Level 1 Parallel RTN-BLAS : Implementation and Efficiency Analysis

Chemseddine Chohra
Philippe Langlois and David Parello

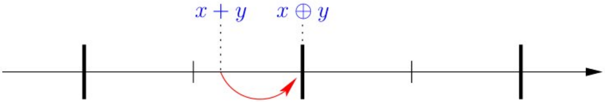
University of Perpignan Via Domitia (UPVD)

24 September 2014

Introduction and problematic

Limited machine precision

- Using floating point numbers as approximation.
- x $\longrightarrow$ X = fl(x) if x $\notin$ F or x if x $\in$ F.
- X + Y $\neq$ X $\oplus$ Y = fl(X + Y).
- IEEE-754 standard defines several rounding modes.



Introduction and problematic

Limited machine precision

- Using floating point numbers as approximation.
- $x \longrightarrow X = fl(x)$ if $x \notin F$ or x if $x \in F$.
- $X + Y \neq X \oplus Y = fl(X + Y)$.
- IEEE-754 standard defines several rounding modes.

Non-associativity of addition

- $A \oplus (B \oplus C) \neq (A \oplus B) \oplus C$.
- Catastrophic cancelation : $M = 2^{53}$; $0 = -M \oplus (M \oplus 1) \neq (-M \oplus M) \oplus 1 = 1$.

Introduction and problematic

Limited machine precision

- Using floating point numbers as approximation.
- x $\longrightarrow$ X = fl(x) if x $\notin$ F or x if x $\in$ F.
- X + Y $\neq$ X $\oplus$ Y = fl(X + Y).
- IEEE-754 standard defines several rounding modes.

Non-associativity of addition

- A $\oplus$ (B $\oplus$ C) $\neq$ (A $\oplus$ B) $\oplus$ C.
- Catastrophic cancelation : M = 2^{53}; 0 = -M $\oplus$ (M $\oplus$ 1) $\neq$ (-M $\oplus$ M) $\oplus$ 1 = 1.

Non-reproducibility of summation

- For a sum ($\sum_{i=1}^{n} X_i$), the final result depends on the order of the computations.
- In parallel programs, dynamic scheduling and reductions could change this order.
- Exascale computing.
  - Reached in 2020 : 10^{18} flop/s, Millions of cores.
  - Reproducibility of results will be a challenge.

Introduction and problematic

Why numerical reproducibility is important ?

- Problem for debugging.
  - We can not debug errors that we can not reproduce.
- Problem for validating results.
  - For contractual and legacy reasons.
- The problem arises in real applications.
  - Energetics (Villa and al., 2009).
  - Climate modeling (Y. He and al., 2001).
  - Molecular dynamics (P. Saponaro., 2010).

How to fix the numerical reproducibility problem ?

- Fix the computation order.
  - Static scheduling.
  - Deterministic reduction (Katranov, 2012).
- Deterministic error (Demmel and Nguyen, 2013).
  - ReprodSum.
  - FastReprodSum.
  - 1-Reduction.
- Enhanced precision.
  - Higher precision (quadruple precision for instance).
    - Reduce the probability of non-reproducibility (Villa and al., 2009).
    - Get more reproducible bits.
  - Correctly rounded arithmetic.
    - Deterministic Bit-Accurate Parallel Summation (S. Collange and al., 2014).

Our aim

Guarantee the numerical reproducibility for BLAS (Basic Linear Algebra Subroutines)

- Level 1 : max, min, scal, axpy, norm, asum, dot.
- dot can be transformed to a sum $\sum_{i=1}^{n} X_i \cdot Y_i = \sum_{i=1}^{2n} Z_i$.

Compute an accurate sum

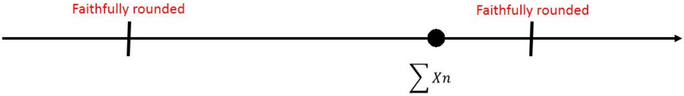
- When the result is correctly rounded, then it is reproducible.
- Several algorithms available.
- Is the cost acceptable ?

Table of contents

Recent summation algorithms

Faithfully rounded (one of the floating-point neighbors)

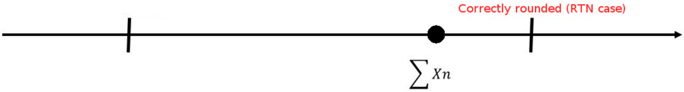
- AccSum (Rump and al., 2008).
- FastAccSum (Rump, 2008).



Recent summation algorithms

Faithfully rounded (one of the floating-point neighbors)

- AccSum (Rump and al., 2008).
- FastAccSum (Rump, 2008).



Correctly rounded (according to the rounding mode)

- NearSum (Rump and al., 2008).
- iFastSum (Zhu and Hayes, 2009).
- HybridSum (Zhu and Hayes, 2009).
- OnlineExact (Zhu and Hayes, 2010).

Experimental framework of this work

Implementation

- Implemented using C language.

Hardware

- Xeon E5 socket.
- Cache L1 = 32 KB, L2 = 256 KB, L3 = 20 MB.
- Memory max bandwidth 51,2 GB/s.
- Turbo boost turned off.

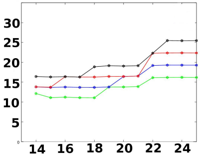
Compiler

- Intel ICC 14.0.0.
- Options : -O3 -axCORE-AVX-I -fp-model double -fp-model strict -funroll-all-loops.

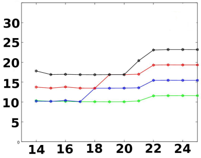
HybridSum and OnlineExact do not depend on the condition number

Implementation

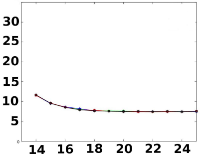
- Manually optimized version for all algorithms (see details in next section).



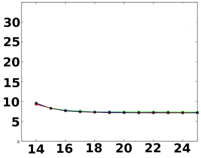
(a) AccSum



(b) FastAccSum



(c) OnlineExact



(d) HybridSum

Legends.

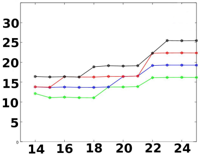
X axis → Log2 of size.
Y axis → Runtime(cycles) / size.

- Cond $= 10^{8}$
- Cond $= 10^{16}$
- Cond $= 10^{24}$
- Cond $= 10^{32}$

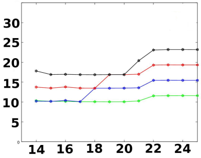
HybridSum and OnlineExact do not depend on the condition number

Implementation

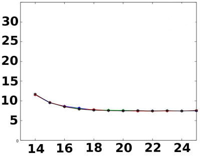
- Manually optimized version for all algorithms (see details in next section).



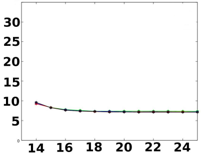
(e) AccSum



(f) FastAccSum



(g) OnlineExact



(h) HybridSum

Legends.

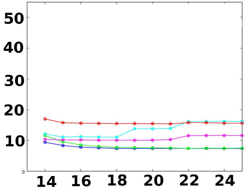
X axis → Log2 of size.
Y axis → Runtime(cycles) / size.

- Cond $= 10^{8}$
- Cond $= 10^{16}$
- Cond $= 10^{24}$
- Cond $= 10^{32}$

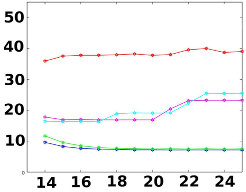
Note

- HS and OLE : condition number independents.

OnlineExact and HybridSum are faster for large vectors



(i) Condition Number = 10^8



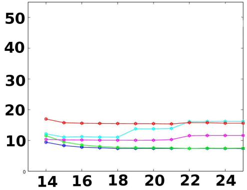
(j) Condition Number = 10^{32}

Legends.

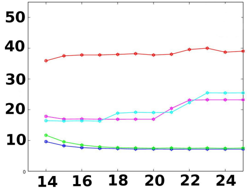
X axis → Log2 of size.
Y axis → Runtime(cycles) / size.

- iFastSum
- AccSum
- FastAccSum
- OnlineExact
- HybridSum

OnlineExact and HybridSum are faster for large vectors



(k) Condition Number = 10^8



(l) Condition Number = 10^{32}

Legends.

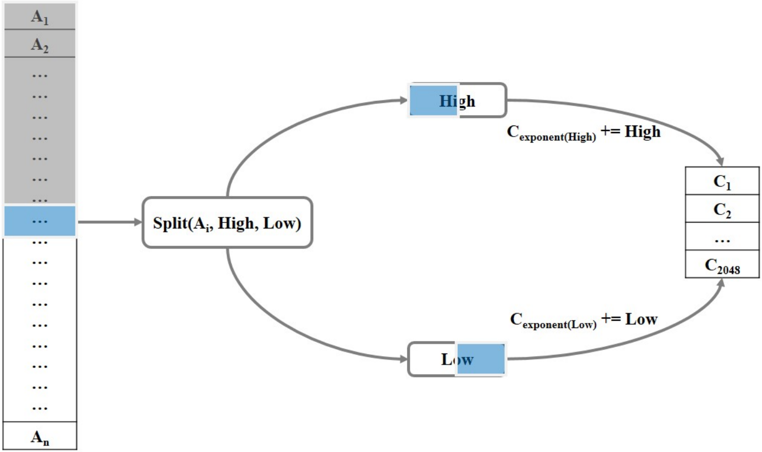
X axis → Log2 of size.
Y axis → Runtime(cycles) / size.

- iFastSum
- AccSum
- FastAccSum
- OnlineExact
- HybridSum

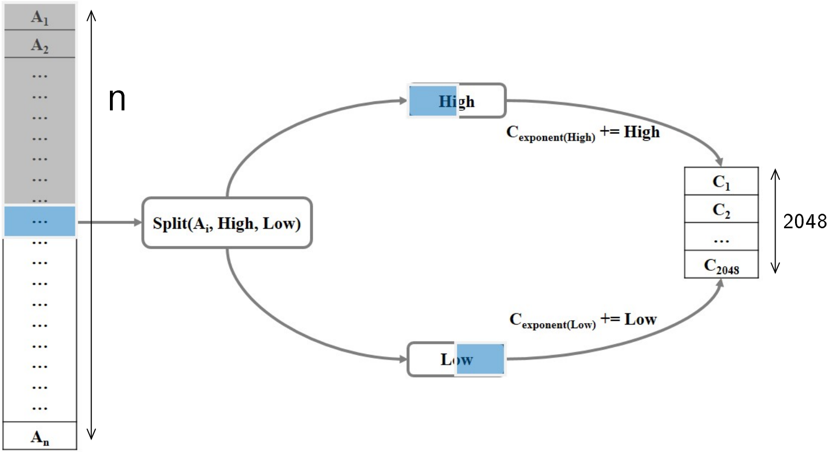
Note

- HS and OLE : linear to size.

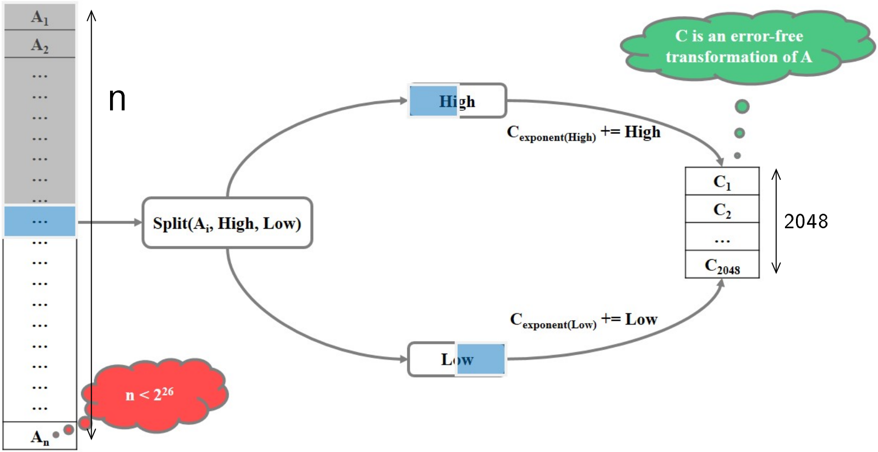
Description of algorithm HybridSum (Zhu and Hayes, 2009)



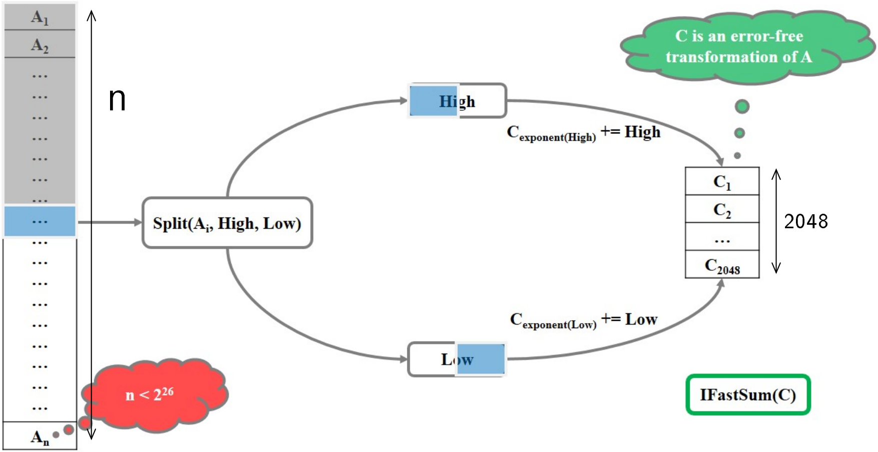
Description of algorithm HybridSum (Zhu and Hayes, 2009)



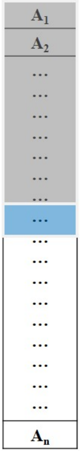
Description of algorithm HybridSum (Zhu and Hayes, 2009)



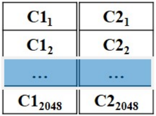
Description of algorithm HybridSum (Zhu and Hayes, 2009)



Description of algorithm OnlineExact (Zhu and Hayes, 2010)



A_1
A_2
…
…
…
…
…
…
…
…
…
…
…
…
…
…
…
…
…
A_n



$C1_1$	$C2_1$
$C1_2$	$C2_2$
…	…
$C1_{2048}$	$C2_{2048}$

Description of algorithm OnlineExact (Zhu and Hayes, 2010)

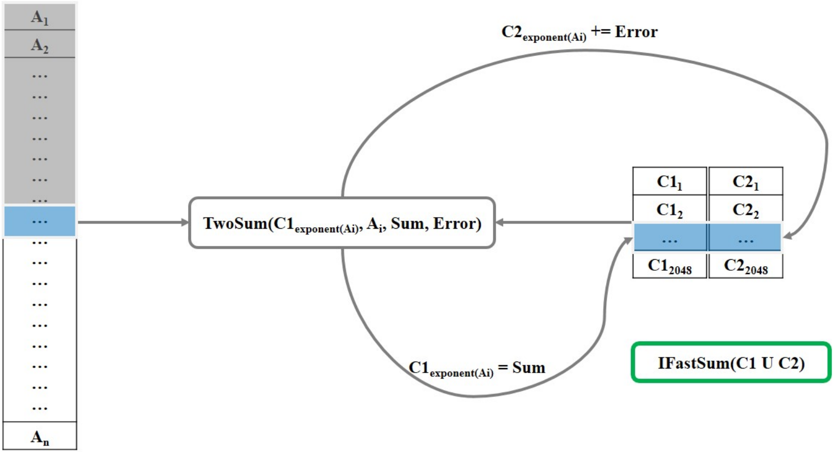


Table of contents

Optimization of HybridSum

```
ALGORITHM HybridSum.
INPUT : A, an array of floating point summands.
OUTPUT : S, the correctly rounded sum of A.
BEGIN.
  1 Declare an intermediate array C.
  2 for i=1:n do.
      1 maskSplit(A[i], $a_h$, $a_l$).
      2 e = exponent($a_h$).
      3 C[e] += $a_h$.
      4 e = exponent($a_l$).
      5 C[e] += $a_l$.
  3 end for.
  4 RETURN iFastSum(C).
END.
```

Optimization of HybridSum

```
ALGORITHM HybridSum.
INPUT : A, an array of floating point summands.
OUTPUT : S, the correctly rounded sum of A.
BEGIN.
  1 Declare an intermediate array C.
  2 for i=1:n do.
      1 veltkampSplit(A[i], $a_h$, $a_l$).  step(1)
      2 e = exponent($a_h$).
      3 C[e] += $a_h$.
      4 e = exponent($a_l$).
      5 C[e] += $a_l$.
  3 end for.
  4 RETURN iFastSum(C).
END.
```

Optimization of HybridSum

```
ALGORITHM HybridSum.
INPUT : A, an array of floating point summands.
OUTPUT : S, the correctly rounded sum of A.
BEGIN.
  1 Declare an intermediate array C.
  2 for i=1:n (unrolled) do.   step(2)
        1 veltkampSplit(A[i], $a_h$, $a_l$).   step(1)
        2 e = exponent($a_h$).
        3 C[e] += $a_h$.
        4 e = exponent($a_l$).
        5 C[e] += $a_l$.
  3 end for.
  4 RETURN iFastSum(C).
END.
```

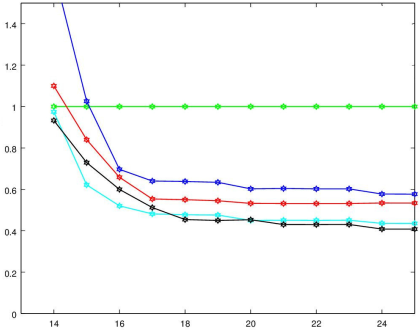
Optimization of HybridSum

```
ALGORITHM HybridSum.
INPUT : A, an array of floating point summands.
OUTPUT : S, the correctly rounded sum of A.
BEGIN.
  1 Declare an intermediate array C.
  2 for i=1:n (unrolled) do.  step(2)
      1 prefetch data.  step(3)
      2 veltkampSplit(A[i], $a_h$, $a_l$).  step(1)
      3 e = exponent($a_h$).
      4 C[e] += $a_h$.
      5 e = exponent($a_l$).
      6 C[e] += $a_l$.
  3 end for.
  4 RETURN iFastSum(C).
END.
```

Optimization of HybridSum

```
ALGORITHM HybridSum.
INPUT : A, an array of floating point summands.
OUTPUT : S, the correctly rounded sum of A.
BEGIN.
  1 Declare an intermediate array C.
  2 for i=1:n (unrolled) do.  step(2)
      1 prefetch data.  step(3)
      2 veltkampSplit(A[i], $a_h$, $a_l$).  step(1)
      3 e = exponent($a_h$).
      4 C[e] += $a_h$.
      5 e = e - 27.  step(4)
      6 C[e] += $a_l$.
  3 end for.
  4 RETURN iFastSum(C).
END.
```

Gain of 60% of runtime after optimization of HybridSum

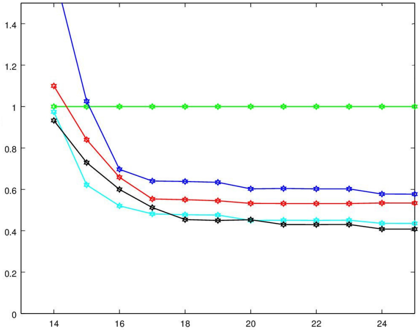


Legends.

X axis → Log2 of size.
Y axis → Runtime(cycles) / naive.

- Naive implementation
- Step 1 : replace the mask
- Step 2 : unrolling loop
- Step 3 : prefetching
- Step 4 : compute exponent

Gain of 60% of runtime after optimization of HybridSum



Legends.

X axis → Log2 of size.
Y axis → Runtime(cycles) / naive.

- Naive implementation
- Step 1 : replace the mask
- Step 2 : unrolling loop
- Step 3 : prefetching
- Step 4 : compute exponent

Note

- We gain 60% of runtime.

Optimization of OnlineExact

```
ALGORITHM OnlineExact.
INPUT : A, an array of floating point summands.
OUTPUT : S, the correctly rounded sum of A.
BEGIN.
  ① Declare two intermediate arrays C1, C2.
  ② for i=1:n do.
       ① i = exponent(a).
       ② (C1[i], a) = 2Sum(C1[i], a).
       ③ C2[i] += a.
     end for.
  ③ RETURN iFastSum(C1 ∪ C2).
END.
```

Optimization of OnlineExact

```
ALGORITHM OnlineExact.
INPUT : A, an array of floating point summands.
OUTPUT : S, the correctly rounded sum of A.
BEGIN.
  1 Declare two intermediate arrays C1, C2.
  2 for i=1:n (unrolled) do.  step(1)
      1 i = exponent(a).
      2 (C1[i], a) = 2Sum(C1[i], a).
      3 C2[i] += a.
    end for.
  3 RETURN iFastSum(C1 $\cup$ C2).
END.
```

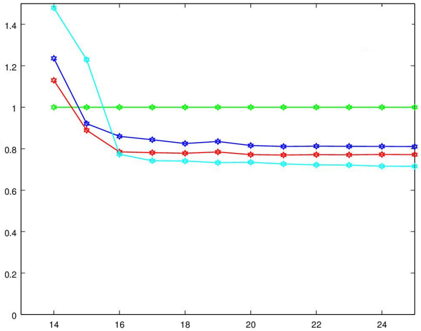
Optimization of OnlineExact

```
ALGORITHM OnlineExact.
INPUT : A, an array of floating point summands.
OUTPUT : S, the correctly rounded sum of A.
BEGIN.
  1 Declare two intermediate arrays C1, C2.
  2 for i=1:n (unrolled) do.  step(1)
      1 prefetch data.  step(2)
      2 i = exponent(a).
      3 (C1[i], a) = 2Sum(C1[i], a).
      4 C2[i] += a.
    end for.
  3 RETURN iFastSum(C1 ∪ C2).
END.
```

Optimization of OnlineExact

```
ALGORITHM OnlineExact.
INPUT : A, an array of floating point summands.
OUTPUT : S, the correctly rounded sum of A.
BEGIN.
  1 Declare an intermediate arrays C. step(3)
  2 for i=1:n (unrolled) do.  step(1)
      1 prefetch data.  step(2)
      2 i = exponent(a).
      3 (C[2*i], a) = 2Sum(C[2*i], a).  step(3)
      4 C[2*i+1] += a.  step(3)
    end for.
  3 RETURN iFastSum(C). step(3)
END.
```

Gain of 25% of runtime after optimization of OnlineExact

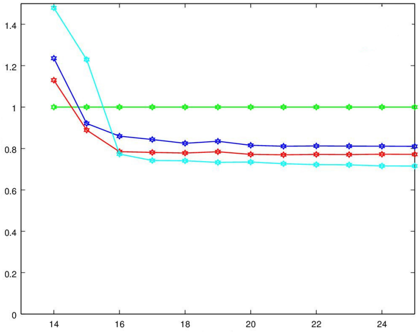


Legends.

X axis → Log2 of size.
Y axis → Runtime(cycles) / naive.

- Naive implementation
- Step 1 : unrolling loop
- Step 2 : prefetching
- Step 3 : use one vector

Gain of 25% of runtime after optimization of OnlineExact



Legends.

X axis → Log2 of size.
Y axis → Runtime(cycles) / naive.

- Naive implementation
- Step 1 : unrolling loop
- Step 2 : prefetching
- Step 3 : use one vector

Note

- We gain 25% of runtime.

Compare to dasum, ReprodSum and FastReprodSum

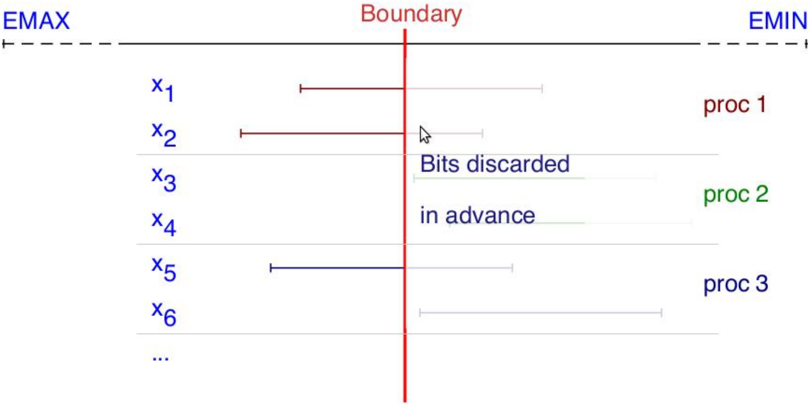
Optimized sum

- dasum : optimized by Intel in the library MKL.

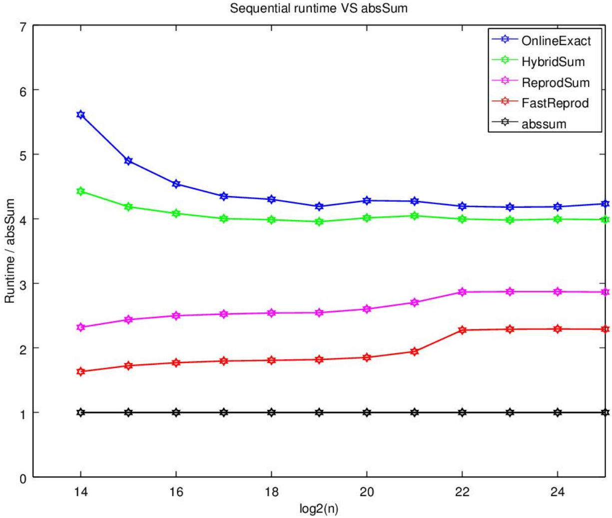
Reproducible sum

- ReprodSum : guarantee reproducibility.
- FastReprodSum : faster than ReprodSum but requires direct rounding.

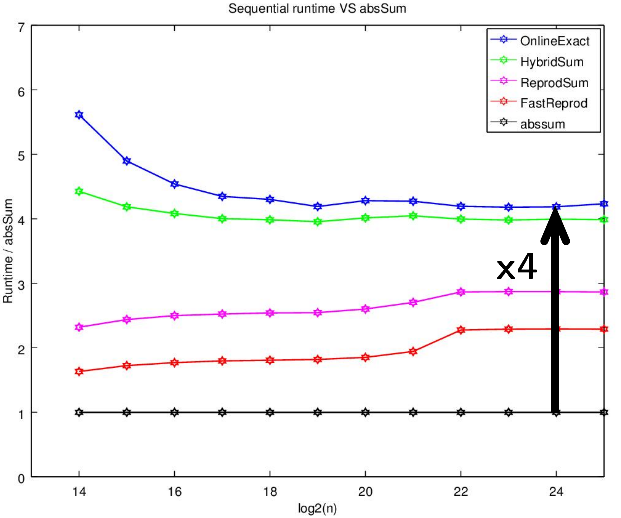
ReprodSum and FastReprodSum (Demmel and Nguyen, 2013)



Overhead in the sequential case



Overhead in the sequential case



Overhead in the sequential case

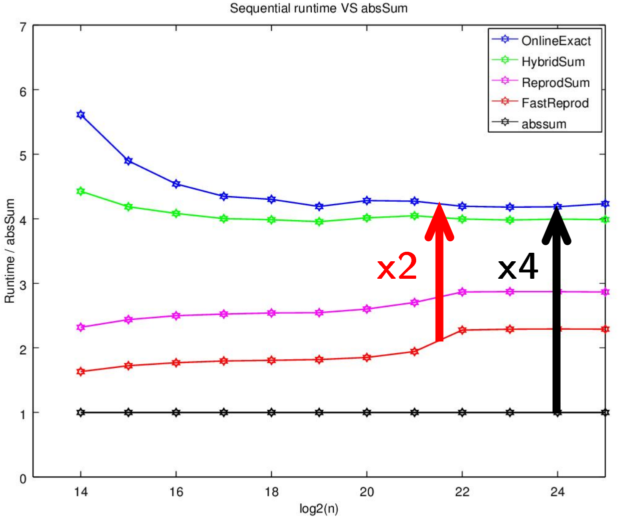
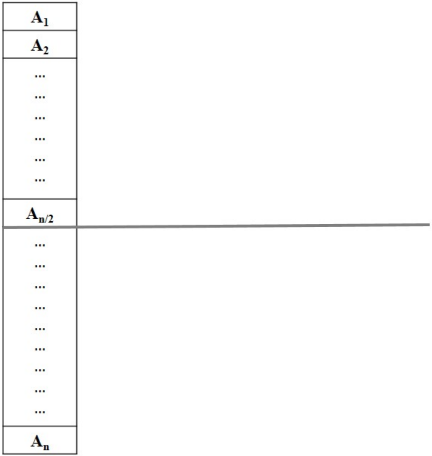
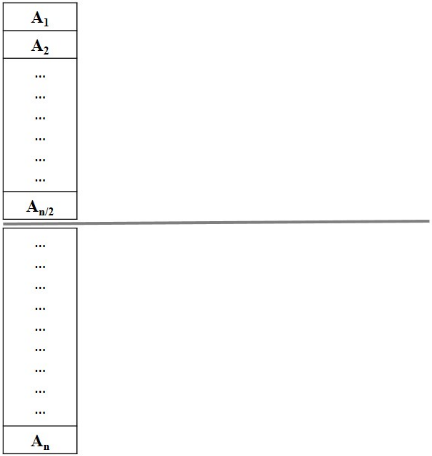


Table of contents

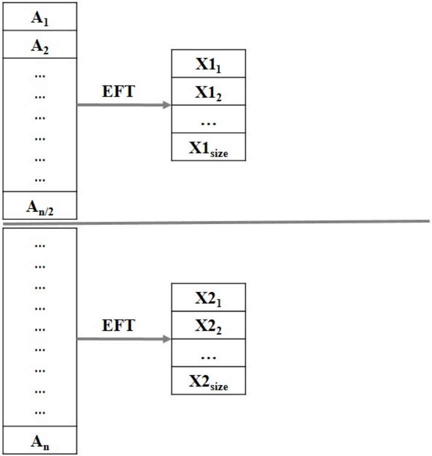
Parallel algorithm (2 processors case)



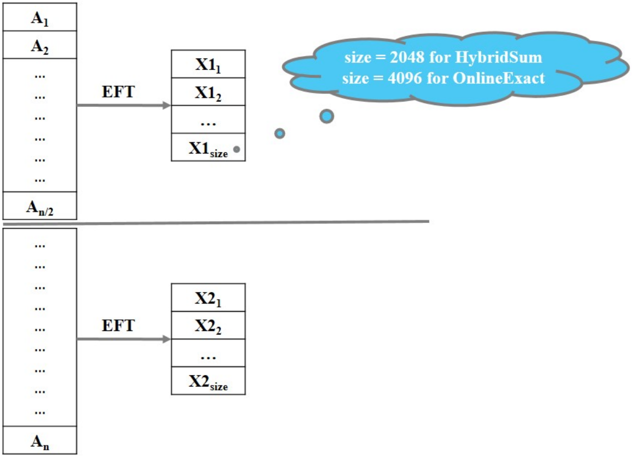
Parallel algorithm (2 processors case)



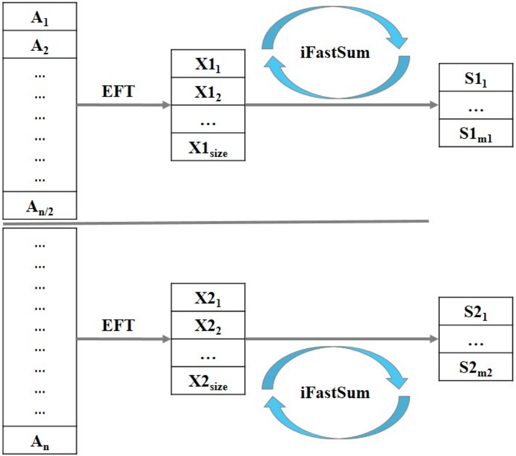
Parallel algorithm (2 processors case)



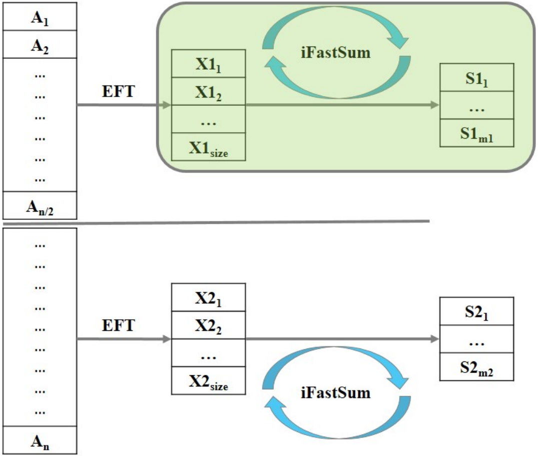
Parallel algorithm (2 processors case)



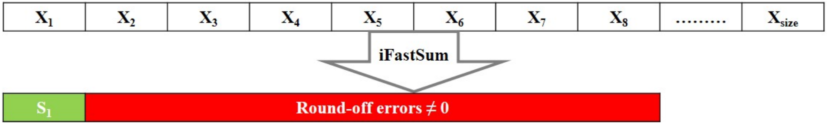
Parallel algorithm (2 processors case)



Parallel algorithm (2 processors case)



Parallel algorithm (2 processors case)

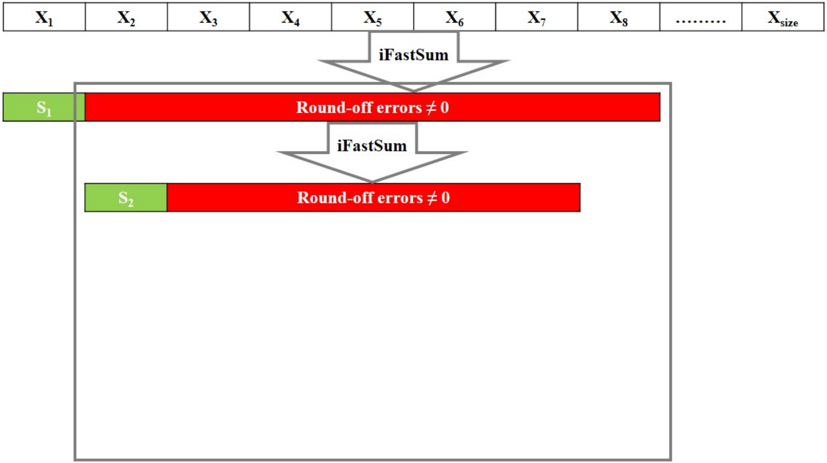


X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	………	X_{size}

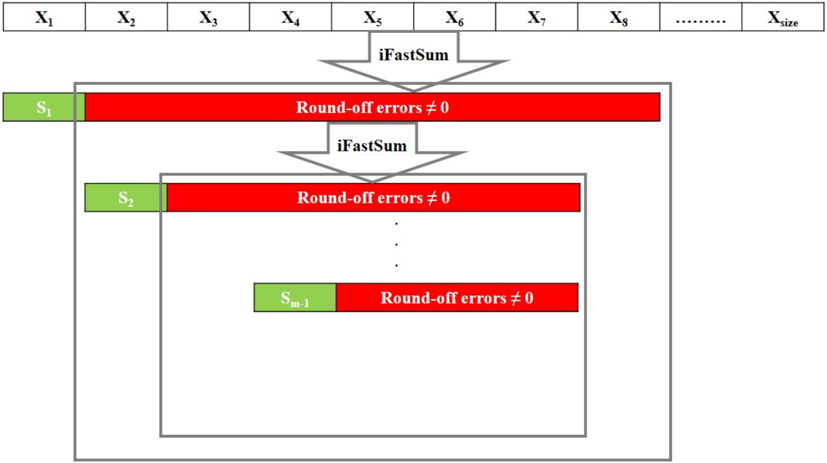
iFastSum

S_1 Round-off errors $\neq 0$

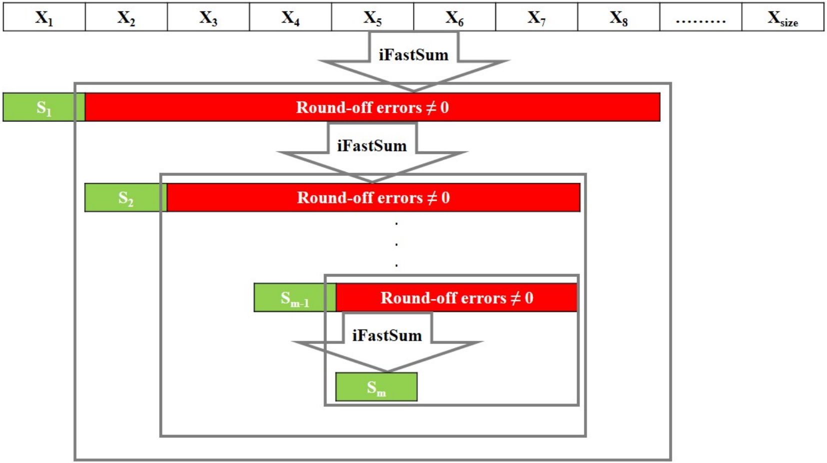
Parallel algorithm (2 processors case)



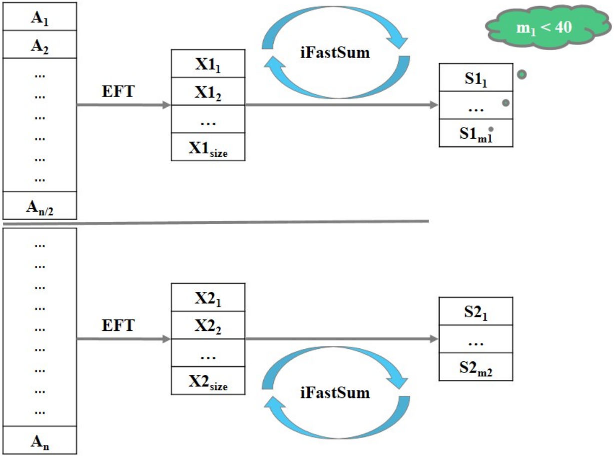
Parallel algorithm (2 processors case)



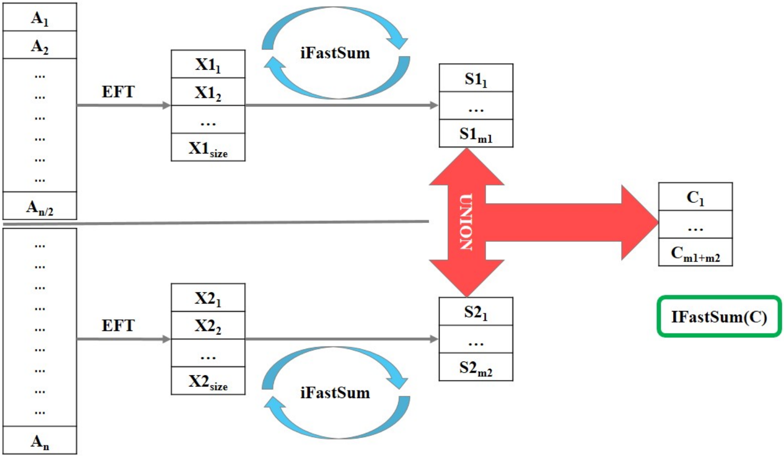
Parallel algorithm (2 processors case)



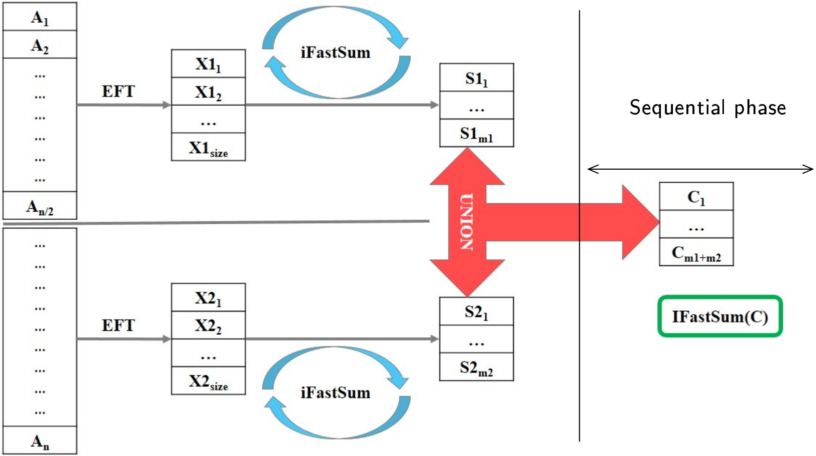
Parallel algorithm (2 processors case)



Parallel algorithm (2 processors case)



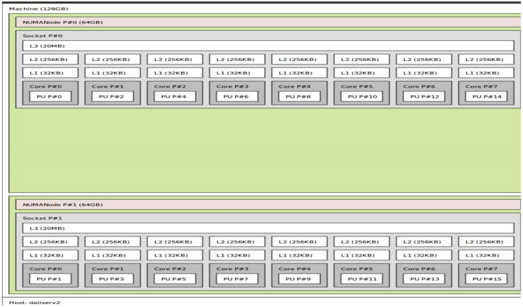
Parallel algorithm (2 processors case)



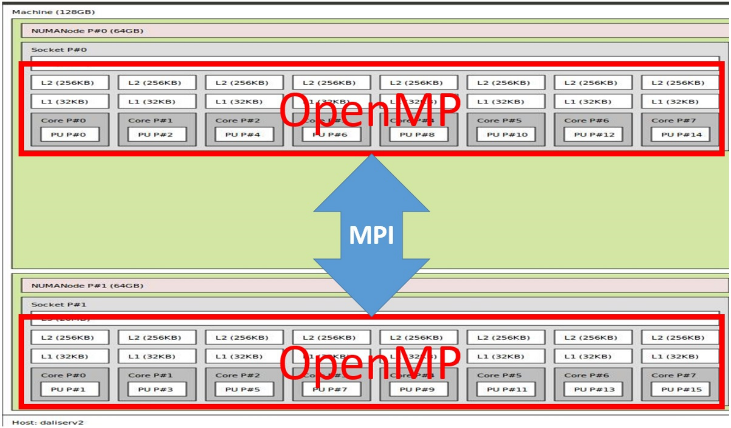
Experimental framework

Hardware

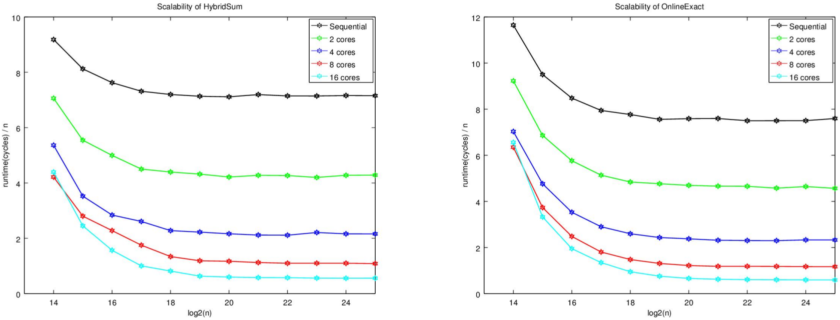
- Two Xeon E5 sockets.
- 8 cores on each socket.
- Multi-threading is turned off.



Implementation details



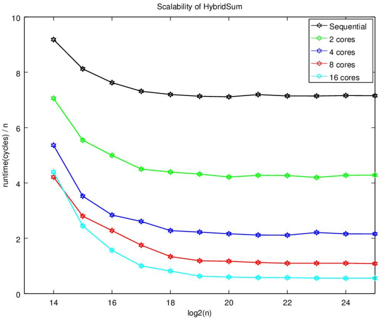
Strong scaling of HybridSum and OnlineExact



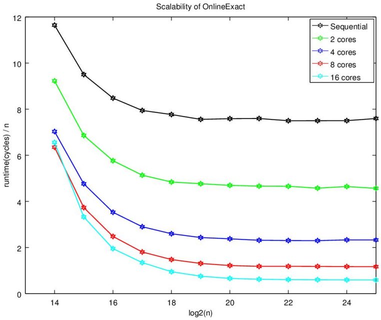
(m) HybridSum

(n) OnlineExact

Strong scaling of HybridSum and OnlineExact



(o) HybridSum

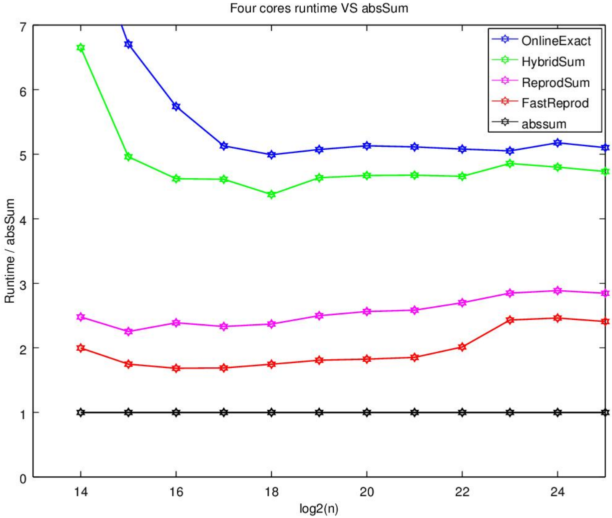


(p) OnlineExact

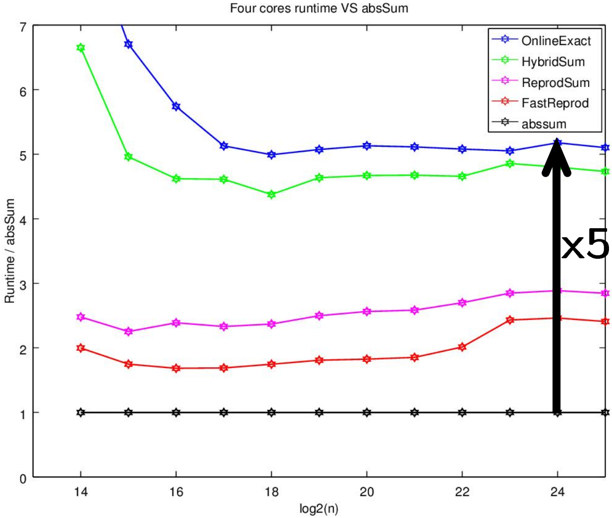
Note

- Good scalability up to 16 cores at least.

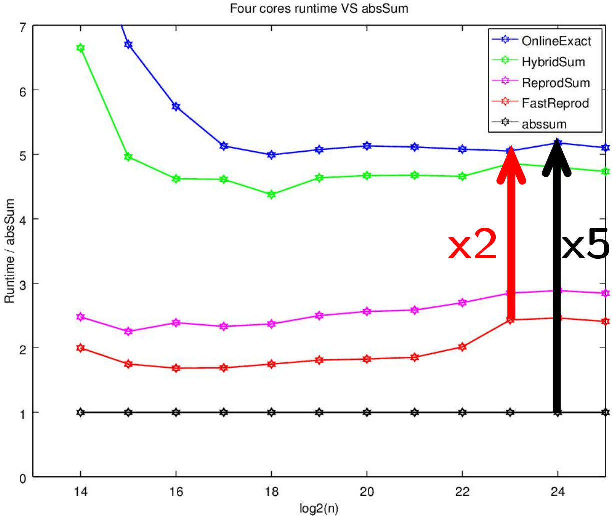
4 cores parallel results



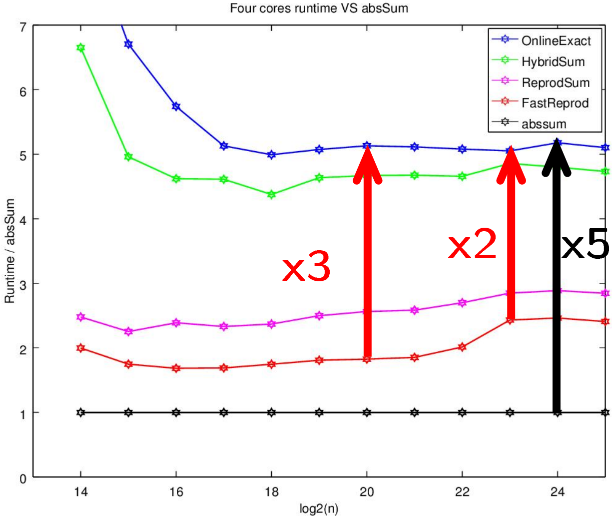
4 cores parallel results



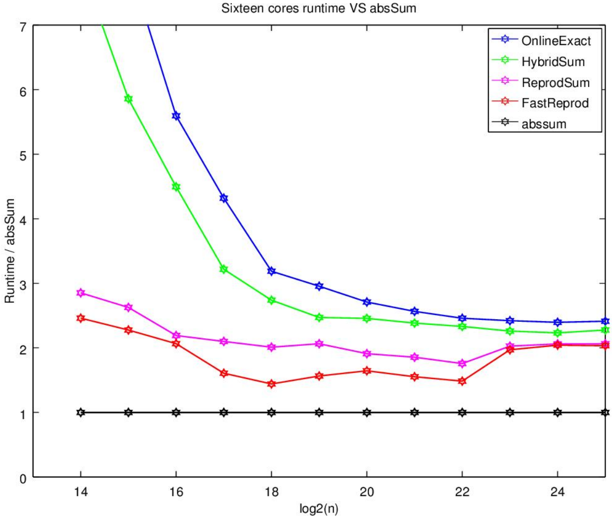
4 cores parallel results



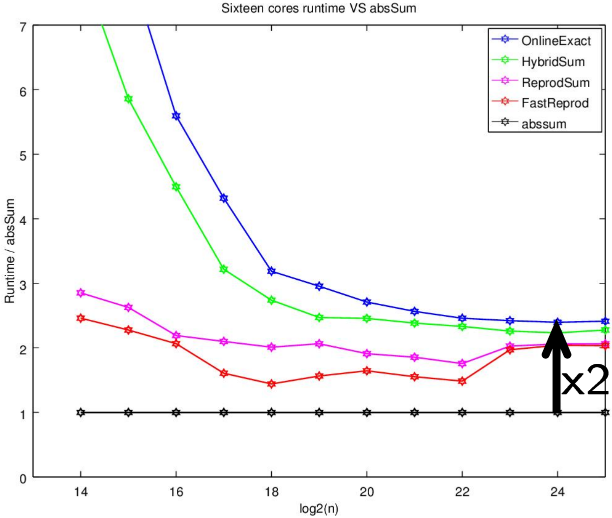
4 cores parallel results



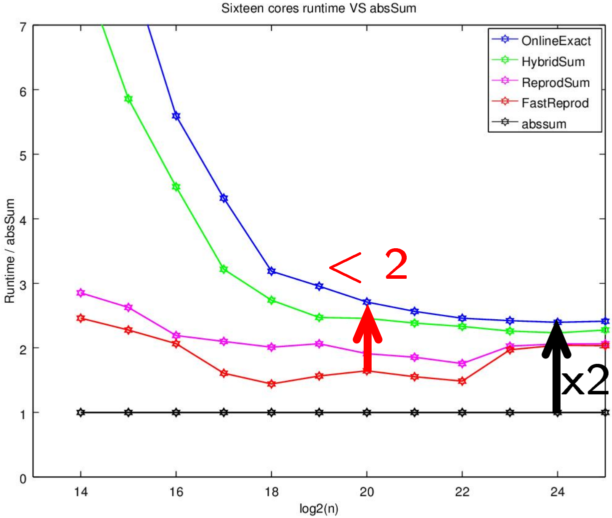
16 cores parallel results



16 cores parallel results

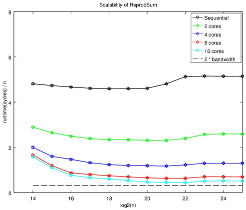


16 cores parallel results

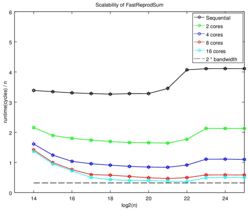


Limit of bandwidth for dasum, ReprodSum and FastReprodSum

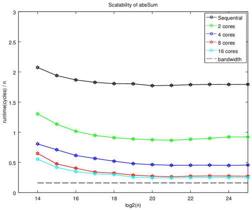
Strong scalability using 1 core, 2 cores, 4 cores, 8 cores and 16 cores.



(q) ReprodSum



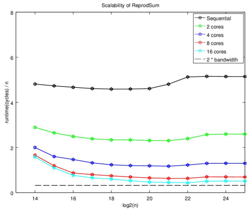
(r) FastReprodSum



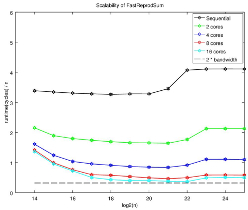
(s) dasum

Limit of bandwidth for dasum, ReprodSum and FastReprodSum

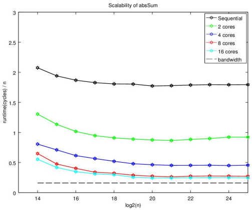
Strong scalability using 1 core, 2 cores, 4 cores, 8 cores and 16 cores.



(t) ReprodSum



(u) FastReprodSum

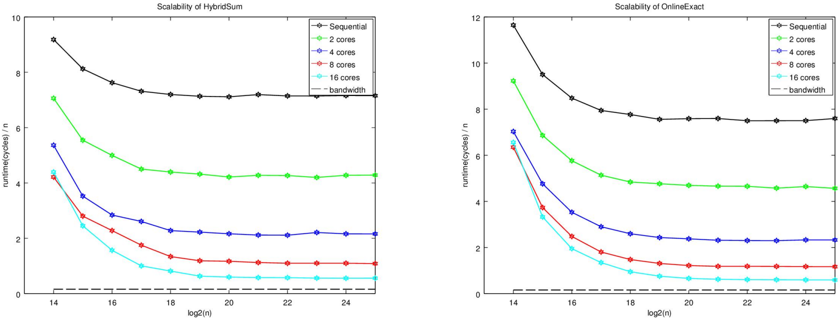


(v) dasum

Note

- dasum, ReprodSum and FastReprodSum : bandwidth limit.

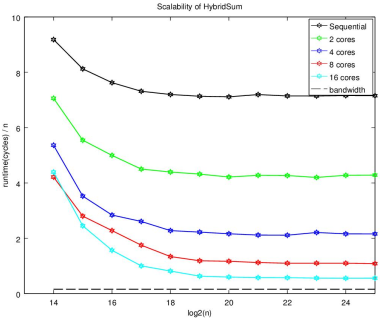
HybridSum and OnlineExact are not limited by bandwidth



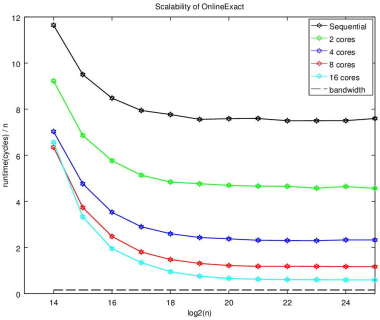
(w) HybridSum

(x) OnlineExact

HybridSum and OnlineExact are not limited by bandwidth



(y) HybridSum

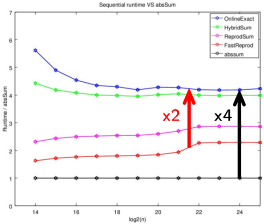


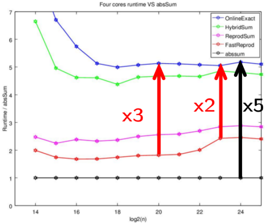
(z) OnlineExact

Note

- HybridSum and OnlineExact : no bandwidth limit.

Summarization





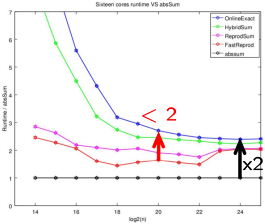


Table of contents

Conclusion

Our paradigm

- If we are accurate enough, then we are reproducible.
- How much does it cost ?

The used algorithms are convincingly

- Not depending on condition number.
- Only one pass through the input vector.
- No reuse of data, so not depending on cache.
- Suitable to shared memory and NUMA architectures.
- Scale correctly until 16 cores using hybrid parallel programming.

The use could be restricted

- RTN sum have up to 5 times overhead.
- Use on applications with no strict temporary limits.
- Use for debugging and validating steps.

Future work

Future work

- Test on a large-scale system.
- Compare to 1-Reduction algorithm (adapted for large-scale systems).
- Upgrade to BLAS level 2 and 3.

THANK YOU

FOR

YOUR ATTENTION