

Bitwise reproducible sum

Project D-HPC

Xavier Lapillonne and Boris Peltekov

17.12.2013

- Motivations and algorithms
- Reduce implementation
- Results

Motivations

- The problem:
 - Floating point addition is not associative :
On a parallel architecture, different decomposition may lead to different results
- For scientific applications bitwise reproducibility could be needed:
 - to redo runs on different systems, e.g. to have new diagnostics, more outputs
 - for validation purpose
 - for debugging

Double Sweep Sum (DS)

- [Demmel, SCA, 2013]
 - > Double sweep sum

Idea : apply some pre-rounding before summing.

- We want $S = \sum v_i$
- Introduce extractor $M(\max(v_i))$, $q_i = (v_i + M) - M$.
 - > Partial contribution $T^1 = \sum q_i$
- Improve accuracy by using the remainder $r_i = v_i - q_i$
Introduce new extractor M^2
 - > Partial contribution T^2
- ...
- Sum contributions of same level from all process (no need for a deterministic reduce)
 $\text{Reduce}([T_1, T_2, \dots], \text{SUM})$
- $t = T_1 + T_2 + \dots$

AllReduce

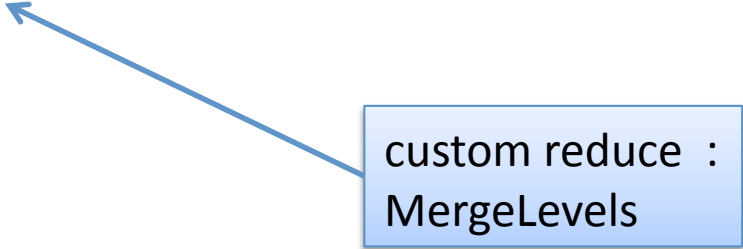
Reduce

Single Sweep Sum (SS)

- [Arteaga et al. IPDPS, 2014]
-> Single sweep sum

Idea: Use only one communication

- compute dynamically on each process an appropriate extractor (no global max)
- Each process computes partial contribution for different levels T^0, T^1, \dots
- Use a custom reduce operation to sum partial contributions which correspond to the same level
 $\text{Reduce}([T_1, T_2 \dots], \text{MergeLevels})$
- $t = T_1 + T_2 + \dots$

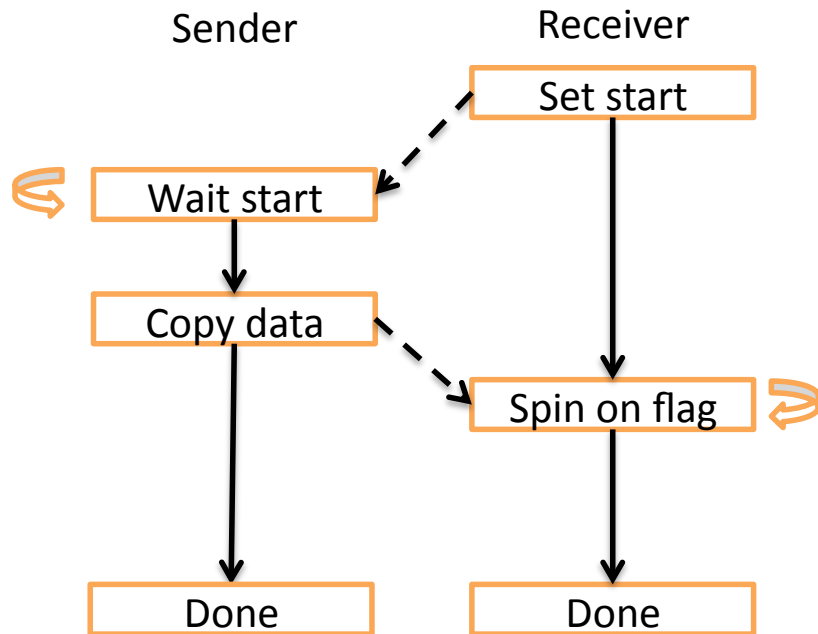


custom reduce :
MergeLevels

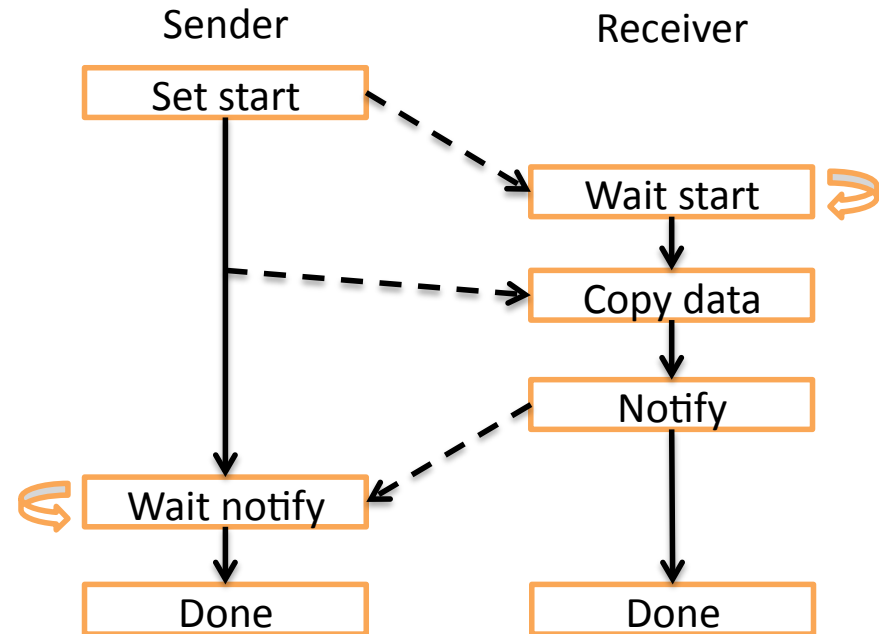
- Motivation
- **Reduce implementation**
- Results

Simple data exchange approaches

Sender driven



Receiver driven



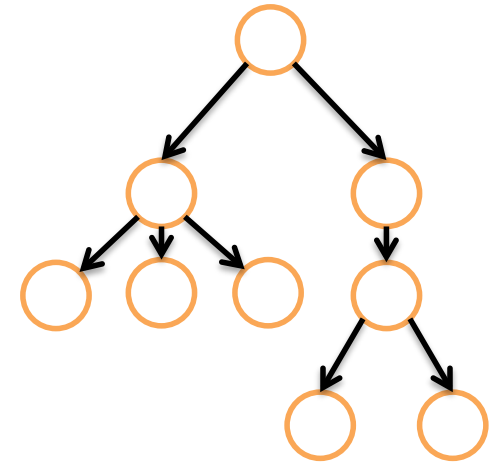
- Requires 2 cache lines
 - Start flag
 - Data + ready flag
- Good for 1 to 1 communication
- Used in Reduce and Dissemination

- Requires 2 cache lines
 - Start flag + data
 - Notify flag
- Good for 1 to N communication
- Benefits from reading cache-line in Shared state
- Used in Broadcast

Reduce implementations: SingleNotify & MultiNotify

Similarities:

- They work on arbitrary trees (given on input)
- Every thread is bound to a single node
- Supported operations:
 - Sum
 - Max
 - MergeLevels (needed for the Single Sweep algorithm)
- Similar storage space requirements: (#children+ 1 or 2) cache lines
- Both use sender-driven communication between each node and its children
- No need for synchronization barriers or memory fences (proved for x86 only)



Differences:

- In SingleNotify every node first waits for its children, then applies the operation
- Whereas in MultiNotify every node continuously scans the results from its kids and applies the operation on-the-fly
 - Reduces the congestion caused by the atomic increment of notify count
 - Allows overlapping between waiting for results and application of a reduction
 - The flag and the data always reside on a single cache line, so they are transferred together, not incurring extra cost
 - Not surprisingly, turned out to be faster 😊

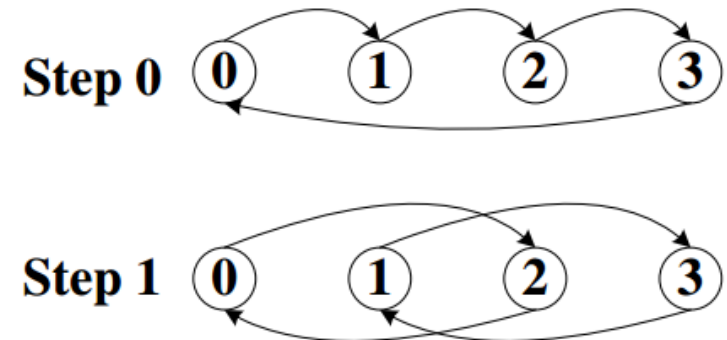
AllReduce implementations

(Reduce +) Broadcast:

- Uses the same generic trees as the reductions
- Based on receiver-driven communication
- Every node needs 2 cache lines -> (data & flag) and (notification_count)
- Doesn't require any barriers or memfences

Dissemination:

- Based on sender-driven communication
- Difficult to implement – every thread acts as sender and as receiver at the same time
- Causes a lot of CC traffic when used on core from different sockets
- Possible to combine with broadcast on later stages
- Our implementation is only for $n = 2^k$ threads
- Thread x on step t sends to thread $(x + 2^t) \bmod n$
- Runs in $k-1$ steps
- If used without data => essentially a barrier

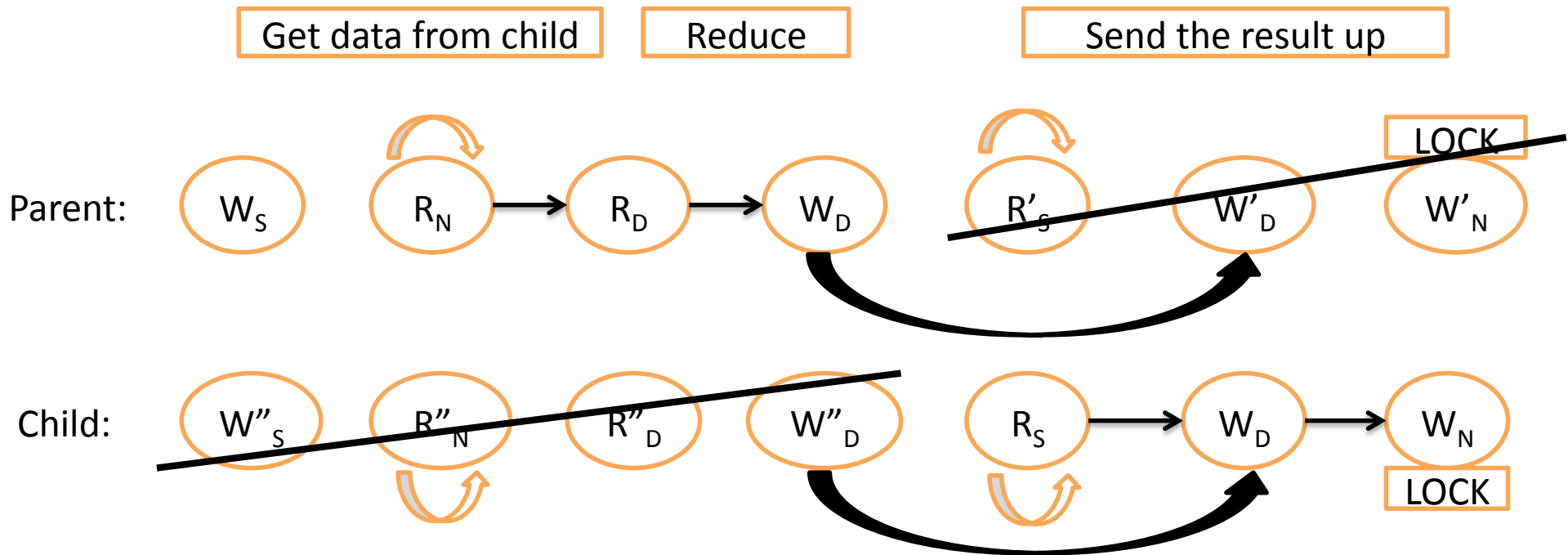


[Li, HPDC, 2013]

Correctness

- Testing against serial versions and parallel OpenMP and MPI versions.
Validation of bit identical results for the whole algorithms
- Implementation difficulties due to the nature of the problem
 - Lockouts
 - Internal data state invariant maintenance
 - Problems with multiple instances of reduce / broadcast issued by the same thread
- How we got rid of the synchronization barriers
 - Odd / even ping pong method
 - Enables certain amount of overlapping / pipelining in case of unfair scheduling
 - Could be done even better (modulo N)
- Why we don't need any memory fences (on x86 machines)
 - Weak memory model analysis

(Simplified) Example for SingleReduce

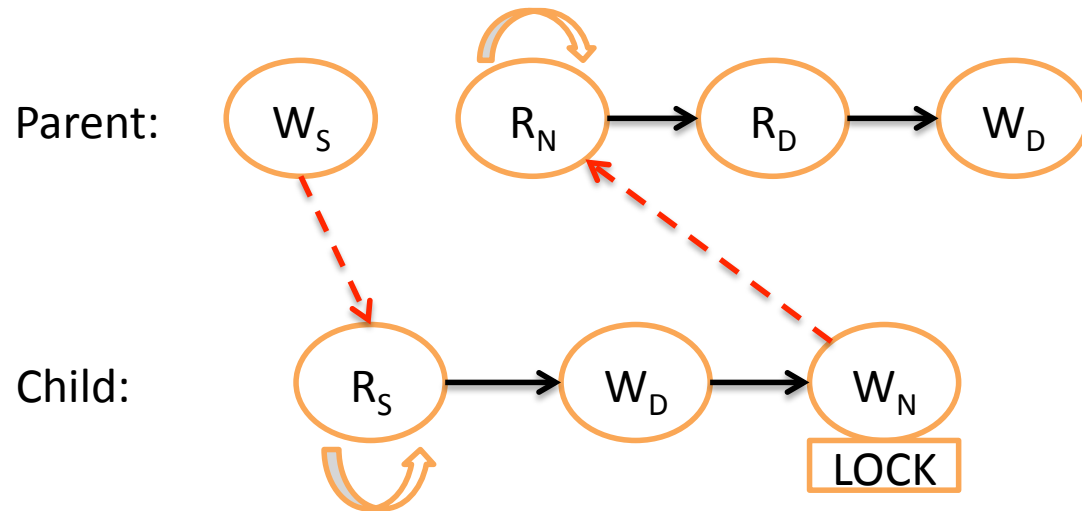


- We do not need the crossed-out parts
- We only care that the reduce results will be propagated upwards: $W_D \rightarrow W'_D$ & $W''_D \rightarrow W_D$



No W->R on x86

(Simplified) Example for SingleReduce

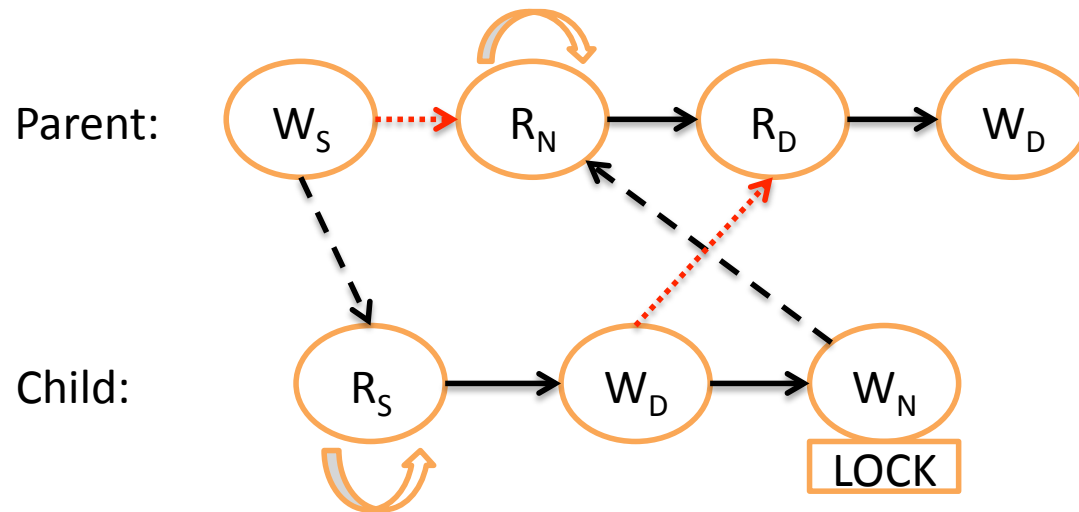


Add data dependencies



No W->R on x86

(Simplified) Example for SingleReduce



Now we know:

- ✓ The results of the child are written before being read by the parent
- ✓ Each thread's actions are observed in-order by the others (only in this domain)

Infer transitive relations

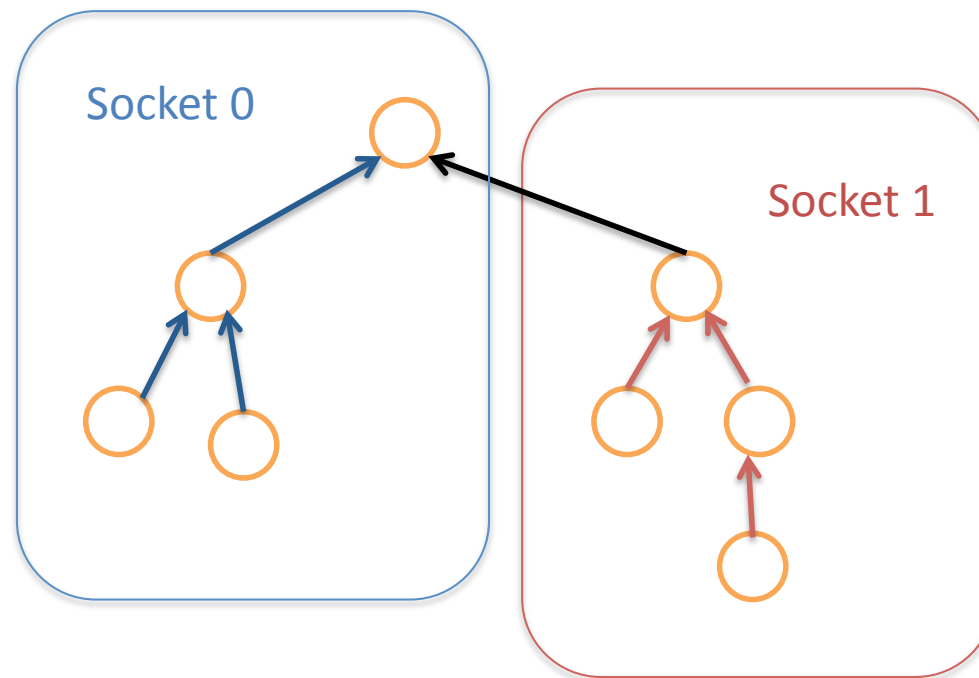


No W->R on x86

HWLOC Library

- NUMA aware library [Broquedis, ECPDNP, 2010]
- Enables to get topological information at runtime on the current processor
- Used to bind nodes close to each other in the tree to cores on the same sockets [Li, HPDC, 2013]

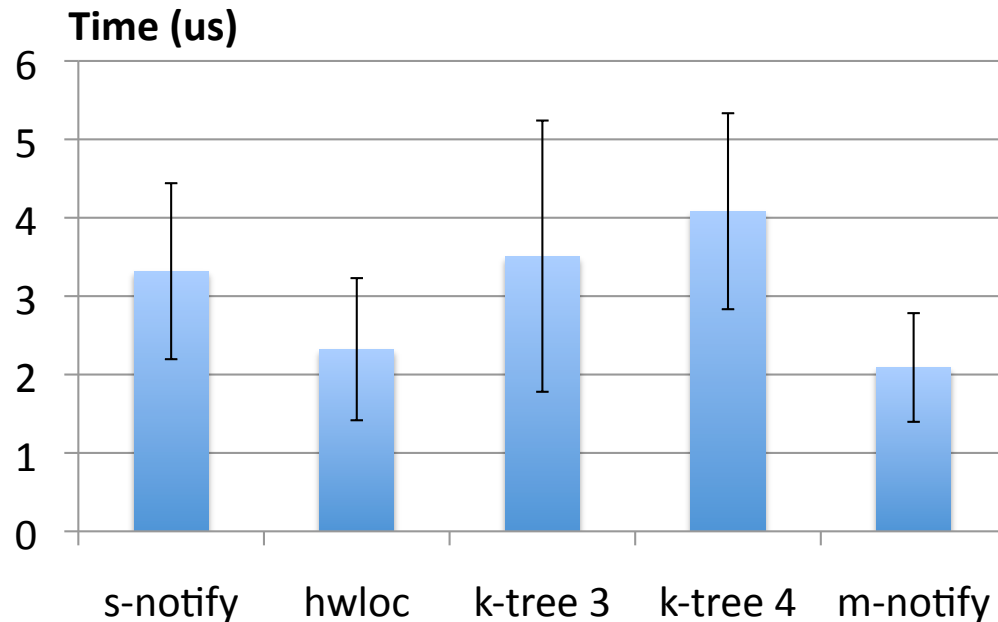
Reduce intersocket
memory access



- Motivation
- Reduce implementation
- **Results**

Performance of the reduce operation

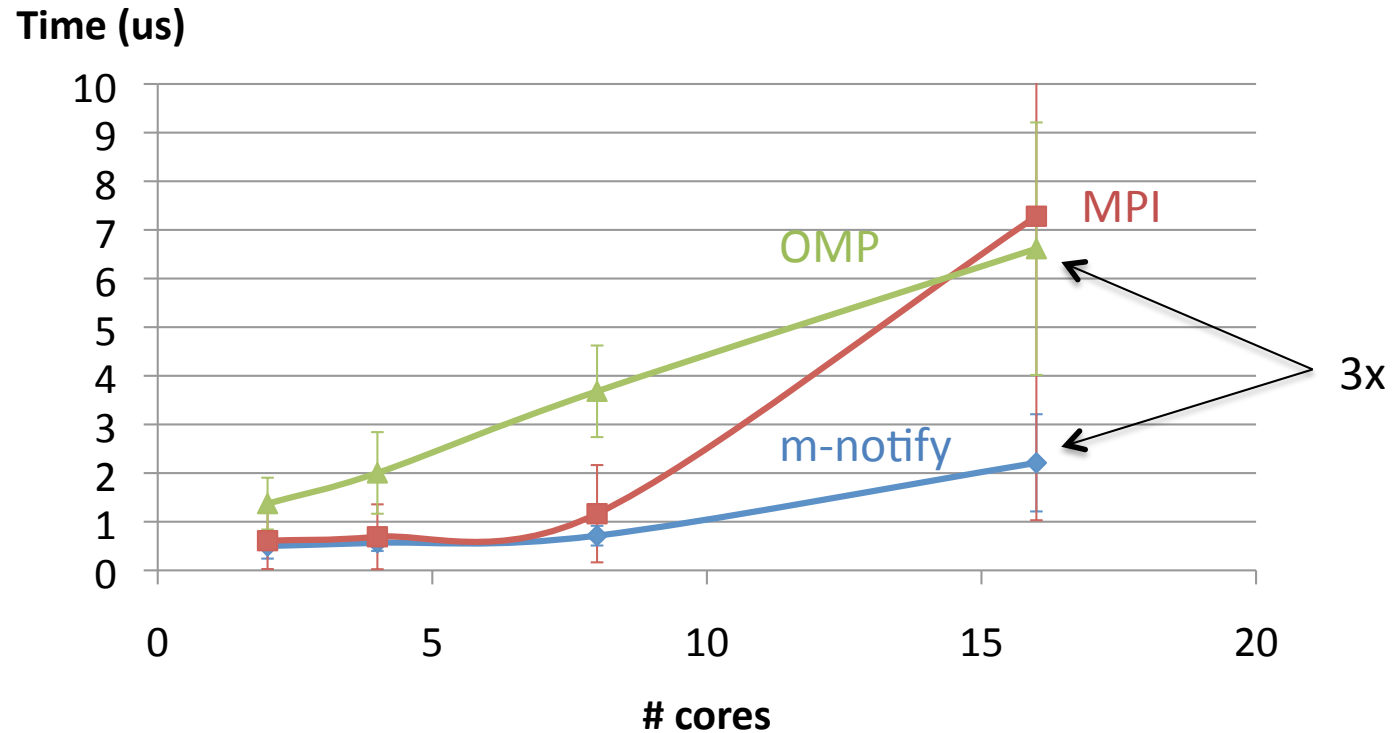
- Test system : 2 Intel Sandy bridge sockets node with 16 cores
- Test : reduce 16 doubles, calling reduce inside a timing loop 10^5 times



- Best performance with binary Tree
- HWLOC (NUMA aware) reduces time by a factor 1.7x
- Multinotify (m-notify) get best performance overall

Comparison with MPI and OMP libraries : reduce

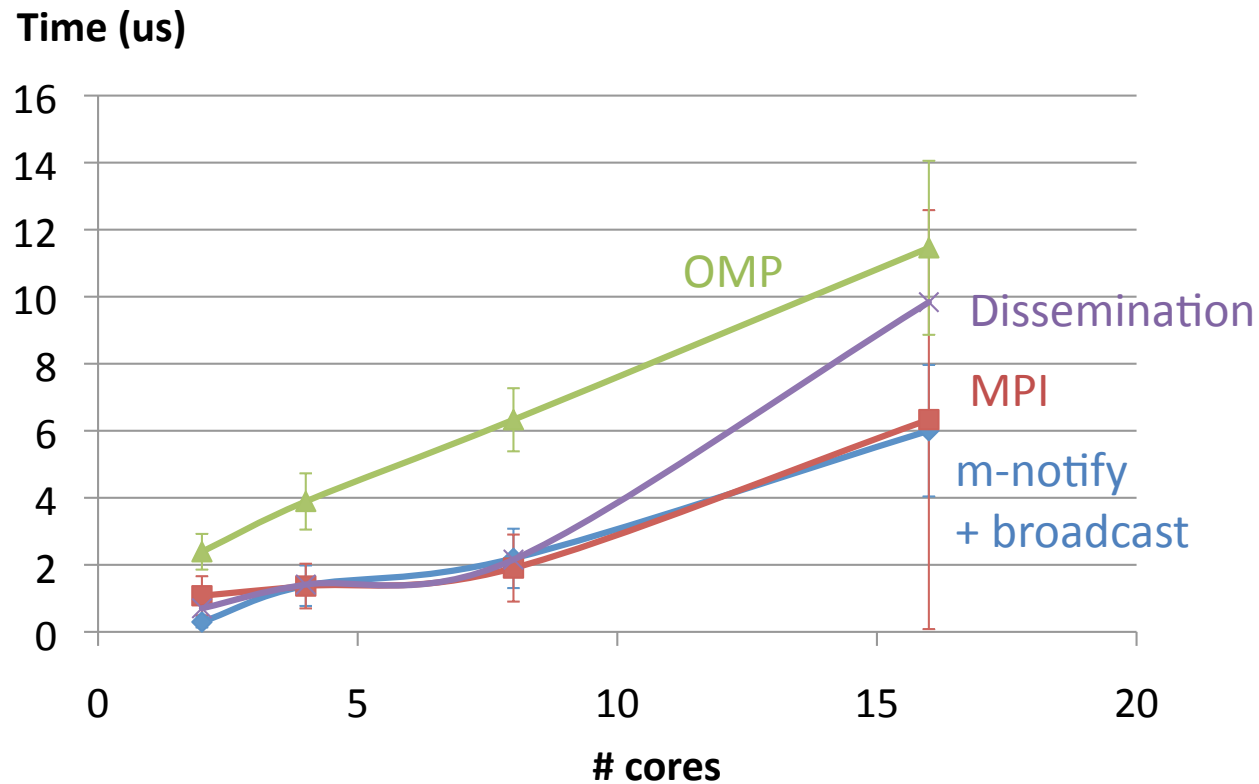
- Test : reduce N doubles using N cores



- Overall our implementation performs better
- Largest differences when using 2 sockets (3x improvement)

Comparison with MPI and OMP libraries : All reduce

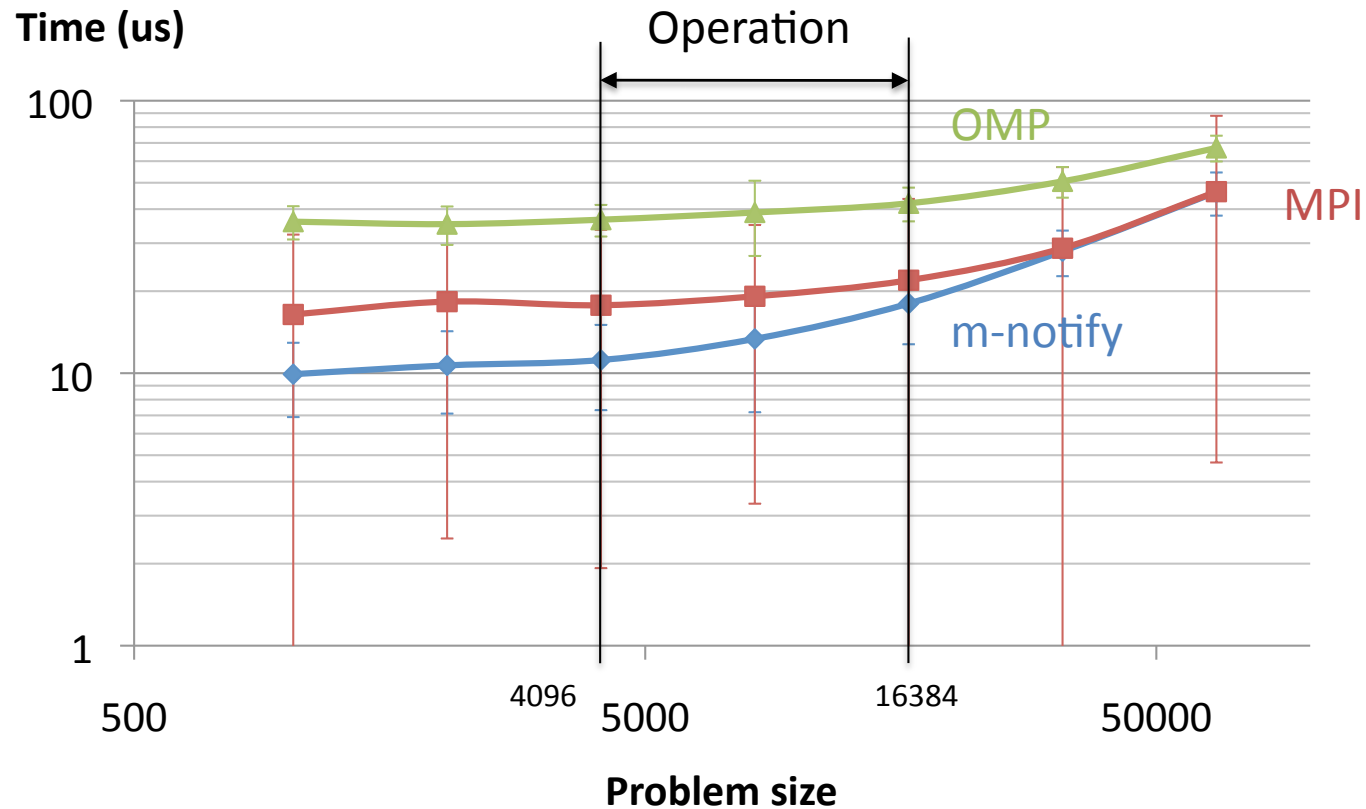
- Test : reduce N doubles using N cores



- Better performance with reduce + broadcast than dissemination
- Similar performance on average than MPI but much lower fluctuations

Performance of the Double Sweep sum

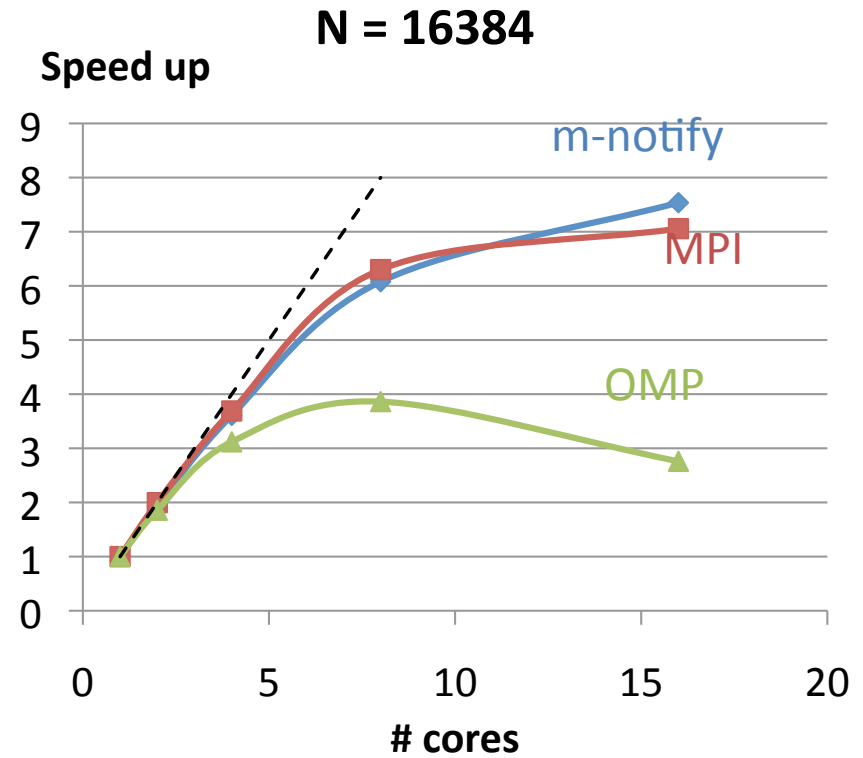
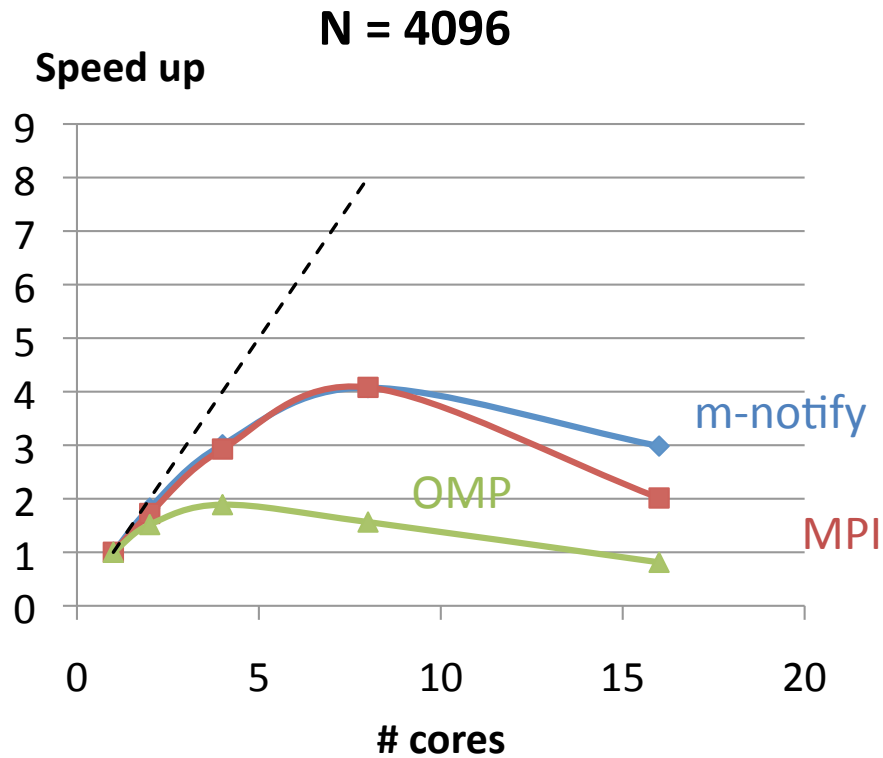
- Test : Sum of $v[\text{problem size}]$ using 16 cores



- Largest difference for smaller problem size
- For large enough vector the sum operation dominates the reduction

Scaling : Double Sweep sum

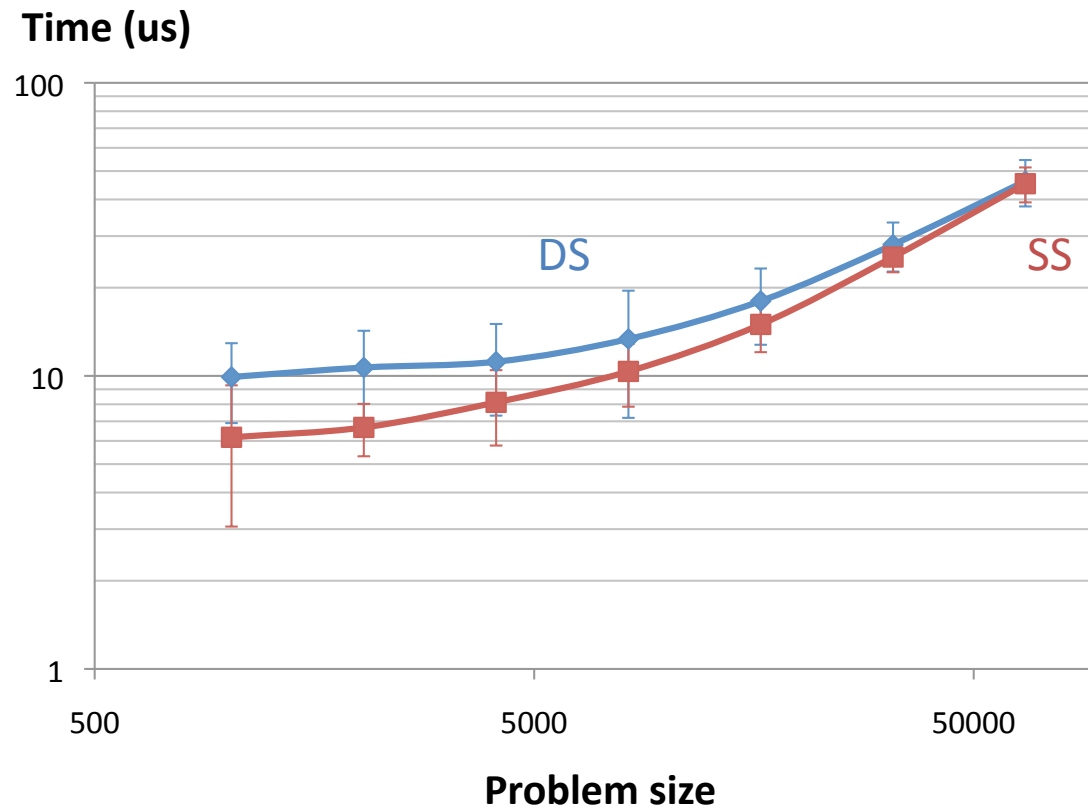
- Strong scaling



- For small problem decrease in performance when using all cores
- m-notify extends the scaling curve

Comparison between Double Sweep and Single Sweep sum

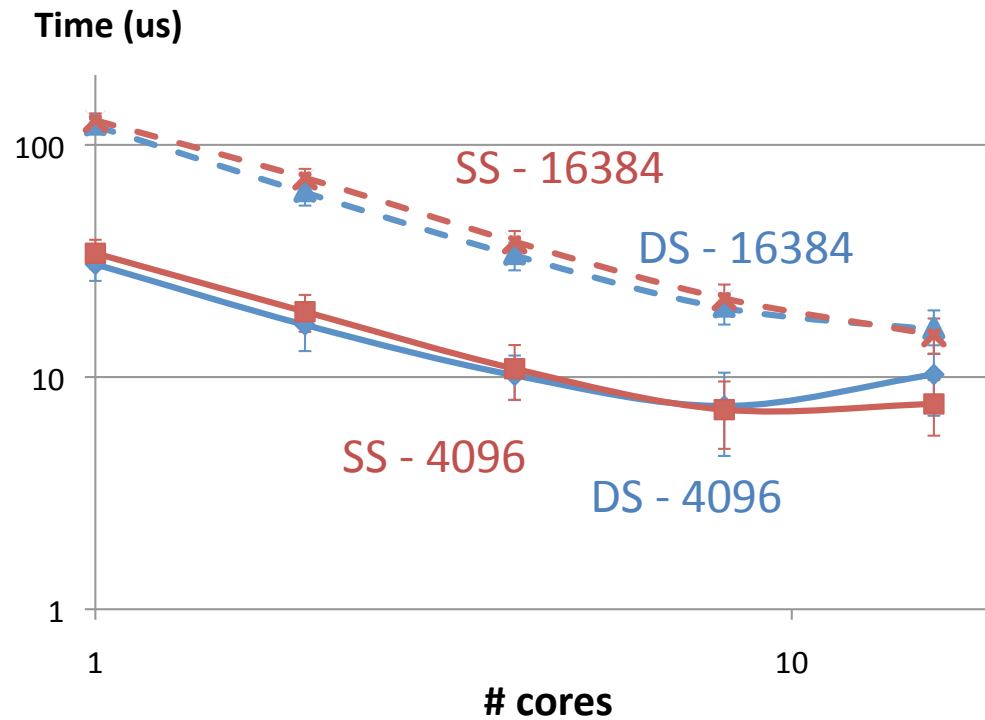
- Test : Sum of $v[\text{problem size}]$ using 16 cores



- Overall better performance of SS when using all cores

Scaling of double Sweep and Single Sweep sum

- Strong scaling considering a problem size $N = 4096$ or 16384



- For large problem the Single and Double sweep methods have similar performance
- For small problem the Single Sweep is better when using 2 sockets

Conclusion

- We have implemented different reduce and all reduce NUMA-aware operations to be applied to the bitwise reproducible sum
- The reduce operations do not require any synchronizations
- Our implementation generally performs better than the MPI and OMP version (up to 3x)
- The main advantage of the Single Sweep versus Double Sweep method is obtained when using more than one sockets, as it only requires one reduction