More Efficient Reduction Algorithms for Non-power-of-two Number of Processors in Message-Passing Parallel Systems

Rolf Rabenseifner (HLRS), Jesper Larsson Träff (NEC)
rabenseifner@hlrs.de trafi@ccrl-nece.de

University of Stuttgart, High-Performance Computing-Center Stuttgart (HLRS), www.hlrs.de
C&C Research Laboratories NEC Europe Ltd.
St. Augustin (www.ccrl-nece.de)

More Efficient Reduction Algorithms for Non-power-of-two Number of Processors in Message-Passing Parallel Systems

Methods

- Factorizing \( \# \text{processes} = 2^n \cdot q_1 \cdot q_2 \cdot q_3 \cdots q_m \) \( (n \geq 0, m \geq 0, \text{odd } q_i) \)
  - Processes logically in a hypercube with \( \text{dim} = n + m \)
  - Latency-optimization \( \rightarrow \) with full buffer exchange
  - Bandwidth-optimization \( \rightarrow \) with buffer halving (or splitting) \([\text{butterfly alg.}]\)
  - Mixing both methods for medium vector sizes
  - Factorization implemented with recursive distance doubling \((x^2)\) or \(xq_i\)
  - Factor \(q=2\): optimal butterfly algorithm
- Odd factors \(q_i\): several new algorithms
  - 2 elimination algorithms:
    - only with 3-to-2 elimination steps
    - with 3-to-2 and 2-to-1 elimination steps
  - Ring algorithm
- Buffer handling
- Further methods
Processes ordered as hypercube

- Reduction separately in each direction

Input vectors in 6 processes

Interim result

Final result vector

- **Pro:** Minimal number of messages – Goal: $O(\log \#\text{proc})$
- **Con:** Full size vector transfer in each direction

Splitting the vectors

Input vectors in 6 processes

Interim result: 2 processes → vector halving

Next interim result: 3 processes → scattered into 3 parts

- **Reduce_scatter phase**
- **Additional allgather phase → next slide**
In reverse sequence, with MPI_Allgather:

Reduction results gathered to all processes

Interim result vector parts after reduction phase

Interim result: 2 processes → vector doubling

- **Pro:** Sum of all message sizes < 2 * vector size
- **Con:** Doubling the number of messages

Allreduce / Reduce_scatter phase for power-of-two processes

- Communication and reduction protocol with **distance doubling**

  Rank: 0 1 2 3 4 5 6 7

  Shortcut:

  - Latency optimized — total reduction result on all processes
    - **Send total buffer** → receive and reduce it with own total buffer
    - **Send total buffer** × receive and reduce it with own total buffer

  - Bandwidth optimized — on each process: only \( \frac{1}{\#\text{processes}} \) of reduction result
    - **Send 2\text{nd} half of buffer** → receive and reduce it with own 1\text{st} half of own buffer
    - **Send 1\text{st} half of buffer** × receive and reduce it with own 2\text{nd} half of own buffer

- Distance and number of pairs in each block
  - Type of protocol:
    - \( A = \text{Allreduce} \)
    - \( R = \text{Reduce_scatter} \)
Allreduce = Reduce_scatter + Allgather (Butterfly)

- Pure Protocol:

```
Ranks:
0 1 2 3 4 5 6 7
1 2 3 4 5 6 7 0
2 3 4 5 6 7 0 1
3 4 5 6 7 0 1 2
4 5 6 7 0 1 2 3
5 6 7 0 1 2 3 4
6 7 0 1 2 3 4 5
7 0 1 2 3 4 5 6
```

- Mixed Protocol:

```
Ranks:
0 1 2 3 4 5 6 7
1 2 3 4 5 6 7 0
2 3 4 5 6 7 0 1
3 4 5 6 7 0 1 2
4 5 6 7 0 1 2 3
5 6 7 0 1 2 3 4
6 7 0 1 2 3 4 5
7 0 1 2 3 4 5 6
```

- Allgather protocol:

1. Send provisional result (receive after own provisional result)
2. Send provisional result (receive before own provisional result)

Methods

- Factorizing \#processes = 2^n \cdot q_1 \cdot q_2 \cdot q_3 \cdots q_m (n \geq 0, m \geq 0, odd q_i)
  - Processes logically in a hypercube with dim = n + m
- Full buffer exchange for best latency
- Buffer halving (or splitting) for best bandwidth (butterfly algorithm)
- Mixing both methods for medium vector sizes
- Factorization implemented with recursive distance doubling (×2) or ×q_i
- Factor q=2: optimal butterfly algorithm
- Handling odd factors q_i: several new algorithms
  - 2 elimination algorithms with doubling&halving:
    - only with 3-to-2 elimination steps
    - with 3-to-2 and 2-to-1 elimination steps
  - Ring algorithm
- Buffer handling
- Further methods
Methods for non-power-of-two number of processes

- 3-to-2-process (Triple) and 2-to-1-process (Double) reduction eliminates processes in the allreduce/reduce_scatter epoch, i.e., those processes are not used in further reduction steps

- 3-to-2-process reduction (Triple):
  - Allreduce/reduce_scatter phase:
    - Latency optimized
      - Send buffer
      - Reduce \( \oplus (p_0 + (p_1 + p_2)) \)
      - Reduce \( \oplus (p_0 + (p_1 + p_2)) \) [now eliminated]
    - Bandwidth optimized
      - Send buffer
      - Reduce \( \oplus (p_1 + p_2) \)
      - Send buffer
      - Reduce \( \oplus (p_0 + (p_1 + p_2)) \)
      - Send buffer
      - Reduce \( \oplus (p_0 + (p_1 + p_2)) \) [now eliminated]

This work was done together with Jesper Larsson Träff, NEC.
Example with $2^z - 1$ processes, bandwidth optimized

<table>
<thead>
<tr>
<th>Ranks</th>
<th>Message Size</th>
<th>Overlapping Protocol Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1/8 1/4 1/8</td>
<td>Total message size $&lt; 3 \times$ buffer size</td>
</tr>
<tr>
<td>1</td>
<td>1/8 1/4 1/8</td>
<td>Total computation size $&lt; 1.5 \times$ buffer size at each process that is never eliminated</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Same rules for any odd number of processes</td>
</tr>
</tbody>
</table>

This work was done together with Jesper Larsson Träff, NEC.

Example with odd number of processes, bandwidth optimized

<table>
<thead>
<tr>
<th>Ranks</th>
<th>Message Size</th>
<th>Overlapping Protocol Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1/8 1/4 1/8</td>
<td>Total message size $&lt; 3 \times$ buffer size</td>
</tr>
<tr>
<td>1</td>
<td>1/8 1/4 1/8</td>
<td>Total computation size $&lt; 1.5 \times$ buffer size at each process that is never eliminated</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Same rules for any odd number of processes</td>
</tr>
</tbody>
</table>

This work was done together with Jesper Larsson Träff, NEC.
Example with $2^z - 1$ processes, latency optimized

- Overlapping protocol blocks: $\lceil \log(#\text{processes}) \rceil + 1$ latencies
- Same rules for any non-power-of-two number of processes

This work was done together with Jesper Larsson Träff, NEC.

Methods

- Factorizing $\#\text{processes} = 2^n \cdot q_1 \cdot q_2 \cdot q_3 \cdot \ldots \cdot q_m$ ($n \geq 0$, $m \geq 0$, odd $q_i$)
  - Processes logically in a hypercube with $\text{dim} = n + m$
- Full buffer exchange for best latency
- Buffer halving (or splitting) for best bandwidth (butterfly algorithm)
- Mixing both methods for medium vector sizes
- Factorization implemented with recursive distance doubling ($\times 2$) or $\times q_i$
- Factor $q = 2$: optimal butterfly algorithm
- Handling odd factors $q_i$: several new algorithms
  - 2 elimination algorithms with doubling & halving:
    - only with 3-to-2 elimination steps
    - with 3-to-2 and 2-to-1 elimination steps
  - ring algorithm
- Buffer handling
- Further methods
2. Algorithm with Triple and Double Reduction steps

- 2-to-1-process reduction (Double):
  - Same protocol for "Latency optimized" and "Bandwidth optimized"
  - Allreduce/reduce_scatter epoch:
    - Send buffer receive and reduce \( (p_0 + p_i) \)
      - [now eliminated]
  - Allgather epoch:
    - Send buffer receive

Example with \( 2^z - 1 \) processes, bandwidth optimized

- Non-overlapping protocol blocks:
  - Total message size < \( 3 \times \) buffer size
  - Total computation size < \( 1.5 \times \) buffer size at each process that is never eliminated
- Same costs as with previous algorithm
- Same rules for any odd number of processes
Example with odd number of processes, bandwidth optimized

- Splitting the number of processes into
  - \( L \times (2\text{-process-symmetric-reduction}[F]) \)
  - \( M \times (2 \times 2\text{-process-symmetric-reduction}[F] + 2 \times 2\text{-1-reduction}[D]) \)
  - \( 1 \times (3\text{-2-reduction}[T]) \)

with \( 2(L+M+1) = 2^\left\lfloor \log \text{#processes} \right\rfloor \)

Methods

- Factorizing \( \text{#processes} = 2^n \cdot q_1 \cdot q_2 \cdot q_3 \cdot \ldots \cdot q_m \quad (n \geq 0, m \geq 0, \text{odd } q_i) \)
  - Processes logically in a hypercube with \( \text{dim}=n+m \)
- Full buffer exchange for best latency
- Buffer halving (or splitting) for best bandwidth (butterfly algorithm)
- Mixing both methods for medium vector sizes
- Factorization implemented with recursive distance doubling (\( \times 2 \)) or \( \times q_i \)
- Factor \( q=2 \): optimal butterfly algorithm

- Handling odd factors \( q_i \): several new algorithms
  - 2 elimination algorithms with doubling&halving:
    - only with 3-to-2 elimination steps
    - with 3-to-2 and 2-to-1 elimination steps

- Ring algorithm
  - Buffer handling
  - Further methods
Further Optimization of odd #processes with ring-algorithms

- **Latency optimization:**
  - Algorithm: Use Bruck’s “Concatenation” = Allgather algorithm (optimal, i.e., $O(\log p)$ for any #p)
  - Then reduce all data in each process

- **Bandwidth optimization:**
  - Algorithm: Divide input vector logically into (#processes) parts
    - Alltoall: send $j$th parts from all input vectors to process j
    - Then local reduction of all $j$th vector-parts in each process j
    - Then Bruck’s Allgather

The inner part for odd #processes can be additionally improved:

- **Latency optimization:**
  - Algorithm: Use Bruck’s “Concatenation” = Allgather algorithm
  - Then reduce all data in each process
  - **Criterion**
    - Latency: $O(\log(p))$ always faster
    - Transfer: $O(p - 1)$ faster if $p \geq 3.5$
    - Operations: $O(p - 1)$ faster if $p \geq 3$

<table>
<thead>
<tr>
<th>#processes = p</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
<th>11</th>
<th>13</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latencies</td>
<td>2/3</td>
<td>3/4</td>
<td>3/4</td>
<td>4/5</td>
<td>4/5</td>
<td>4/5</td>
<td>4/5</td>
</tr>
<tr>
<td>Transfer / input_buffer_size</td>
<td>2/3</td>
<td>4/4</td>
<td>4/4</td>
<td>6/4</td>
<td>8/5</td>
<td>10/5</td>
<td>12/5</td>
</tr>
<tr>
<td>Operations / input_buffer_elements</td>
<td>2/2</td>
<td>4/3</td>
<td>6/3</td>
<td>8/4</td>
<td>10/4</td>
<td>12/4</td>
<td>14/4</td>
</tr>
</tbody>
</table>
Further Optimization of odd #processes with ring-algorithms

- **Bandwidth optimization:**
  - Algorithm: Divide input vector logically into (#processes) parts
  - Alltoall: send \( j \)th parts from all input vectors to process \( j \)
  - Then local reduction of all \( j \)th vector-parts in each process \( j \)
  - Then Bruck’s Allgather

- **Criterion**
  - Bandwidth optimization
    - Algorithm:
      - Divide input vector logically into (#processes) parts
      - Alltoall: send \( j \)th parts from all input vectors to process \( j \)
      - Then local reduction of all \( j \)th vector-parts in each process \( j \)
      - Then Bruck’s Allgather

<table>
<thead>
<tr>
<th>#processes</th>
<th>Latency</th>
<th>Transfer</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4/4</td>
<td>1.33/2.00</td>
<td>1.82/2.75</td>
</tr>
<tr>
<td>5</td>
<td>7/6</td>
<td>1.60/2.50</td>
<td>1.85/2.75</td>
</tr>
<tr>
<td>7</td>
<td>9/7</td>
<td>1.71/2.50</td>
<td>1.87/2.75</td>
</tr>
<tr>
<td>9</td>
<td>12/8</td>
<td>1.78/2.75</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>14/8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>16/8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>18/9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Better algorithms are shown with **.

Best algorithm for odd factor \( q_i \)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Latency</th>
<th>Transfer</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lat.-opt. Ring</td>
<td>( O(\log(p)) ) + ( s( O(p-1) + O(1) ) )</td>
<td>( O(0.5) ) + ( s( O(2) + O(1) ) )</td>
<td>( O(0.5) ) + ( s( O(2) + O(1) ) )</td>
</tr>
<tr>
<td>Lat.-opt. Elim.</td>
<td>( O(\log(p)+1) ) + ( s( O(\log(p)+1) + O(1) ) )</td>
<td>( O(1.5) ) + ( s( O(2) + O(1) ) )</td>
<td>( O(1.5) ) + ( s( O(2) + O(1) ) )</td>
</tr>
<tr>
<td>Bw.-opt. Elim.</td>
<td>( O(2\log(p)) ) + ( s( O(3.2/p) + O(1.5) ) )</td>
<td>( O(1.5) ) + ( s( O(2) + O(1) ) )</td>
<td>( O(1.5) ) + ( s( O(2) + O(1) ) )</td>
</tr>
<tr>
<td>Bw.-opt. Ring</td>
<td>( O(p+1) ) ( \log(p)+1 ) ( ) + ( s( O(2) + O(1) ) )</td>
<td>( O(0.5) ) + ( s( O(2) + O(1) ) )</td>
<td>( O(0.5) ) + ( s( O(2) + O(1) ) )</td>
</tr>
</tbody>
</table>

- \( p \)=number of processes, \( s \)=vector size
- \( ^* \)=some additional messages to prohibit local buffer copy in Bruck’s allgather algorithm
- \( ^{**} \)=may be reduced to \( O(24) \) with \( \text{ring}(5) \)*\( \text{ring}(5) \)*\( \text{ring}(5) \) instead of \( \text{ring}(125) \)

Example for \( p=125 \)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Latency</th>
<th>Transfer</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lat.-opt. Ring</td>
<td>( O(7) ) + ( s( O(24) + O(124) ) )</td>
<td>( O(0.5) ) + ( s( O(2) + O(1) ) )</td>
<td>( O(0.5) ) + ( s( O(2) + O(1) ) )</td>
</tr>
<tr>
<td>Lat.-opt. Elimi.</td>
<td>( O(8) ) + ( s( O(3) + O(1) ) )</td>
<td>( O(0.5) ) + ( s( O(2) + O(1) ) )</td>
<td>( O(0.5) ) + ( s( O(2) + O(1) ) )</td>
</tr>
<tr>
<td>Bw.-opt. Elimi.</td>
<td>( O(14) ) + ( s( O(2) + O(1) ) )</td>
<td>( O(0.5) ) + ( s( O(2) + O(1) ) )</td>
<td>( O(0.5) ) + ( s( O(2) + O(1) ) )</td>
</tr>
<tr>
<td>Bw.-opt. Ring</td>
<td>( O(132)^{**} )</td>
<td>( O(0.5) ) + ( s( O(2) + O(1) ) )</td>
<td>( O(0.5) ) + ( s( O(2) + O(1) ) )</td>
</tr>
</tbody>
</table>
Buffer handling

- If operation is commute
  - No local buffer copying,
  - Except in Bruck’s allgather, used in latency optimized ring (i.e., with small vectors)

- If operation is not commute (only with user defined operations)
  - Minimizing buffer copying be receiving data to optimized location

Further optimization methods

- Pipelining:
  - Splitting vectors into chunks
  - Overlaying chunk message transfer with reduction already received chunk

- Segmentation
  - For algorithms that do not use all processors at the same time
    - E.g., MPI Reduce with binomial tree
  - Executing the operation for several segments of the input vector
  - (Recursive halving and doubling should be faster)
  - Smaller scratch buffers
Measurements on 128 x Dual-Opteron with Myrinet 2000

MPIAllreduce(sum, double) bandwidth: Ratio “new routine” / “mpich1.2.5..10-gm-pgi”

1 MPI process per dual-SMP-node

2 MPI processes per dual-SMP-node

Is automatic protocol-choosing real best?

Bandwidth-ratio “best of some choices” / “automatic choosing”

In few cases, manual choices may be up to 1.5 better than automatic
Summary

- Optimized MPI_Allreduce
  - any number of processes $#p$
  - especially non-power-of-two $#p$
  - any vector length
- Optimal latency for $#p = 2^n$ or $#p = 2^n \cdot 3$ → $O(\lceil \log_2 p \rceil)$
  - one additional round for other $#p$ → $O(\lceil \log_2 p \rceil + 1)$
- Bandwidth optimized: if $#p = 2^n \cdot q$ then $O(2(1 + 1/2^{n+1})s)$ between 2 and 3
- Smooth transition
  - from latency to bandwidth optimized
- Implementations
  - By NEC: (non-disclosed, 3-2-elimination)
  - By HLRS: freely available on request (2-1-elimination and ring)
    (e-mail to rabenseifner@hlrs.de)