D4.4.2 – Community prototype for optimized reduction approaches

WP4: Algorithms and Libraries

Project Acronym	CRESTA
Project Title	Collaborative Research Into Exascale Systemware, Tools and Applications
Project Number	287703
Instrument	Collaborative project
Thematic Priority	ICT-2011.9.13 Exa-scale computing, software and simulation

Due date:	M38
Submission date:	30/11/2014
Project start date:	01/10/2011
Project duration:	39 months
Deliverable lead organization	HLRS
Version:	1.0
Status	Final
Author(s):	José Gracia & Vladimir Marjanovic (HLRS)
Reviewer(s)	Luis Cebamanos (UEDIN), Jussi Timonen (JYU)

Dissemination level	
PU	*PU*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	01/11/2014	Initial version	J. Gracia, V. Marjanovic
0.2	11/11/2014	Draft version, submitted for internal review	J. Gracia, V. Marjanovic
0.3	21/11/2014	Addressed reviewer comments	J. Gracia, V. Marjanovic
1.0	27/11/2014	Final version for submission	Catherine Inglis (UEDIN)

Table of Contents

1 Executive Summary

Collective reduction operations such as global summation of a collection of floating point numbers is an important operation in numerical simulations and used for instance as a convergence criterion to control iterative numerical solvers.

The summation of floating point numbers in particular suffers from inaccuracy due to limited numerical precision and round-off errors. Numerical schemes to mitigate the effects of round-off errors in user code and MPI reduce operations have been discussed in a previous document, i.e. D4.4.1 [1].

Based on the work presented in D4.4.1, we have developed the High-Precision Reduction library (libHPR) which offers various algorithms for critical local reductions. Further, we have developed wrappers for MPI collective reductions (libHPRmpi) that allow users to replace the MPI collective reduction, specifically for summation, with a high-precision version.

It is worth noting that the high-precision optimized version of MPI reduce as well as the routine to do local summations are slower than their standard counterparts. The user thus needs to trade off performance for accuracy on a case-by-case basis.

This document is organized as follows: section 2 gives a brief introduction to the topic; section 2.3 lists the various routines provided by the library, and gives details about how to obtain and use the libraries; finally in section 4 we briefly summarize and draw conclusions.

2 Introduction

In computer science, or more specifically in the area of functional programming, *reduction* is defined as a higher-order function which analyses a given data structure, and combines its data elements through use of a given combining operation in order to build up a return value (1). The mathematical expression

$$A = \sum_{i \in C} a_i$$

can be interpreted as A being the result of the reduction with the operation "+" (sum) over the sequence of numbers $\{a_i | \ i \in C\}$ in a given collection C.

In computational science, i.e. numerical simulations, etc., typical examples for the collection are vectors and multidimensional arrays, while sum, product, or minimum and maximum are typical operations.

2.1 Reductions and Numerical Accuracy

One particularly important use of reductions in numerical simulations is the calculation of a single value as a global representative of the individual values in the computation domain. For instance, the average temperature might be used to represent the value of temperature in all individual discretization cells. Also, such global values are frequently used to drive the termination of a recursive algorithm or as convergence criterion in iterative algorithms.

As discussed in more detail in D4.4.1 [1], any numerical computation is affected by round-off errors. While single round-off errors are small, they might accumulate and lead to inaccurate results as the number of individual terms in the sequences grows. This problem specifically arises if the reduction operation is a sum (i.e. a summation over all sequence elements). But it is negligible for multiplication across even large sequences (see [3] for instance).

In principal, there are techniques to mitigate the effects of round-off error accumulation (and some of them have been presented in D4.4.1), however, in some situations reduction operations are not under direct control of the user. One such example is reduction operations in MPI.

2.2 Reductions in MPI applications

The message-passing interface (MPI) provides reductions as collective operations. Each MPI process holds a single element of the global collection. The MPI library will take care of applying the given operation to the global collection while doing communication in the background. However, the user cannot control the numerical scheme to perform for instance the summation and thus cannot control the round-off error (unless the user is prepared to write a custom user-defined reduce function and register this with MPI). In addition, the round-off error depends also on the order in which terms are evaluated. The issue of numerical errors is particularly severe, as MPI reduction collectives are frequently used to calculate global convergence criteria. The inaccuracy of these convergence criteria might have a critical influence on the application's result or its performance.

As we move to exascale computing, with possibly millions of MPI processes, the number of terms in a summation reduction approaches the limit where numerical errors will reach a level that can no longer be disregarded a priori.

2.3 Summation Algorithms

In the previous deliverable, D4.4.1, we have discussed three different summation algorithms:

1. Kahan Summation [4]: keeps a running correction term.

2. Knuth Summation [5,6]: also keeps a running correction term.
3. High Floating-point Precision Summation: uses higher floating-point precision to store intermediate results.

Each of these algorithms has its own advantages and shortcomings. Kahan and Knuth summation are practical only if all summation terms are present locally, which makes them unsuitable for use in MPI collectives. Using higher-precision floating-point numbers to store intermediate results increases the accuracy of every single operation and allows the summation to be distributed over many MPI processes.

3 Library Implementations

We have developed two user-level libraries:

1. High-Precision Reduction library (libHPR)
2. High-Precision Reduction library for MPI (libHPRmpi)

The former (libHPR) provides a set of functions allowing users to do local summations with any of the three summation algorithms: Kahan, Knuth, and high-precision summation. The later (libHPRmpi) substitutes the well-known collective MPI reduction operations *MPI_Reduce()* and *MPI_Allreduce()* with versions that use an extended precision representation of floating-point numbers to increase the accuracy in a way that is transparent to the user.

3.1 High-Precision Reduction Library (libHPR)

This library provides routines to sum up a vector of float or double values, using, respectively, the three algorithms: Knuth summation, Kahan summation, and high-precision summation. The latter uses extended precision floating point numbers internally. In addition, we have provided a routine that does multiplication storing intermediate results in extended precision as well. The names and signatures of the routines are:

- *double HPR_sum_kahan_float(float *vec)*
- *double HPR_sum_kahan_double(double *vec)*
- *float HPR_sum_knuth_float(float *vec)*
- *double HPR_sum_knuth_double(double *vec)*
- *float HPR_sum_highprecision_float(float *vec)*
- *double HPR_sum_highprecision_double(double *vec)*
- *double HPR_prod_highprecision_double(double *vec)*

Programmers should use these routines to protect any non-trivial summation in their codes. The usage of the high-precision multiplication algorithm is only recommended in cases where issues are known to exist.

3.2 High-Precision Collective Reduction Library for MPI (libHPRmpi)

We have developed a library for implementing MPI reduction collectives, in particular we wrap the routines *MPI_Reduce()* and *MPI_Allreduce()*. The wrappers analyse the arguments of the function call and will delegate summation operations on floating-point numbers to a special routine, e.g. *delegate_summation()*. The structure of these delegates is roughly:

```
delegate_summation(sendbuf, recvbuf, count,comm) {
        high_sendbuf = convert2highprecission(sendbuf);
        high_recvbuf = allocate_highprecission();

        op = high_precission_summation;
        ierr = MPI_Reduce(high_sendbuf, high_recvbuf, op, comm);

        recvbuf = convert2lowprecission(high_recvbuf);

        return ierr;
}
```

The send and receive buffers are converted to buffers capable of holding higher precision floating-point numbers. Then the reduce operation is done using the high-precision buffers with a custom reduce operation *op*. For some architectures and compilers/MPI libraries, the operation *op* can be the standard sum operator.

Notably the Cray compiler and Cray MPI do not support extended precision floating-point numbers, and we had to provide a custom *sum_extended_precision* operation compiled with the GNU compiler.

Specifically, we have provided substitutes for the following MPI routines:

- int MPI_Reduce(…, MPI_Datatype datatype, MPI_Op op, …)
- int MPI_Allreduce(…, MPI_Datatype datatype, MPI_Op op, …)

But only for the relevant cases, i.e. *datatype* is one of

- MPI_FLOAT
- or MPI_DOUBLE,

and *op* is one of

- MPI_SUM
- or MPI_PROD.

3.3 Availability of the library

At the time of delivery of this document the source code of the library is available at the HLRS web page

https://www.hlrs.de/index.php?id=2132

The code is distributed under the BSD open source license.

3.4 Usage of the library

Building the library is as simple as executing the command make in the distribution directory:

```
$> cd HPR-library
$> make
```

This will result in two library files

- *libHPR.so*
- *libHPRmpi.so*

which need to be copied into the library path, i.e. any directory listed in LD_LIBRARY_PATH.

The target *test* will build a small test application *test_optimized_reduction*

```
$> make test
$> aprun –n 32 ./test_optimized_reduction
```

3.4.1.1 Using MPI wrappers libHPRmpi

There are two possibilities to use the optimized reduction in a user-provided application: 1) linking against libHPR at compile time, and 2) pre-loading libHPR at runtime.

The library is used at compile/link time and replaces the corresponding MPI routines for all invocations of the application. To do this, compile with the options:

```
$> cc –o a.out –lHPRmpi source_code.c
$> aprun –n 32 ./a.out
```

Note that on non-Cray systems the library HPRlib needs to be invoked after *–lmpi* as:

```
$> gcc –o a.out –lmpi –lHPRmpi source_code.c
$> mpirun –n 32 ./a.out
```

This is not necessary when using the Cray compiler wrappers (CC, cc, ftn) as they load the MPI library, etc, before any user provided library is loaded.

The second option is to overload the MPI reduce functions at the time when the application is executed. This is particularly useful if one wants to test the impact of summation on one's application without the need to recompile it. Use the following commands to accomplish this:

```
$>LD_PRELOAD=libHPRmpi.so aprun –n 32 ./a.out
```

3.4.1.2 High-Precision reductions in user code with libHPR

An application developer can also use the High-Precision Reduction library in user code to perform large summation or multiplication operations at high precision. The developer needs to copy the header file *libHPR.h* into a suitable location, e.g. *HPR_HOME/include*. The routines can be accessed by including them in the source code with:

```
#include "libHPR.h"
```

At compile time, the path to the header file and the library need to be specified with:

```
$> cc –o a.out –IHPR_HOME/include –lHPR source_code.c
$> aprun –n 32 ./a.out
```

4 Conclusions

We have briefly introduced the problem of numerical accumulation of round-off errors in reduction operations, in particular summation over large sequences. Such summation operations are relevant in many simulation applications and are used for instance to calculate convergence criteria, etc.

While there are techniques to mitigate some of the effects of round-off error propagation, there are situations where the application developer cannot easily apply these. One such example is reductions done by MPI on large distributed sequences.

In this report we present the High-Precision Reduction library (libHPR), which provides a simple-to-use routine for local reductions, and wrappers for MPI collective reduction operations (libHPRmpi), which allow users to replace the MPI collective reduction, specifically for summation, with a high-precision version.

Working on this topic we have learnt that few developers of numerical algorithms concern themselves with the impact of numerical round-off errors on their numerical simulations and thus on their scientific conclusions. Today, given the still relatively low number of MPI ranks in the order of a few tens of thousands, this might be justifiable for MPI reductions. However, this is not true for local summations, which can easily impact on sequences with many millions of terms. In the worst case this might bring the true precision of a double float down to the range of a single precision float.

We hope that application developers will soon realize the importance of numerical round-off errors and will find use for the tools we have developed during the CRESTA project.

5 Bibliography

1. **José Gracia, Wahaj Sethi:** *D4.4.1 - Initial prototype for optimized reduction approaches for Project internal validation,* CRESTA Consortium Partners 2011, 2013.

2. Fold (higher-order function): http://en.wikipedia.org/wiki/Reduce_(higher-order_function).

3. **David Goldberg:** "*What every computer scientist should know about floating-point arithmetic",* ACM Comput.Surv. 23(1): 5-48, 1991. doi=10.1145/103162.103163.

4. **William Kahan,** *"Further remarks on reducing truncation errors",* Communications of the ACM 8 (1): 40 : s.n., 1965. doi:10.1145/363707.363723.

5. **Nicholas J. Higham:** "*The accuracy of floating point summation",* SIAM Journal on Scientific Computing 14 (4): 783–799, 1993. doi:10.1137/0914050.

6. **D.E. Knuth:** "*The Art of Computer Programming, vol 2",* Addison-Wesley Press.

7. **Robert W. Robey, Jonathan M. Robey, Rob Aulwes: "***In search of numerical consistency in parallel programming",* http://dx.doi.org/10.1016/j.parco.2011.02.009, 2011.