Reduction of computational errors

or The pursuit of correctness

Ge Baolai, Western University | Paul Preney, University of Windsor
SHARCNET | Compute Ontario
Digital Research Alliance of Canada | Alliance de recherche numérique du Canada

Outline

- Examples and remedies
- Case study 1: MPI sum
- Case study 2: OpenMP sum
- Case study 3: Multithreading (not included in this talk)
- Future work/topics
- Reference

Keywords: floating point number and accuracy, Kahan compensated summation, reduction, MPI, OpenMP.

Examples

Examples

Example 1: Rounding error. The number 0.01 can't be represented exactly but approximated in base-2 number systems used on computers, so, if our input data contain numbers such as 0.01, we immediately have an error. How much is this error? The following C code shows the result

```
#include <stdio.h>

int main()
{
   float x = 0.01;

   printf("0.01=%20.18f\n", x);
}

0.01=0.009999999776482582
```

Let's see how this approximation comes from.

Examples

Example 1: (cont'd). We use the following julia code to see the binary representation of 0.01 in single precision: 1 sign bit, 8 bit exponent and 23 bit fraction

```
julia> x=Float32(0.01f0)
0.01f0
julia> bitstring(x)
"00111100010001111010111100001010"
```

IEEE 754 floating point number representation for the binary form $(1+f)\times 2^e$ in single precision: 1 sign bit, 8 bits for exponent and 23 bits for fraction.

We convert this by hand, with the math

Exponent minus bias: $e = 2^6 + 2^5 + 2^4 + 2^3 - 127$

Fraction: $f = 2^{-2}+2^{-6}+2^{-7}+2^{-8}+2^{-9}+2^{-11}+2^{-13}+2^{-14}+2^{-15}+2^{-20}+2^{-22}$

Normalized form: $(1+f)2^e = 0.009999999776482582$

Error: $0.01 - 0.009999999776482582 = 2.23517418 \times 10^{-10}$

This kind of errors are inevitable. Keep in mind, however, they can be accumulated and magnified.

Examples

Example 2: Rounding errors in input/output data. Often we use the output from one program and read it in as the input for another. If we are not careful with the format of data stored, we will bring in errors. In the following a is written to a text file a.dat and then read into b, we see errors

a(out)	b(in)	err
0.315349	0.315349	-8.9407e-08
0.907212	0.907212	4.76837e-07
0.510926	0.510926	4.76837e-07
0.86706	0.86706	1.78814e-07
0.634931	0.634931	-1.19209e-07
0.119563	0.119563	-2.23517e-07
0.31076	0.31076	4.76837e-07
0.854888	0.854888	-1.78814e-07
0.446859	0.446859	-2.68221e-07
0.810167	0.810167	5.96046e-08

```
fp = fopen("a.dat","w+");
for (int i=0; i<10; i++)
    fprintf(fp,"%f\n", a[i]);
fclose(fp);


fp = fopen("a.dat","r");
printf("%10s%10s%15s\n", "a(out)", "b(in)", "err");
for (int i=0; i<10; i++) {
    fscanf(fp,"%f", &b[i]);
    err[i] = a[i] - b[i];
    printf("%10g%10g%15g\n", a[i], b[i], err[i]);
}
fclose(fp);
```

Examples

Example 2: (cont'd). One remedy is to save the data in full precision with maximum digits defined by the standards. These are defined in float.h in C and limits in C++

a(out)	b(in)	err
0.315348923	0.315348923	0
0.907212496	0.907212496	0
0.510926485	0.510926485	0
0.867060184	0.867060184	0
0.634930909	0.634930909	0
0.119562775	0.119562775	0
0.310760468	0.310760468	0
0.854887843	0.854887843	0
0.446858734	0.446858734	0
0.810167074	0.810167074	0

```
fp = fopen("a.dat","w+");
for (int i=0; i<10; i++)
   fprintf(fp,"%.*g\n", FLT_DECIMAL_DIG, a[i]);
fclose(fp);
```

```
fp = fopen("a.dat","r");
printf("%10s%10s%15s\n", "a(out)", "b(in)", "err");
for (int i=0; i<10; i++) {
   fscanf(fp,"%f", &b[i]);
   err[i] = a[i] - b[i];
   printf("%10g%10g%15g\n", a[i], b[i], err[i]);
}
fclose(fp);
```

Examples

Example 2: (cont'd). Another way is to save the data in full precision using hex decimal format, supported with %a specifier in C, hexfloat() in C++ and EX edit descriptor in Fortran 2018 (currently supported only by the intel compiler).

a(out)	b(in)	err
0x1.42ead4p-2	0x1.42ead4p-2	0
0x1.d07e28p-1	0x1.d07e28p-1	0
0x1.059828p-1	0x1.059828p-1	0
0x1.bbef5p-1	0x1.bbef5p-1	0
0x1.4515aap-1	0x1.4515aap-1	0
0x1.e9baa8p-4	0x1.e9baa8p-4	0
0x1.3e37fep-2	0x1.3e37fep-2	0
0x1.b5b3dcp-1	0x1.b5b3dcp-1	0
0x1.c99556p-2	0x1.c99556p-2	0
0x1.9ece38p-1	0x1.9ece38p-1	0

```
fp = fopen("a.dat","w+");
for (int i=0; i<10; i++)
   fprintf(fp,"%a\n", a[i]);
fclose(fp);


fp = fopen("a.dat","r");
printf("%10s%10s%10s\n", "a(out)", "b(in)", "err");
for (int i=0; i<10; i++) {
   fscanf(fp,"%a", &b[i]);
   err[i] = a[i] - b[i];
   printf("%10g%10g%10g\n", a[i], b[i], err[i]);
}
fclose(fp);
```

Examples

Example 2: (cont'd). Remarks

- Saving data in maximum number of digits is the simplest way to preserve the accuracy of data. One must pay attention to the format for the portability between languages.
- Hex decimal form keeps the exact representation of data, it is hence more reliable. However, it may not be generally portable between languages, for instance, Fortran code can't read the data output in hexfloat by C/C++, and vice versa.
- C/C++ uses normalized form, Fortran doesn't (as of 2018 standard).
- The EX descriptor in Fortran for hex decimal isn't yet supported as of GCC 14.
- Binary form or more network portable forms are encouraged to be used for data I/O.

Examples

Example 3: Incorrect odometer[1]. The following C code shows a surprise

```
#include <stdio.h>
int main() {
    float meters = 0.0f;
    int iterations = 100000000;
    for (int i = 0; i < iterations; i++) {
        meters += 0.01;
    }
    printf("Expected: %f km\n", 0.01 * iterations / 1000 );
    printf("Got: %f km \n", meters / 1000);
}
```

```
$ gcc error.c
$ ./a.out
Expected: 1000.000000 km
Got: 262.144012 km
```

1. https://jvns.ca/blog/2023/01/13/examples-of-floating-point-problems/

Examples

Example 3: Incorrect odometer[1](cont'd). The cause is during additions, two operands are out of range. A simple fix is to use double precision

```
#include <stdio.h>
int main() {
    float meters = 0.0f;
    int iterations = 100000000;
    for (int i = 0; i < iterations; i++) {
        meters += 0.01;
    }
    printf("Expected: %f km\n", 0.01 * iterations / 1000 );
    printf("Got: %f km \n", meters / 1000);
}
```

```
$ gcc error.c
$ ./a.out
Expected: 1000.000000 km
Got: 262.144012 km
```

```
#include <stdio.h>
int main() {
    double meters = 0.0f;
    int iterations = 100000000;
    for (int i = 0; i < iterations; i++) {
        meters += 0.01;
    }
    printf("Expected: %f km\n", 0.01 * iterations / 1000 );
    printf("Got: %f km \n", meters / 1000);
}
```

```
$ gcc error.c
$ ./a.out
Expected: 1000.000000 km
Got: 1000.000001 km
```

1. https://jvns.ca/blog/2023/01/13/examples-of-floating-point-problems/

Examples

Example 3: Incorrect odometer[1](cont'd). Yet another way, still using float, but more complex – Kahan sum

```
#include <stdio.h>
int main() {
    float meters = 0.0f;
    int iterations = 100000000;
    for (int i = 0; i < iterations; i++) {
        meters += 0.01;
    }
    printf("Expected: %f km\n", 0.01 * iterations / 1000 );
    printf("Got: %f km \n", meters / 1000);
}
```

```
$ gcc error.c
$ ./a.out
Expected: 1000.000000 km
Got: 262.144012 km
```

```
#include <stdio.h>
int main() {
    float meters = 0.0f;
    int iterations = 100000000;
    volatile float c = 0.0f, t, y; // To avoid compiler optimization
    for (int i = 0; i < iterations; i++) {
        y = 0.01f - c;
        t = meters + y;
        c = (t - meters) - y;       // ≡ 0 algebraically
        meters = t;                 // ≡ meters + 0.01f algebraically
    }
    printf("Expected: %f km\n", 0.01 * iterations / 1000 );
    printf("Got: %f km \n", meters / 1000);
}
```

```
$ gcc error.c
$ ./a.out
Expected: 1000.000000 km
Got: 1000.000000 km
```

1. https://jvns.ca/blog/2023/01/13/examples-of-floating-point-problems/

Examples

Example 4: Solving an ill-conditioned problem A*x = b, where

$$A = \begin{bmatrix} 2 & 3.9998 \\ 1 & 2 \end{bmatrix}, \quad \text{let } x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad b = Ax = \begin{bmatrix} 5.9998 \\ 3 \end{bmatrix}.$$

We construct the right hand side (RHS) b by multiplying matrix A by a unit vector.

We now make changes first in b and then in A and examine the solutions in MATLAB

1) Change b by less than 0.001, keep A the same

$$b' = \begin{bmatrix} 5.9998 \\ 3.0003 \end{bmatrix}, \quad \Longrightarrow x' = \begin{bmatrix} -5.7589 \\ 4.3796 \end{bmatrix}.$$

2) Change the elements in A by less than 0.001, keep b the same

$$A' = \begin{bmatrix} 2.0000 & 3.9999 \\ 1.0005 & 2.0002 \end{bmatrix}, \quad \Longrightarrow x' = \begin{bmatrix} -0.8036 \\ 1.9018 \end{bmatrix}.$$

Small changes in the input lead to large changes in the output.

Examples

Example 4: Solving an ill-conditioned problem A*x = b (cont'd). We see two solutions, corresponding to the tiny differences in the RHS and the matrix representation:

$$\mathrm{x}' = \begin{bmatrix} -5.7589 \\ 4.3796 \end{bmatrix},$$

$$\mathrm{x}' = \begin{bmatrix} -0.8036 \\ 1.9018 \end{bmatrix}.$$

This raises the questions, especially when we face real problems:

- How do we know the solution we have obtained is correct?
- How reliable is the computed solution, if we have a small change in the input and we see a large change in the output?
- etc.

Examples

Example 5: Solving system of linear equations H*x = b with Hilbert matrix, with

$$h_{ij} = 1/(i + j - 1).$$

For example, for n=6*

$$H_6 = \begin{bmatrix} 1/1 & 1/2 & 1/3 & 1/4 & 1/5 & 1/6 \\ 1/2 & 1/3 & 1/4 & 1/5 & 1/6 & 1/7 \\ 1/3 & 1/4 & 1/5 & 1/6 & 1/7 & 1/8 \\ 1/4 & 1/5 & 1/6 & 1/7 & 1/8 & 1/9 \\ 1/5 & 1/6 & 1/7 & 1/8 & 1/9 & 1/10 \\ 1/6 & 1/7 & 1/8 & 1/9 & 1/10 & 1/11 \end{bmatrix}.$$

Examples

Example 5: Solving system of linear equations H*x = b with Hilbert matrix (cont'd). Hilbert matrix is ill-conditioned, solving the linear system with an ill-conditioned matrix becomes problematic when n is large.

$$H_6 = \begin{bmatrix} 1/1 & 1/2 & 1/3 & 1/4 & 1/5 & 1/6 \\ 1/2 & 1/3 & 1/4 & 1/5 & 1/6 & 1/7 \\ 1/3 & 1/4 & 1/5 & 1/6 & 1/7 & 1/8 \\ 1/4 & 1/5 & 1/6 & 1/7 & 1/8 & 1/9 \\ 1/5 & 1/6 & 1/7 & 1/8 & 1/9 & 1/10 \\ 1/6 & 1/7 & 1/8 & 1/9 & 1/10 & 1/11 \end{bmatrix}.$$

We construct the RHS b by multiplying H by a unit vector x, i.e. b = H*x.

We then solve H*x = b and see if we get the solution the same, or "close" to the the unit vector.

Note: One may see in the literature the advice that Hilbert matrix is not a good test matrix because it is ill-conditioned. We use it intentionally for the purpose of testing the features of MPLAPACK.

Examples

Example 5: Solving system of linear equations H*x = b with Hilbert matrix (cont'd). Hilbert matrix is ill-conditioned, solving the linear system via LU decomposition may become problematic when n becomes large.

```
Solving H * x = b of size 10

Solution:
          [ 1 ]
             1
     0.9999999
      1.000003
     0.9999758
      1.000116
      0.999682
      1.000522
     0.9994947
      1.000266
     0.9999415
```

```
Solving H * x = b of size 11

Solution:
          [ 1 ]
             1
     0.9999982
      1.000047
     0.9994746
      1.003154
     0.9888376
       1.02444
     0.9665197
      1.027928
     0.9870307
       1.00257
```

```
Solving H * x = b of size 13

Solution:
          [ 1 ]
     0.9999999
      1.000013
     0.9994794
        1.0088
     0.9199938
      1.438536
    -0.5437174
       4.60748
    -4.656326
      6.882018
    -2.890713
      2.481753
     0.7526833
```

```
Solving H * x = b of size 50

Solution:
          [ 1 ]
      1.000001
      0.999892
      1.004564
     0.9173155
      1.790654
     -3.380056
      15.46936
     -26.46469
      25.66145
      1.871251
      -15.6143
      15.50733
     -30.80703
      22.92981
      52.54445
     -54.27587
       ,,, ,,,
```

When n>=12, one will get warning: the matrix is singular to machine precision.

Examples

Example 5: Solving system of linear equations H*x = b with Hilbert matrix, ***revisited***. The RHS b is built by multiplying H by a unit vector x. The solution, however, deviates wildly when n gets bigger.

```
Solving H * x = b of size 10

Solution:
          [ 1 ]
             1
     0.9999999
      1.000003
     0.9999758
      1.000116
      0.999682
      1.000522
     0.9994947
      1.000266
     0.9999415
||dx||_2/||x||_2 = 0.00026777989621392767
```

```
Solving H * x = b of size 12

Solution:
          [ 1 ]
             1
     0.9999867
      1.000421
     0.9942459
      1.042286
     0.8138542
      1.519414
    0.05870534
      2.104559
     0.1904543
      1.336781
     0.9392925
||dx||_2/||x||_2 = 0.51517152763376584
```

Examples

Example 5: Solving system of linear equations H*x = b with Hilbert matrix (cont'd)

- The inverse of Hilbert matrix has a closed form containing integer values. But when n >= 12, some of its values fall beyond what can be represented exactly in IEEE double precision.
- This means one can't get accurate answer by the multiplication of the inverse matrix and the right RHS.

The following is output of the 9th to 13th columns of the inverse of an Hilbert matrix of order 13 in julia*

1891439550//1	-1849407560//1	1160082924//1	-421848336//1	67603900//1
-285985659960//1	282454972800//1	-178652770296//1	65418941952//1	-10546208400//1
10724462248500//1	-10680328659000//1	6802547792040//1	-2505779115840//1	406029023400//1
-174769014420000//1	175266931840000//1	-112296027043200//1	41577371996160//1	-6767150390000//1
1542672646515000//1	-1556276461740000//1	1002242041360560//1	-372734643481200//1	60904353510000//1
-8251094840788800//1	8366542258314240//1	-5412107023347024//1	2020660279013376//1	-331319683094400//1
28450997395460640//1	-28977867717598800//1	18818568211899456//1	-7050482856248832//1	1159618890830400//1
-65321167489578000//1	66791723910912000//1	-43525940081944320//1	16357726068203520//1	-2697888848054400//1
100863567447142500//1	-103492384705710000//1	67651337791837800//1	-25495049614114080//1	4215451325085000//1
-103492384705710000//1	106518477825760000//1	-69822862214785680//1	26379357625420800//1	-4371579151940000//1
67651337791837800//1	-69822862214785680//1	45883595169716304//1	-17374404181470336//1	2885242240280400//1
-25495049614114080//1	26379357625420800//1	-17374404181470336//1	6592659294154752//1	-1096868950850400//1
4215451325085000//1	-4371579151940000//1	2885242240280400//1	-1096868950850400//1	182811491808400//1

* This matrix is generated in julia with SpecialMatrices package. The values are represented in Rational format.

Examples

Example 5: Using MPLAPACK for size n=13 with precision of around 30 digits, we are able to compute the solution with a relative error in the order of magnitude of 10^{-14}.

```
Solving H * x = b with size 13

sol:
+1.000000000000000000019833737841857494393889254788727991192509581e+00
+9.999999999999999969242948730919752968953432873454178776142717289e-01
+1.000000000000000011773889486882940969002326333177844752469684563 9e+00
+9.999999999999980482308732723047657239344617743593409759294762806e-01
+1.000000000000001747800087572751223465981508226717093976653426279 4e+00
+9.999999999999905360423615239651901080140474753238049009920404564 0e-01
+1.000000000000329819971730344280395815656306243408600267644562032e+00
+9.999999999992356977521657943478950632608591484138707528809414393e-01
+1.000000000001189860591369017227118683853904575954407447490462912e+00
+9.999999999987702381277727144025979229541132918715740627874782875e-01
+1.000000000000809103848263016443933566861115827148172915256330029e+00
+9.999999999996932995554366055196135877841182573019827780095789005e-01
+1.000000000000050978832457269717364766462322147851017362400402599e+00
||dx||_2/||x||_2 = 5.8055680326398419e-14
```

MPI reduction for sum

MPI reduction for sum

Example 6: Calculate the summation of an array

$$\sum_{i=1}^{N} a_i$$

Our test: Generate an array a of N random numbers (uniformly distributed on [0,1]) in single precision (so easy to see the effect), then calculate the sum with a loop in the following pseudo code:

```
s = 0;
for i in 1 to N
    s += a[i];
end
```

MPI reduction for sum

Example 6: Summation of an array (cont'd). We first calculate the sum in two ways:

1) Sequential

```
s = 0;
for i=1:N
    s += a[i];
end
```

2) Alternating

```
s = 0;
for i=odd numbers
    s += a[i];
end
for i=even numbers
    s += a[i];
end
```

We know with floating point arithmetic, we lose the associativity. Looking at two answers

Sum of a (seq): 500156.750000

Sum of a (alt): 500148.468750

Which one is more close to the truth? How significant is the difference to your work?

MPI reduction for sum

Example 6: Summation of an array (cont'd). We calculate the sum in two ways:

1) Sequential

```
s = 0;
for i=1:N
    s += a[i];
end
```

2) Alternating

```
s = 0;
for i=odd numbers
    s += a[i];
end
for i=even numbers
    s += a[i];
end
```

We know with floating point arithmetic, we lose the associativity. Looking at two answers

Sum of a (seq): 500156.750000

Sum of a (alt): 500148.468750

Which one is more close to the truth?

To see the answer "close to the truth", we perform the summation in double precision and get

Sum of a (in double precision): 500144.934192

This is what we trust (as an accuracy of more digits is guaranteed by IEEE standard).

Can we get the same accuracy with single precision?

MPI reduction for sum

Example 6: Summation of an array (cont'd). We calculate the sum in single precision using the Kahan summation. We chose the Kahan-Babuška-Neumaier algorithm

```
s = 0.0, c = 0.0;
for i=1:N
   t = s + a[i];
   if (|s| >= |a[i]|)
      c += (s - t) + a[i];        // Algebraically c ≡ 0, but in floating point arithmetic it is not
   else
      c += (a[i] - t) + s;        // Algebraically c ≡ 0, but in floating point arithmetic it is not
   s = t;
end
s += c;
```

We get the answer

Sum of a (Kahan-Babuška-Neumaier compensated): 500144.937500

compared with the answer in double precision

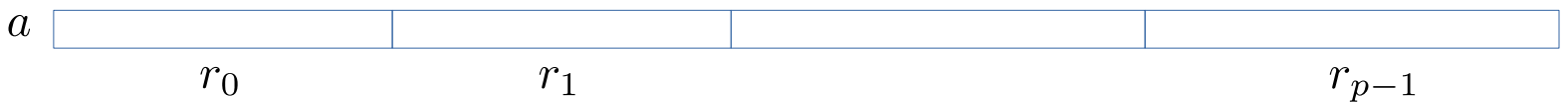
Sum of a (in double precision): 500144.934192

MPI reduction for sum

Example 7: Calculate the summation of an N element array a

$$\sum_{i=1}^{N} a_i$$

distributed across p processes



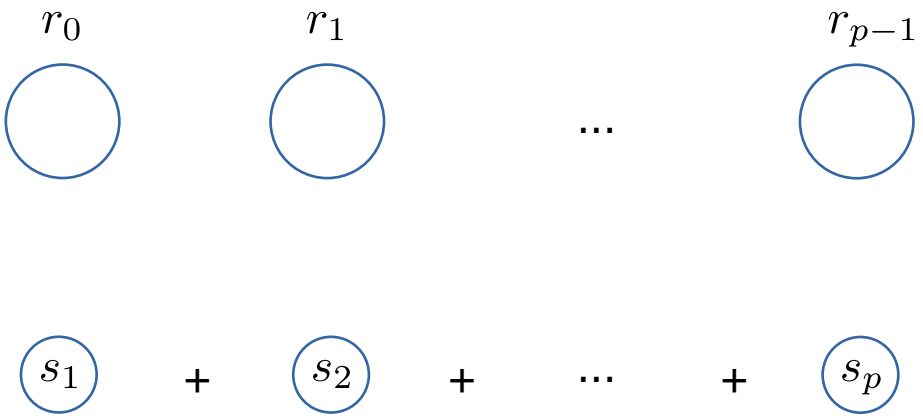
The procedure

- Each rank computes the local sum s_i, $i=1,...,p$.
- An MPI reduce is called on the local sum to get aggregated global sum.

MPI reduction for sum

Example 7: Using reduce to get sum across processes with MPI

- Each process holds one value.



MPI reduction for sum

Example 7: Calculate the summation of an N element array a

$$\sum_{i=1}^{N} a_i$$

distributed across p processes. The results with $N{=}1{,}000{,}000$

Number of processes	Summation
1	500156.75
2	500136.53
4	500144.43
8	500143.71
16	500145.28
64	500144.84

MPI reduction for sum

Example 7: Calculate the summation of an N element array a

$$\sum_{i=1}^{N} a_i$$

distributed across p processes with Kahan-Babuška-Neumaier compensation algorithm

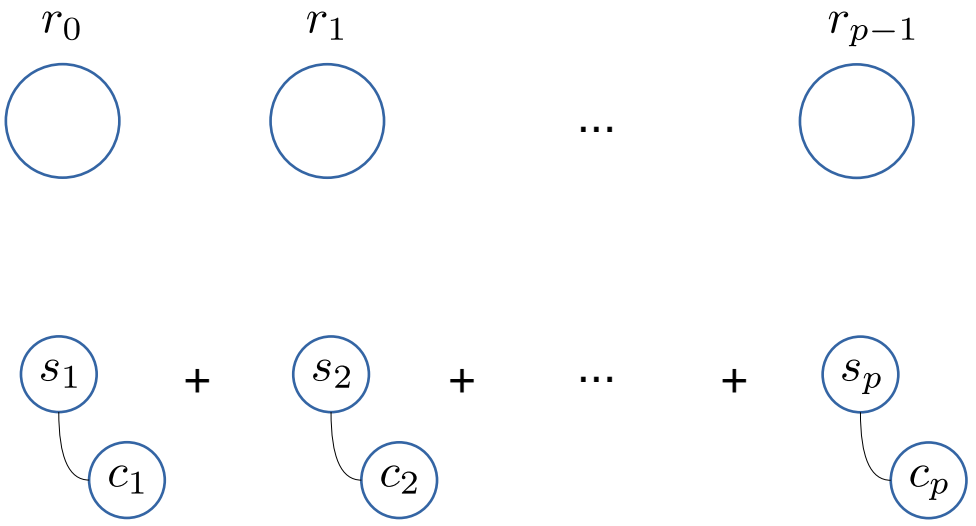
The procedure

- Each rank computes the local sum s_i, $i=1,...,p$ with KBN algorithm
- An MPI reduce with a customized sum operation with KBN procedure is called on the local sum to get aggregated global sum.

MPI reduction for sum

Example 7: Using reduce to get sum across processes with MPI

- Take two values from each process: the local sum and the compensation to feed KBN procedure
- Need a customized sum operator to perform the sum



MPI reduction for sum

Example 7: (cont'd). We use MPI_Reduce() with customized sum operation

```
MPI_Reduce(
    const void *sendbuf,          // Local values
    void *recvbuf, int count,     // Global results
    int count,                    // Number of elements
    MPI_Datatype datatype,        // Data type
    MPI_Op op,                    // Aggregation operation
    int root,                     // Root rank
    MPI_Comm comm                 // Communicator
)
```

Customized sum function (called by MPI_Reduce())

```
void my_sum(
    void *invec,          // invec[0] contains the local sum
    void *inoutvec,       // inoutvec[0] contains the (partial) global sum
                          // and inoutvec[1] carries the running global
                          // compensation
    int *n,               // *n = 2, only relevant to inoutvec
    MPI_Datatype *pdt     // MPI type, not used
)
{
    ... ...
}
```

In the calling routine, we place

```
MPI_Type_contiguous(1,MPI_FLOAT,&dtype);
MPI_Type_commit(&dtype);
MPI_Op_create(my_sum,&my_sum_op);
MPI_Reduce(&lsum,&gsum,2,dtype,my_sum_op,0,comm)[1];
```

1. Compared with the regular call MPI_Reduce(&lsum,&gsum[0],1, MPI_FLOAT,MPI_SUM,0,comm);

MPI reduction for sum

Example 7: Calculate the summation of an N element array a

$$\sum_{i=1}^{N} a_i$$

distributed across p processes Kahan-Babuška-Neumaier compensation algorithm. The results with N=1,000,000

Number of processes	Summation
1	500144.93
2	500144.93
4	500144.93
8	500144.93
16	500144.96
64	500144.93

MPI reduction for sum

Example 7: Calculate the summation of an N element array a

$$\sum_{i=1}^{N} a_i$$

distributed across p processes Kahan-Babuška-Neumaier compensation algorithm. The results with N=1,000,000

Number of processes	Summation
1	500144.93
2	500144.93
4	500144.93
8	500144.93
16	500144.96
64	500144.93

More consistent

OpenMP reduction for sum

OpenMP reduction for sum

We only show a segment of the C code that resembles the workflow of the MPI code

```
#pragma omp parallel num_threads(num_threads)
{
    float lsum = 0.0f;
    volatile float c = 0.0f, t = 0.0f;
    #pragma omp for private(t) nowait // Calculate the local sum Kahan-Babuška-Neumaier compensated sum within a thread
    for (int i = 0; i < N; i++) {
        t = lsum + a[i];
        if (fabs(lsum) >= fabs(a[i]))
            c += (lsum - t) + a[i];
        else
            c += (a[i] - t) + lsum;
            lsum = t;
    }
    lsum += c;

    #pragma omp atomic // Aggregate local sums to obtain the global sum
    sum += lsum;
}
```

OpenMP reduction for sum

Example 8: Calculate the summation of an N element array a

$$\sum_{i=1}^{N} a_i$$

using threads Kahan-Babuška-Neumaier compensation algorithm. The results with $N{=}1{,}000{,}000$ are show below. They are now always identical.

Number of threads	Summation
1	500144.93
2	500144.93
4	500144.93
6	500144.93
8	500144.96
12	500144.93

Future topics

Future topics

Follow-up work/topics

- Explorations of floating point number operation examples.
- Compensated summations in reduce operations in multithreaded codes.
- Compensated summations in reduce operations on GPUs.

References

1. 中田真秀 (NAKATA Maho), **MPLAPACK**, https://github.com/nakatamaho/mplapack.
2. David Bailey, **MPFUN2020**, https://www.davidhbailey.com/dhbsoftware/.
3. **Julia**, a language for scientific and high performance computing, https://docs.julialang.org/en/v1/.
4. Implementation of Kahan summation for MPI reductions https://github.com/jeffhammond/KahanMPI.
5. R. W. Robey, J. M. Robey, R. Aulwes, “In search of numerical consistency in parallel programming”, *Parallel Computing*, Vol 37, p. 217-229, 2011.
6. IEEE-754 Floating Point Converter, https://www.h-schmidt.net/FloatConverter/IEEE754.html.