Effects of Floating-Point non-Associativity on Numerical Computations on Massively Multithreaded Systems

Daniel Chavarría, Oreste Villa, Andrés Márquez, Vidhya Gurumoorthi

Pacific Northwest National Laboratory
Non-determinism in floating point reduction

- Relevant Background
- Effect of floating point reductions on Applications
  - Power State Estimation
- Floating-point Reductions (used in dot products)
  - Performance evaluation
  - Accuracy and Precision (non-determinism)
- Evaluated Strategies
  - Quad-precision
  - Deterministic Tree
- Conclusions
Non-associativity of IEEE floating point operation:

$$(a + b) + c \neq a + (b + c)$$

Example: $S = 0.001 \times 10^{32} - 10^{32}$

Considering an architecture with 32 “digits” of precisions, depending on the relative order of the additions the result could be $S = 0$ or it could $S = 0.001$ (Truncation and Rounding errors)

This behavior in general introduces **accuracy** errors (they are indeed present in each serial code, or message passing based code)

However this behavior introduces **non-determinism** in shared memory machines where for example many threads may interleave in different ways updating a shared variables.
Issues with precision of floating point reduction

- Reductions of floating-point vectors can produce non-deterministic results with the same inputs and same processor count.
- Intermediate results can vary significantly depending on thread scheduling and accumulation implementations strategies.
- Problem is more evident for larger accumulation, where accumulated values differs of many orders of magnitude.
- Iterative algorithms carrying precision errors over multiple iterations could generate “unpredictable” convergence problems.

```c
double sum = 0.0;
for (i = 0; i < n; i++)
    sum += A[i];
```
Power System State Estimation

Situational Awareness?

Do we know what really happened? Could it be prevented?

Compute a **reliable** estimate of the system state (voltages)

Source: NOAA/DMSP
Power System State Estimation (PSE)

- **Given:** power grid topological information, telemetry on line flows, bus injections or bus voltages
- **Compute:** a reliable estimate of the system state (bus voltages), validate model structure and parameter values
- Calculated using Weighted Least-Squares (WLS) method
- **WLS:** minimize
  \[ J = \sum_{i=1}^{m} w_i r_i^2 = r^T W r \]

  - Where \( r = z - h(x) \) (\( r \) is the residual vector) \( x \) is the system state, \( z \) is a vector of measured quantities, \( h \) is a vector function, \( w_i \) is the weight for residual \( r_i \) and \( W \) is as diagonal matrix.
- This is a non-linear problem, which is solved using the Newton-Raphson iterative procedure
Power System State Estimation (cont.)

- Every iteration requires solving a large set of sparse linear equations.
- Sparse matrices are derived from the topology of the power grid being analyzed.
- The set of linear equations is solved with Conjugate Gradient (CG).

PSE is a critical element of the software used by power grid control centers.
- Has to operate under real-time constraints.
- Has to produce reliable results.

```
Compute Norm_WLS
While (Norm_WLS < ξ1)
    Linearization
    Compute Norm_CG
    While ( Norm_CG < ξ2 )
        SparseMatVecProd
    ... 
    Compute Norm_CG
End Loop CG
Compute Norm_WLS
End Loop WLS
```
Ported Fortran-based sequential WLS PSE

- Uses a CG solver at its core (which scales better than direct solvers based on LU or Cholesky factorization)

95% of the computation time is spent in the Newton-Raphson WLS iterative solver

- Most of it inside the CG solver for the linearized formulation computing a sparse matrix-vector product
  - Rest of the CG steps are vector-vector operations (addition/subtraction and dot products)

The XMT compiler was able to automatically parallelize the vector-vector operations (based on their dependence patterns)

We added some directives to guide the parallelization of the sparse matrix-vector product
Floating-point Reductions

- Initial tests of our XMT implementation of PSE on the 14,000 nodes WECC* model produced non-deterministic results between runs on the same number of processors (!!!)
  - J index and several node estimation fluctuating on the last digits!!
  - Immediately, we suspected a race condition in our code
- The culprits were **dot products** in the CG solver which were producing non-deterministic results

\[ d = \sum_{i=1}^{n} v_i w_i \]

- The fine-grained parallel execution on the ThreadStorm processors combined with the compiler based reduction code was causing the non-associative nature of double-precision IEEE floating-point addition to produce different results (depending on the particular thread interleaving)

*WECC = Western Electricity Coordinating Council
Floating-point Reductions in PSE

- We tightened our PSE code to use statically scheduled parallel loops: `#pragma mta block schedule`
  - This guarantees deterministic assignment of iterations to threads
    - for (i = 0; i < 100; i++)…, in this example if the loop is executed on 10 threads each thread should get 10 contiguous iterations: thread 0 gets iterations 0 to 9, thread 1 gets iterations 10 to 19, etc.
  - We performed this modification on each accumulation or reduction loop in the form:
    ```
    for (i = 0; i < 100; i++)
        S += ...
    ```
  - In particular, we focused on the computation of the Euclidean norm, used for testing for convergence on the CG loop:
    - Single scalar value “highly observable”
    - Recorded before and after every iteration of the external WLS loop
  \[ \|v\| = \sqrt{\sum_{i=1}^{n}(v_i)^2} \]
Floating-point Reductions in PSE (cont.)

Variability for the norm is significant
- For 64-bit double precision

Example (norm on entry to CG routine) PSE/WECC:

<table>
<thead>
<tr>
<th>WLS Iteration</th>
<th>Run #1</th>
<th>Run #2</th>
<th>Diff. vs. Run #1</th>
<th>Run #3</th>
<th>Diff. vs. Run #1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.64E+09</td>
<td>1.64E+09</td>
<td>0.00%</td>
<td>1.64E+09</td>
<td>0.00%</td>
</tr>
<tr>
<td>2</td>
<td>1.88E+09</td>
<td>1.88E+09</td>
<td>0.00%</td>
<td>1.88E+09</td>
<td>0.00%</td>
</tr>
<tr>
<td>3</td>
<td>3.29E+07</td>
<td>3.29E+07</td>
<td>0.00%</td>
<td>3.29E+07</td>
<td>0.00%</td>
</tr>
<tr>
<td>4</td>
<td>4.01E+05</td>
<td>4.01E+05</td>
<td>0.02%</td>
<td>4.01E+05</td>
<td>0.01%</td>
</tr>
<tr>
<td>5</td>
<td>1.50E+02</td>
<td>1.29E+02</td>
<td>14.25%</td>
<td>1.24E+02</td>
<td>17.63%</td>
</tr>
<tr>
<td>6</td>
<td>5.92E+00</td>
<td>5.13E+00</td>
<td>13.30%</td>
<td>7.37E+00</td>
<td>24.64%</td>
</tr>
<tr>
<td>7</td>
<td>5.22E-01</td>
<td>4.46E-01</td>
<td>14.52%</td>
<td>4.59E-01</td>
<td>12.06%</td>
</tr>
</tbody>
</table>

Multiple runs with same input and same number of processors produce different norms.
Floating-point Reductions in the Compiler

Given the code:
```
#pragma mta block schedule
for (i = 0; i<n; i++)
    snm += r[i]
```

The programmer expects:

- Thread 0 reduces sequentially
- Thread 1 reduces sequentially
- Thread 2 reduces sequentially
- Thread 3 reduces sequentially
- ...

What really happens behind the scenes:
- Final accumulation is performed likely using concurrent atomic updates to single scalar
- Dump of the code shows `readff` and `reduce_f8_add`
Floating-point Reductions in the Compiler

Why does the variability occur?

- Compiler is in charge of generating code for the computation of the reduction
- Even with the static block scheduling, there is some degree of dynamic reordering occurring due to implementation decisions
  - For performance reasons
- For many applications, variability will be OK (within tolerance)
- For other applications, variability could be problematic

We remind that in PSE, variability:

- Leads to different overall results in “observable” significant digits
- Could increase the number of iterations used in the CG or WLS loops depending on the norm (timing constraint)
What about accuracy? Which of the three PSE runs is more “correct”? 

- The literature indicates that a full sequential reduction of a long vector can be very bad for accuracy
  - Except if the data is fully sorted in ascending order (this is the most accurate case)
- For vectors of random, uniformly distributed numbers using some form of partial sums (i.e. through threading) increases accuracy

Partial sums tend to accumulate towards comparable values, reducing the number of cancellation errors

Larger numbers of threads should increase accuracy, but not necessarily determinism

*Uniform Distribution of 10K Elements 0÷10e18
Explored approaches

**Quad-precision accumulators**
- Problem in PSE is the cancellation of the contribution of relatively small values to the accumulation
- Increase dynamic range by using `long double` accumulators (128-bit floating-point)
- The small values should still contribute to the total sum due to more significant digits in the accumulator
- Quad-precision is expensive: software emulation via combination of two double-precision variables
- However, it’s more precise and also more accurate for reductions

**Deterministic tree-based algorithm**
- Uses the concept of partial sums by thread
- But, combines the partial sums in a deterministic manner using a reduction tree
- Similar to existing reduction algorithms for distributed memory (MPI)
- Not as costly as quad-precision
- Completely precise, but potentially less accurate than quad-precision
Quad-precision Accumulators

- Quad-precision accumulators
  - Problem in PSE is the cancellation of the contribution of relatively small values to the accumulation
  - Multiple runs produce always the same result reorders of the arrays do not change results either

<table>
<thead>
<tr>
<th>WLS Iteration</th>
<th>Quad Run</th>
<th>Double Run #1</th>
<th>Diff. vs. Quad</th>
<th>Double Run #2</th>
<th>Diff. vs. Quad</th>
<th>Double Run #3</th>
<th>Diff. vs. Quad</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.64E+09</td>
<td>1.64E+09</td>
<td>0.00%</td>
<td>1.64E+09</td>
<td>0.00%</td>
<td>1.64E+09</td>
<td>0.00%</td>
</tr>
<tr>
<td>2</td>
<td>1.88E+09</td>
<td>1.88E+09</td>
<td>0.00%</td>
<td>1.88E+09</td>
<td>0.00%</td>
<td>1.88E+09</td>
<td>0.00%</td>
</tr>
<tr>
<td>3</td>
<td>3.29E+07</td>
<td>3.29E+07</td>
<td>0.00%</td>
<td>3.29E+07</td>
<td>0.00%</td>
<td>3.29E+07</td>
<td>0.00%</td>
</tr>
<tr>
<td>4</td>
<td>4.01E+05</td>
<td>4.01E+05</td>
<td>0.01%</td>
<td>4.01E+05</td>
<td>0.01%</td>
<td>4.01E+05</td>
<td>0.02%</td>
</tr>
<tr>
<td>5</td>
<td>1.43E+02</td>
<td>1.50E+02</td>
<td>5.26%</td>
<td>1.29E+02</td>
<td>9.73%</td>
<td>1.24E+02</td>
<td>13.30%</td>
</tr>
<tr>
<td>6</td>
<td>6.14E+00</td>
<td>5.92E+00</td>
<td>3.66%</td>
<td>5.13E+00</td>
<td>16.47%</td>
<td>7.37E+00</td>
<td>20.09%</td>
</tr>
<tr>
<td>7</td>
<td>5.73E-01</td>
<td>5.22E-01</td>
<td>8.77%</td>
<td>4.46E-01</td>
<td>22.02%</td>
<td>4.99E-01</td>
<td>12.77%</td>
</tr>
</tbody>
</table>

- Accuracy in norm: up to 22% difference between Quad and Double-precision
- Error propagates as the number of iterations increases
Deterministic Tree-based Algorithm

Accumulates in levels performing partial sums of size “Degree”

Degree = 2

Tree-based Algorithm (properties)

- “Left-leaning” tree: threads with lower rank do more work
- Clearly, the algorithm is not load-balanced but for a given array size there is a tradeoff between accuracy and performance (next slides)
- It is possible to use small degree for the first levels, large degree after the second
- Precision is absolute (in the sense of determinism)
- Accuracy varies with degree NOT with processor/thread count
- It allows “right-leaning” correction tree can be used to increase accuracy

Varying the Degree

- Reductions for 28K double-precision elements uniformly randomly distributed using the deterministic tree
  - Changing the degree changes the accuracy
  - For an arbitrary degree precision is absolute

Experiment performed with 1 processor (same result with more processors/threads)
Performance Comparison

- Single reduction of 28K double-precision elements uniformly randomly distributed
Tree accuracy in the overall PSE application

We executed PSE using the deterministic tree-based reductions with degree 460

- Accuracy is comparable to double precision runs
- Precision is absolute **also for the final result**

<table>
<thead>
<tr>
<th>WLS Iteration</th>
<th>Quad Run</th>
<th>Tree Run</th>
<th>Diff. vs. Quad</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.64E+09</td>
<td>1.64E+09</td>
<td>0.00%</td>
</tr>
<tr>
<td>2</td>
<td>1.88E+09</td>
<td>1.88E+09</td>
<td>0.00%</td>
</tr>
<tr>
<td>3</td>
<td>3.29E+07</td>
<td>3.29E+07</td>
<td>0.00%</td>
</tr>
<tr>
<td>4</td>
<td>4.01E+05</td>
<td>4.01E+05</td>
<td>0.00%</td>
</tr>
<tr>
<td>5</td>
<td>1.43E+02</td>
<td>1.72E+02</td>
<td>20.51%</td>
</tr>
<tr>
<td>6</td>
<td>6.14E+00</td>
<td>5.79E+00</td>
<td>5.76%</td>
</tr>
<tr>
<td>7</td>
<td>5.73E-01</td>
<td>4.28E-01</td>
<td>25.25%</td>
</tr>
</tbody>
</table>
Comparison of the different approaches

**Micro-benchmark**: Single reduction of 28K taken from PSE data on 16 processors (requesting 100 streams per processors)
- Tree with degree 460 (2 levels)
- Double precision run 100 times (taken the maximum errors)

<table>
<thead>
<tr>
<th></th>
<th>16 processors</th>
<th>Quad Prec.</th>
<th>Double Prec.</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>1.190ms</td>
<td>0.519ms</td>
<td>0.634ms</td>
<td></td>
</tr>
<tr>
<td>Accuracy</td>
<td>“perfect”</td>
<td>&lt; 52,946</td>
<td>1,688</td>
<td></td>
</tr>
<tr>
<td>Precision</td>
<td>“Absolute”</td>
<td>&lt;151,844</td>
<td>Absolute</td>
<td></td>
</tr>
</tbody>
</table>

Sum = 2.69E18
Need for Compiler Integration

Integration as a library is not quite feasible:

- Having to call a function with internal parallel loops has some overhead
- In many cases, a reduction will be performed in a parallel loop that also has other operations in the loop body
  - Moving the reduction out of the loop implies a loop distribution transformation which could reduce performance and inhibit powerful transformations such as software pipelining

It will be significantly better to integrate the precise reduction algorithm as a compiler transformation

- Potentially, we could use a new pragma (i.e. `#pragma mta precise reduction`)
- The new pragma could indicate where the programmer intends to have reductions executed with absolute precision
- Run time and/compiler can choose the right degree based on some heuristic