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.

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 invocation 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, but when the arguments are larger, performance improves when using remote memory access (RMA) operations between nodes to avoid copies and reduce latency.

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 and in order to be used in HPC applications must give high performance.

1.1 Related work

.

1. serialization

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 database), they can be separated broadly into categories as follows

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

2.1 Serialization of arguments

To solve the problem of serialization and zero-copy of arguments, HPX uses a chunk-based archive format that differs from ‘flat’ archives, consider the following function invocation

Figure 1: The archive is split into chunks, index chunks hold binary data, pointer chunks only hold a pointer and chunk data size

**Index**

**chunk**

**Pointer**

**chunk**

**Pointer**

**chunk**

**Index**

**chunk**

**Pointer**

**chunk**

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)

Mercury (Soumagne et al. 2013)

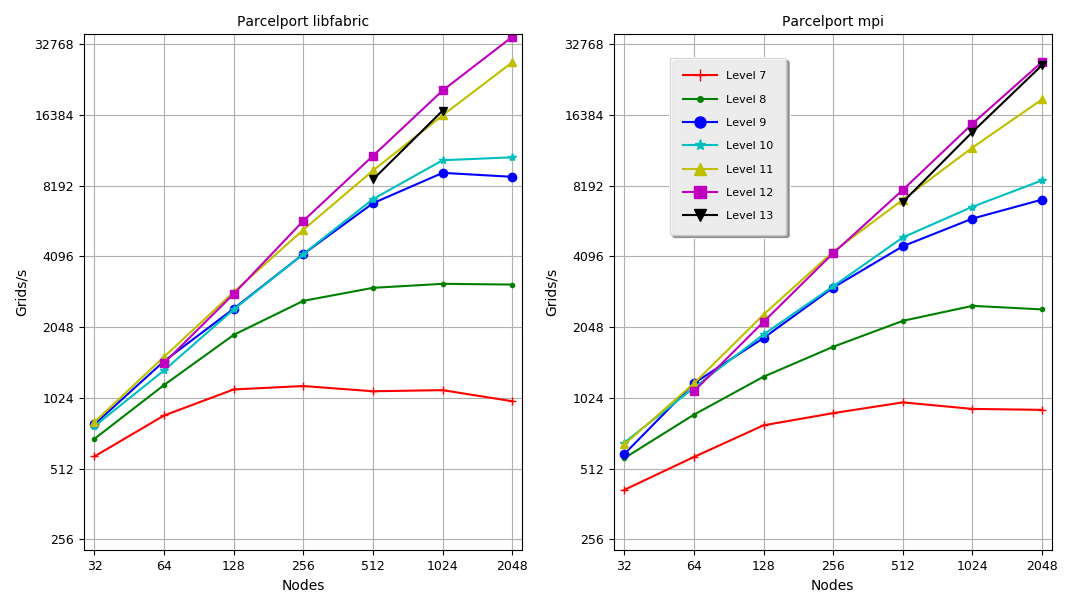


Figure 2: Comparison of the number of AMR blocks processed per second for different levels of refinement when using the libfabrics and MPI parcelports with OctoTiger

#ifndef HPX\_SERIALIZATION\_CONTAINER\_HPP

#define HPX\_SERIALIZATION\_CONTAINER\_HPP

#include <hpx/config.hpp>

#include <hpx/lcos\_fwd.hpp>

#include <hpx/runtime/naming/name.hpp>

#include <hpx/runtime/serialization/basic\_archive.hpp>

#include <hpx/runtime/serialization/binary\_filter.hpp>

#include <hpx/util/assert.hpp>

#include <cstddef>

namespace hpx { namespace serialization

{

struct erased\_output\_container

{

virtual ~erased\_output\_container() {}

virtual bool is\_preprocessing() const { return false; }

virtual void await\_future(

hpx::lcos::detail::future\_data\_refcnt\_base & future\_data) = 0;

virtual bool has\_gid(naming::gid\_type const & gid) = 0;

virtual void add\_gid(

naming::gid\_type const & gid,

naming::gid\_type const & split\_gid) = 0;

virtual void set\_filter(binary\_filter\* filter) = 0;

virtual void save\_binary(void const\* address, std::size\_t count) = 0;

virtual void save\_binary\_chunk(void const\* address, std::size\_t count) = 0;

virtual void reset() = 0;

virtual std::size\_t get\_num\_chunks() const = 0;

virtual void flush() = 0;

};

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.

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

A brief acknowledgement section may be included here.

REFERENCES

References and Citations should follow the Harvard (or author-date) system convention. Check the examples. If a referenced paper has three or more authors the reference should always appear as the first author followed by et al.

**Book**

Author, year. *Title (in italics).* Publisher, location of publisher.

Abiteboul, S. et al, 2000. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, San Francisco, USA.

**Journal**

Author, year. Paper title. *Journal name (in italics)*, volume and issue numbers, inclusive pages.

Bodorik P. et al, 1991. Deciding to Correct Distributed Query Processing. *In IEEE Transactions on Data and Knowledge Engineering*, Vol. 4, No. 3,pp 253-265.

**Conference paper or contributed volume**

Author, year, paper title. *Proceedings title (in italics)*. City, country, inclusive pages.

Beck, K. and Ralph, J., 1994. Patterns Generates Architectures.  *Proceedings of European Conference of Object-Oriented Programming.* Bologna, Italy, pp. 139-149.

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

The C++ Standards Committee 2017The C++ Standards Committee (2017), ISO/IEC Draft International Standard 14882, Programming Language C++, Technical report, Geneva, Switzerland: International Organization for Standardization (ISO).

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.