D4.4.1 – Initial prototype for optimized reduction approaches for Project internal validation

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 Exascale computing, software and simulation

Due date:	M24
Submission date:	30/09/2013
Project start date:	01/10/2011
Project duration:	36 months
Deliverable lead organization	HLRS
Version:	1.0
Status	Final
Author(s):	José Gracia (HLRS), WahajSethi (HLRS)
Reviewer(s)	Luis Cebamanos (EPCC), Gregor Matura (DLR), Jan Westerholm (ABO)

Dissemination level	
PU	*PU - Public*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	01/09/2013	Initial version	J. Gracia (HLRS), W. Sethi (HLRS)
0.2	10/09/2013	Draft version, submitted for internal review	J. Gracia (HLRS), W. Sethi (HLRS),
0.3	15/09/2013	Incorporated recommendations from 1st review	J. Gracia (HLRS), W. Sethi (HLRS), Luis Cebamanos (UEDIN)
0.4	19/09/2013	Incorporated recommendations from reviews 2 & 3	J. Gracia (HLRS), W. Sethi (HLRS), G. Matura (DLR), J. Westerholm (ABO)
1.0	19/09/2013	Final version	D. Khabi (HLRS)

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 convergence criterion to control iterative numerical solvers.

Particularly the summation of floating point numbers suffers from inaccuracy due to limited numerical precision and round-off errors. While there are numerical schemes to mitigate these effects as for instance the Kahan Summation algorithm (see chapter 3 for a discussion of this and other algorithms), collective summations in an MPI application are beyond the control of the user and may introduce large error for large numbers of MPI ranks. However, a priori it cannot be determined whether an application is affected by these numerical inaccuracies and to what extent. The user needs to verify this, possibly for every input data set.

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

We have developed a prototypical version of a library that allows the user to replace the MPI collective reduction, specifically for summation, with a high-precision version. This can be used to test whether an application or a use-case is affected by inaccuracies in the MPI summations.

However, this will only show difference due to inaccuracies in the MPI part, not the computations done locally. To analyse those, we also provide a set of routines to do local summation of a vector of values at high precision. These routines can be used by the application developer in critical sections of the code.

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 deliverable has not evaluated possible support of the networking hardware for summation or other reduction operations. Nonetheless, we recommend adding computing capabilities using high-precision math to the networking interfaces of future Exascale systems as well as to use high-precision buffers for data transport in reduction operations. The performance impact should be minimal with dedicated hardware.

This document is organized as follows: section 2 gives a brief introduction to the topic, section 3 discusses three different summation schemes, section 4 introduces briefly the design of the summation library for MPI and explains its usage.

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 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. Specifically, the L-norm, defined as the sum of the elements of a sequence raised to the power of L, is frequently used as a convergence criterion.

In mathematics, reductions, as any operations, can be evaluated exactly with infinite precision. In numerical analysis, calculations are done at a finite precision, which are affected by round-off errors. So-called *single precision* numbers (*float* in C, 32 bits) can represent roughly only 8 significant digits; *double precision* numbers (*double* in C, 64 bits) can represent roughly 16 significant digits. In other words, the smallest representable relative difference between two numbers is $\varepsilon \approx 10^{-8}$ and $\varepsilon \approx 10^{-16}$, respectively. Adding a number relatively smaller than ε to another larger one will numerically result in a value, which is identical to the larger one due to the limited precision. This is true also if one adds a small number repeatedly to a larger one. For instance let c be $c = \varepsilon/2 = 0.01$ (corresponding to a factious precision of $\varepsilon = 0.02$) then

exact: $1 + c + c + c = (1 + c) + \ c) + \ c) + c) = 1 + 4c = 1.04$

numerical: $1 + c + c + c = (1 + c) + \ c) + \ c) + c) = 1 + O(error)$.

The numerical result is inaccurate and additionally affected by a *round-off error* of order $O(error)$. Both issues, i.e. inaccuracy and round-off error, are particularly severe for summations (and thus subtractions). Multiplication (and division) is considered save due to the way errors propagate (2) (3).

It is worth stressing that any numerical calculation is affected by round-off errors and the aggregated round-off error in most cases increases with the number of terms. *The result of a reduction, particularly summation, over large computational domains with billions or even trillions of degrees of freedom is prone to be affected by round-off errors*.

It is possible to mitigate the issue of round-off error using special algorithms. A few numerical schemes for summation are discussed in section 3 below.

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 were numerical errors will reach a level that can no longer be disregarded a priori.

3 Summation Algorithms

For a naïve implementation, left to right direct summation, the numerical worst-case error grows proportional to the number of terms,n, the mean-square error grows as the square root of the number of terms, i.e. $\sqrt{n}$ (1). Further, the round-off errors are also proportional to the precision of the representation of floating-point numbers, ε. For single and double precision floating-point numbers this is roughly $\varepsilon \approx 10^{-8}$ and $\varepsilon \approx 10^{-16}$, respectively.

3.1 Kahan Summation

Kahan summation (4) keeps a separate running compensate to increase the precision of summation of a sequence of floating-point numbers. With compensated summation, the worst-case error bound is independent of n, so a large number of values can be summed with an error that only depends on the floating-point precision (5).

The running compensate, *correction*, keeps track of the round-off errors lost in the current summation step of the sequence, and is used as an correction term in the next step. The summation routine is roughly:

```
sum = 0;
correction = 0;

for (i = 0; i< size; i++) {
        corrected_next_term = input[i] - correction;
        new_sum = sum + corrected_next_term;
        correction = (new_sum - sum) - corrected_next_term;
        sum = new_sum;
}
```

Advantages and disadvantages:

- Kahan summation works well for long sequences of summations. It is useless if only two terms are summed up, as the correction can only be applied to the next term in the sequence.
- The worst-case and mean-square round-off errors are proportional to $O(\varepsilon)$. The naïve implementation has worst-case and mean-square round-off errors of the order of $O(\varepsilon n)$ and $O\left(\varepsilon\sqrt{n}\right)$, respectively.
- The computational effort is four times as high as compared to the naïve implementation.
- Kahan summation cannot be parallelized. In order to compensate over a large distributed sequence, one would have to calculate sub-sequences in order and transfer the correction term from one the next parallel process to correct across the full sequence. Alternatively one might use Kahan summation only locally and ignore correction across multiple distributed sub-sequences.

3.2 Knuth Summation

Knuth summation works on the same principle as Kahan summation. It calculates the correction term during the summation of two numbers and then this correction term is accounted in next summation. Knuth summation is based on the more detailed formal analysis of precision arithmetic, done by Knuth. This analysis is presented as Theorem B in (6).

Also in this case a correction term is applied to the next summation step. The specifics of the correction term (7) differ from Kahan summation:

$$u + v = (u \oplus v) + correction = (u \oplus v) + ((u \ominus up) + (v \ominus vpp))$$

with

$$up = (u \oplus v) \ominus v$$
$$vp = (u \oplus v) \ominus up$$

Here "+" and "−" represent exact mathematical arithmetic, and "$\oplus$" and "$\ominus$" represents numerical finite-precision arithmetic.

The pseudo code for summation routine is:

```
sum = 0;
c = 0;

for (i = 0; i< size; i++) {
        u = sum;
        v = input [i] + correction;
        upt = u + v;
        up = upt – v;
        vpp = upt – up;
        sum = upt;
        correction = (u – up) + (v -vpp);
}
```

Advantages and disadvantages:

- Similar to the Kahan summation approach, Knuth summation becomes viable, when a large number of arithmetic operations are performed. As the correction term is minute improvements will become visible only as arithmetic operations grow.
- Knuth summation is computationally 7 times more expensive as the normal summation.
- The correction term's absolute value increases as the absolute difference between the operands becomes larger. This makes this summation approach suitable for applications, where data has high variations.
-

3.3 High Floating-point Precision Summation

Doubling the number of significant bits of floating-point numbers, say of precision ε, will yield a precision of ε^2. Often it is relatively simple to use higher precision variables for calculation of intermediate results and thus avoid large round-off errors (relative to the original precision). The pseudo-code for a summation of a sequence of numbers is:

```
float input[n];
float sum = 0;
double dsum = 0;

for (i = 0; i< size; i++)
        dsum = dsum + input[i];   // high precision intermediate result
sum = (float) dsum;               // low precision final result
```

In a distributed setting, one would convert the numbers in the communication buffers to higher precision before actually doing data transfer in order not to lose precision on intermediate results across communication partners.

Advantages and disadvantages:

- Higher precision summation is very flexible and yields better results than lower precision summation in all (but pathological) cases.
- Compared to single precision summation the round-off errors are $O(\varepsilon^2)$, as opposed to $O(\varepsilon)$.

- On modern architectures the floating-point operations on 64-bit (*double*) are nearly as fast as 32-bit (*float*) operations. Many architectures, as for instance x86_64 support an 80-bit *extended precision* mode (*long double*) at no additional cost.
- In a distributed setting the simplest approach is to convert data to high-precision data types at entry to communication routines, do all operations at high precision, and convert back to low-precision on exit from communication routines.

3.4 Discussion

Kahan and Knuth summation algorithms both apply a correction term to the next summation in a sequence and are useless if only two numbers need to be added. They are thus best applicable for large numbers of terms. In a distributed environment one would have to serialize the summation and also transmit the correction terms between processes. This makes these summation schemes very inefficient particularly for large number of processes, as is the target of CRESTA.

On the other hand, increasing the storage size of intermediate results from single to double precision, or double to extended precision, will yield the same or better accuracy (particularly for large number of operations) than Kahan and Knuth summation at little to no additional cost. This is also true in a distributed setting if one uses higher-precision storage formats for all intermediate calculations and messages. The additional time to transfer larger messages might not be very critical, as the latency is dominating message transfer time up to reasonably large message sizes. However, this needs to be judged on a case-by-case basis.

We therefore suggest to use *extended precision* storage formats for calculation and data communication in MPI collective reduce operations.

4 High-Precision Reduction Summation Library for MPI

As discussed above, we have decided to use normal numerical summation but using higher precision, specifically x86_64 *extended precision* (long double in C, 80-bits), as our summation scheme. This yields comparable accuracy as Knuth and Kahan summation at a higher performance and less complexity of the implementation.

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 *delegate_summation()*. The structure of *delegate_summation()* is

```
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);

        returnierr;
}
```

The send and receive buffers are converted to buffer 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.

In addition we provide routines to sum up a vector of float or double values, respectively, using the three algorithms Knuth summation, Kahan summation, and high-precision summation. The last uses extended precision floating point numbers internally. The names and signatures of the routines are:

- *float sum_vec_kahan_float(float *vec)*
- *double sum_vec_kahan_double(double *vec)*
- *float sum_vec_knuth_float(float *vec)*
- *double sum_vec_knuth_double(double *vec)*
- *float sum_vec_highprecision_float(float *vec)*
- *double sum_vec_highprecision_double(double *vec)*

4.1 Availability of the library

At the time of delivery of this document the source code of the library is available through the CRESTA SVN code repository at

https://svn.ecdf.ed.ac.uk/repo/ph/cresta/wp4/optimized_reduction

Access to the CRESTA SVN is subject to the policies of the project. Instructions on obtaining credentials and access to the SVN are available on the project BSCW.

After a testing and evaluation period, the library will be made available through the CRESTA project website or a public code repository and will be distributed under an open source license as e.g. (L)GPL or BSD.

4.2 Usage of the library

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

```
$>cd optimized_reduction
$>make
```

This will result in a library file

libmpi_optimized_reduction.so

which needs 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
```

There are two possibilities to use the optimized reduction in a user-provided application. Either the library is used at compile/link time and replaces the corresponding MPI routines for all invocations of the application. To do so compile with the parameters:

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

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

```
$>gcc –o a.out –lmpi –loptimized_reductionsource_code.c
$>aprun –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 need to recompile it. Use the following commands to accomplish this:

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

5 Conclusions and further work

The accumulation of numerical errors in the summation of large sequences of floating-point numbers can introduce non-negligible errors, particularly as these errors increase in magnitude with the number of summation terms. While there are techniques to mitigate these errors, sometimes the summation is done outside of the control of the user. This is the case for MPI reduction operations.

We have presented three different ways to mitigate the problem and briefly discussed respective advantages and disadvantages. We conclude, that the simplest approach in our context is to transparently convert the payload of MPI reduction operations to extended precision (80 bits) floating-point numbers and the do the summation at this high precision. The result is converted back to lower precision and returned to the user.

The performance impact of this technique has not been evaluated yet and depends also on the optimization done by the compiler. The basic performance difference is doing 64 bits precision math with SSE instructions versus 80 bits precision math with x87 instructions.

This deliverable has not evaluated possible support of the networking hardware for summation or other reduction operations. Nonetheless, we recommend adding computing capabilities using high-precision (a least 80 bits) math to the networking interfaces of future Exascale systems as well as to use high-precision buffers for data transport in reduction operations. The performance impact should be minimal with dedicated hardware.

In collaboration with the application optimization team, we will analyse the impact of optimized reduction, currently summation, on the set of CRESTA applications. We will focus on aspects as:

- binary difference in application results
- scientific relevant difference in application results
- performance impact
- impact convergence behaviour

Depending on the outcome of this analysis, we will tune the implementation of the library to better trade off performance impact for accuracy and allow to user to control this.

Also, we might extend the work to other operations as for instance multiplication, etc.

6 References

[1] Fold (higher-order function). [Online] September 2013. http://en.wikipedia.org/wiki/Reduce_(higher-order_function).

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

[3] Goldberg, David. What every computer scientist should know about floating-point arithmetic. [Online] September 2013. http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html.

[4] Kahan, William. Communications of the ACM 8 (1): 40 : s.n., 1965. doi:10.1145/363707.363723.

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

[6] Knuth, D.E. *The Art of Computer Programming, vol 2.* s.l. : Addison-Wesley Press.

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