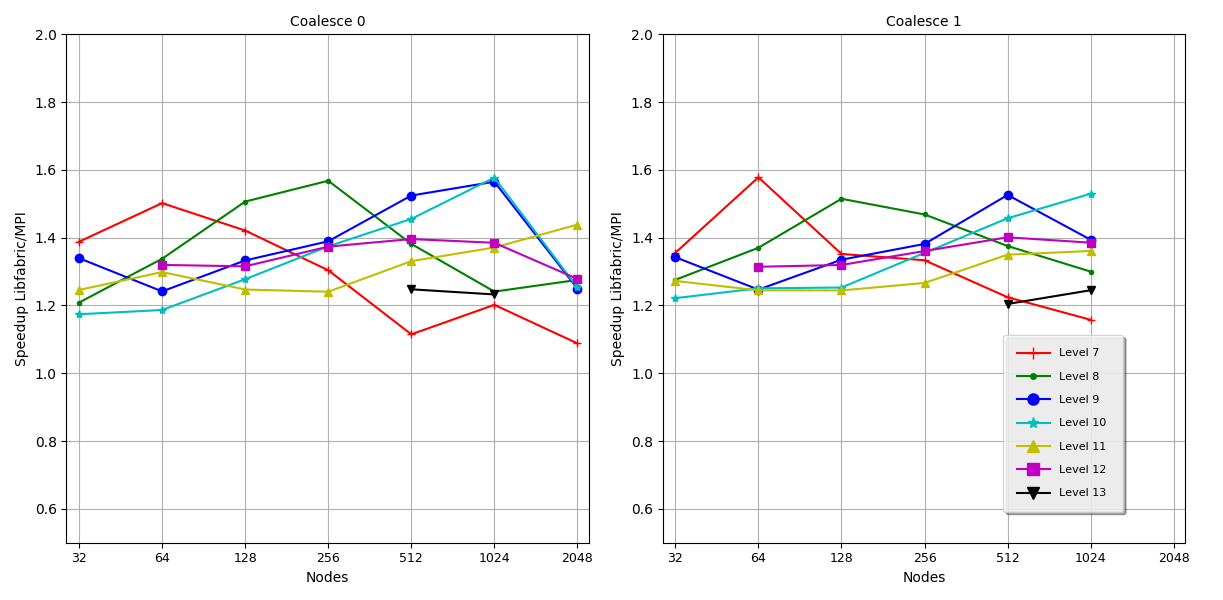
Figure 4: Comparison of serialization libraries. In general, the larger the size, the faster the time, HPX (with zero copy) produces small archives (because pointer chunks are skipped) and achieves good speed for the same reason.

* 1. 2.1 Extension to RMA chunks

Figure 5: Performance of an HPX version of the equivalent OSU BW test between two nodes using Libfabrics and MPI parcelports in HPX

1. Results

Figure 6: Comparison of the number of AMR blocks processed per second for different levels of refinement when using the libfabrics and MPI parcelports with OctoTiger.

Figure 7: The speedup obtained using the Libfabric parcelport compared to the MPI one, (left) results obtained when no parcel coalescing is enabled, (right) results with coalescing. Note that some points are missing as not all combinations of parameter were tested. The coalescing produces a slight reduction in the variance of results.

1. INTRODUCTION

A template is a set of styles and page layout settings that determine the appearance of a document. This template matches the printer settings that will be used in the proceeding and the CD-Rom. Use of the template is mandatory.

Clearly explain the nature of the problem, previous work, purpose, and contribution of the paper.

(Kaiser et al. 2015)

1. BODY OF PAPER

Enter the text here.

* 1. 2.1 Figures and tables

Figures should de numbered consecutively as they appear in the text.

Table 1. Font sizes of headings. Table captions should always be positioned *above* the tables. The final sentence of a table caption should end without a period

|  |  |  |
| --- | --- | --- |
| Heading level | Example | Font size and style |
| First level heading | **1. INTRODUCTION** | 13 point, bold |
| Second level heading  Third level heading | **1.1 Printing Area**  **1.1.1 Text** | 13 point, bold  11 point, bold |

Figures and Tables should be placed as close to their reference point in text as possible. All figures and Tables must have titles and must be referenced from within the text.

Avoid colour images as the proceedings will be printed in black and white.

Table 1. Font sizes of headings. Table captions should always be positioned above the tables. The final sentence of a table caption should end without a period

|  |  |  |
| --- | --- | --- |
| Heading level | Example | Font size and style |
| First level heading | **1. INTRODUCTION** | 13 point, bold |
| Second level heading  Third level heading | **1.1 Printing Area**  **1.1.1 Text** | 13 point, bold  11 point, bold |

Figures and Tables should be placed as close to their reference point in text as possible. All figures and Tables must have titles and must be referenced from within the text.

Avoid colour images as the proceedings will be printed in black and white.

2.1.1 Heading

Enter the text here.

1. CONCLUSION

Clearly indicate advantages, limitations and possible applications.

ACKNOWLEDGEMENT

This work has been partially funded by the European Union Seventh Framework Program (FP7/2007­2013) under grant agreement no. 604102 (HBP).

REFERENCES

[1?] "conf/cluster/2013" not found in database

Agarwal et al. 2007Agarwal, 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 et al. 2012Bauer, 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 2002Bonachea, D. (2002), Gasnet specification, v1.1, Technical report, Berkeley, CA, USA.

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

Choi et al. 2015Choi, 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 et al. 2003El-Ghazawi, T., Carlson, W., Sterling, T. & Yelick, K. (2003), *UPC: Distributed Shared-Memory Programming*, Wiley-Interscience.

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

Grant & Voorhies 2017Grant, W. S. & Voorhies, R. (2017), ‘cereal - a c++11 library for serialization’. http://uscilab.github.io/cereal/

Kaiser et al. 2014Kaiser, 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 et al. 2015Kaiser, 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 & Krishnan 1993Kale, 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

Soumagne et al. 2013Soumagne, 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 2015Varda, K. (2015), ‘Cap’n proto’. https://capnproto.org/

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

Wheeler et al. 2008Wheeler, K., Murphy, R. & Thain, D. (2008), Qthreads: An api for programming with millions of lightweight threads, *in* ‘Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on’, pp. 1–8.