Architectural Support for Parallel Reduction in Scalable Shared-Memory Multiprocessors

María Jesús Garzarán, M. Prvulovic§, Y. Zhang§, A. Jula*, H. Yu*, L. Rauchwerger*, and J. Torrellas§

U. of Zaragoza, Spain * Texas A&M University §University of Illinois
Motivation

- Reductions are important in scientific codes

```cpp
for (...) {
    ...
    x = x \textit{op} \text{ expression};
    ...
}
```

- Reduction parallelization algorithms are not scalable

```cpp
\texttt{parallel\_for} (...) {
    ...
    lock(w[x[i]]);
    x = x \textit{op} \text{ expression};
    unlock(w[x[i]]);
    ...
}
```
Contribution

- New architectural support for parallel reductions in CC-NUMA:
  - Speeds-up parallel reduction
  - Makes parallel reduction scalable
- Increase 16-proc speedup from 2.7 to 7.6
Outline

- **Background on Reduction**
- Parallelizing Reduction in Software
- Our contribution: Private Cache-Line Reduction (PCLR)
- Evaluation
- Related Work
- Conclusions
Background on Reduction

- Reduction operation:
  
  ```
  for (...) {
    ...
    x = x op expression;
    ...
  }
  ```

  - `op`: associative and commutative operator
  - `x`: does not occur in expression or anywhere else in the loop

- There may be complex flow dependences across iterations
  - Parallelization of reductions needs special transformations
    ```
    for (i= 0; i< Nodes; i++)
    w[x[i]] += expression;
    ```
Outline

- Background on Reduction
- **Parallelizing Reduction in Software**
- Our contribution: Private Cache-Line Reduction (PCLR)
- Evaluation
- Related Work
- Conclusions
Parallelizing Reduction in Software (I)

- Enclose access in unordered critical section

```c
parallel_for(...){
    lock(w[x[i]]);
    w[x[i]]+=expression;
    unlock(w[x[i]]);
}
```

- Drawbacks:
  - Overhead
  - Contention increases with the # of processors.
Parallelizing Reduction in Software (II)

- Each processor accumulates on a private array

```c
for (i=0; i<array_size; i++)
    w_priv[pid][i]=0;
```

Clear Private Array

```c
parallel_for (...)
    w_priv[pid][x[i]]+=expression;
barrier();
```

Merge

```c
for (i=MyBegin; i<MyEnd; i++)
    for (p=0; p<NumProcs; p++)
        w[i]+=w_priv[p][i];
barrier();
```
Drawbacks of the Privatization Method

- Initialization phase
  - Sweeps the cache before the parallel loop starts

```c
for (i=0; i<array_size; i++)
    w_priv[pid][i]=0;
```

Clear Private Array
Drawbacks of the Privatization Method

- Merging phase:
  - Work proportional to the array size

![Diagram showing privatization and merging]

Private P1 \hspace{1cm} Private P2

\[\text{Shared array} \]

P1 \hspace{1cm} P2
Drawbacks of the Privatization Method

- Merging phase:
  - Work proportional to the array size

This method is not scalable
Outline

- Background on Reduction
- Parallelizing Reduction in Software
- **Our contribution: Private Cache-Line Reduction (PCLR)**
- Evaluation
- Related Work
- Conclusions
Main Idea of PCLR

Use non-coherent cache lines in the different processors as the temporary private arrays

• Remove initialization phase
• Accumulate on cache lines
• Remove the merging phase
Removing Initialization

- Caches lines are initialized on-demand on cache misses

```c
parallel_for (...) 
  w_priv[pid][x[i]]+=expression;
barrier();
```

```c
for (i=MyBegin;i<MyEnd;i++)
  for (p=0; p<NumProcs; p++)
    w[i]+=w_priv[p][i];
barrier();
```
Removing Initialization

Diagram:
- CPU
- Cache
- Memory
- Neutral Element
- Directory
- Network

Arrows indicate the flow of data or control between components.
Accumulating on Cache Lines

- No need to allocate a private array

```c
parallel_for (...) 
    w[x[i]] += expression;
barrier();
```

```c
for (i=MyBegin; i<MyEnd; i++)
    for (p=0; p<NumProcs; p++)
        w[i] += w_priv[p][i];
barrier();
```

Merge
Removing Initialization

- Load
- Add
- Store

Accumulate on private cache lines

Directory

Memory

Network
Removing the Merge

- Lines are accumulated at the home on displacements

```c
parallel_for (...)
    w[x[i]]+=expression;
CacheFlush();
barrier();
```
Removing the Merge
Recognizing Reduction Data

- Naïve approach
  - new load & store instructions

- Advanced Mechanism
  - shadow addresses [like Impulse]
The directory recognizes shadow addresses and translates them into the original ones.
Atomicity Issues

- A reduction $\varphi$ is composed of a pair load-store inst
- A problem appears if a cache line is displaced between the reduction load and the store

load r1, addr
add r1, r1, r3
store r1, addr
Atomicity Issues

- A reduction operation is composed of a pair load-store instruction.
- A problem appears if a cache line is displaced between the reduction load and the store.

PC → load r1, addr
add r1, r1, r3
store r1, addr

r3 = 1

Home memory addr = 2

Final result should be 8
Atomicity Issues

- A reduction $\phi$ is composed of a pair load-store inst
- A problem appears if a cache line is displaced between the reduction load and the store

```
load r1, addr
PC →
add r1, r1, r3
store r1, addr
```

```
r3 = 1
r1 ← 5
```

```
Home memory addr = 2 + 5
Final result should be 8
```
Atomicity Issues

- A reduction operation is composed of a pair load-store instruction.
- A problem appears if a cache line is displaced between the reduction load and the store.

```
load r1, addr
add r1, r1, r3
store r1, addr
```

```
Home memory addr = 2 + 5
Final result should be 8
```
Atomicity Issues

- A reduction $\phi$ is composed of a pair load-store inst
- A problem appears if a cache line is displaced between the reduction load and the store

```
load r1, addr
add r1, r1, r3
store r1, addr
```

```
r3 = 1
r1 ← 5
r1 ← 6
```

Home memory \( \text{addr} = 2 + 5 \)
Final result should be 8
Atomicity Issues

- A reduction \( \phi \) is composed of a pair load-store inst
- A problem appears if a cache line is displaced between the reduction load and the store

\[
\begin{align*}
\text{load } r1, \text{ addr} \\
\text{add } r1, r1, r3 \\
\text{store } r1, \text{ addr}
\end{align*}
\]

\[
\begin{align*}
\text{r3} &= 1 \\
r1 &= 5 \\
r1 &= 6
\end{align*}
\]

\[
\text{Home memory addr} = 2 + 5 + 6
\]

Final result should be 8
Solution to the Atomicity Problem

- Atomic exchange neutral element - memory contents

```
load r1, addr
add r1, r1, r3
store r1, addr

load r1, neutral
swap r1, addr
add r1, r1, r3
store r1, addr
```

The line can be displaced between swap and store
Summary of Architectural Support

Special support in directory/network controller

- Intercept a reduction cache miss & return neutral elements
- Use ALU to merge data at displacements or at loop’s end
- Reduction lines can be dirty in multiple caches
Advantages of PCLR

- Remove initialization phase:
  - Avoid cache sweeping

- No need allocate private arrays

- Remove merging phase:
  - Work at the end: proportional to cache size instead of to array size
Outline

- Background on Reduction
- Parallelizing Reduction in Software
- Our contribution: Private Cache-Line Reduction (PCLR)
- Evaluation
- Related Work
- Conclusions
Evaluation Methodology

- Execution-driven simulator
- Scalable multiprocessor: 4-16 processors
- Detailed superscalar processor model
- 32 KB L1 2-way, 512 KB L2 4-way
- Round-trip latencies non-contention:
  - L1 (2 cyc), L2 (10 cyc), Local Memory (104 cyc), 2-hop (297 cyc)
- Floating-point unit in the directory controller:
  - fully pipelined,
  - latency (6 processor cyc).
Applications

- Fortran and C codes

- Loops with reduction ops identified by the compiler
  - Euler [HPF-2 suite]
  - Equake [SPECfp2000 suite]
  - Vml* [Sparse BLAS suite]
  - Charmm* [CHARMM appl]
  - Nbf* [GROMOS appl]

Reduction loops account for an avg. of 81.2% of Tsequential

* Kernels
Mechanisms Evaluated

- Software implementation
  - \textit{Sw}: Privatized arrays and merge at loop’s end

- Two implementations of PCLR:
  - \textit{Hw}: Hardwired directory controller
  - \textit{Flex}: Programmable directory controller like MAGIC [FLASH]
    - Implement PCLR with no HW changes in directory controller
    - Contention and latency increase
Execution Time for 16 Processors

Average speedups:
Sw (2.7), Hw (7.6), Flex (6.4)
Scalability

- Sw scales poorly
- Merging phase limits speedups (Amdahl’s Law)
- PCLR truly scalable
Outline

- Background on Reduction
- Parallelizing Reduction in Software
- Our contribution: Private Cache-Line Reduction (PCLR)
- Evaluation
- Related Work
- Conclusions
Related Work

- Larus et al. [ASPLOS94]
  - Reconcilable Shared Memory
- Zhang et al. [Illinois TR]
  - Modified architecture for speculative parallelization
- Hardware support for synchronization
  - Fetch&Add (NYU Ultracomputer)
  - Fetch&Op (IBM RP3)
  - Support for combining trees
  - Memory-based synchronization primitives (Cedar)
  - Set of synchronization primitives (Goodman et al.)
Outline

- Background on Reduction
- Parallelizing Reduction in Software
- Our contribution: Private Cache-Line Reduction (PCLR)
- Evaluation
- Related Work
- Conclusions
Conclusions

- Proposed novel architectural support for scalable parallel reduction
- Architectural modifications concentrated in directory controller
- Average speedup for 16 processors increases from 2.7 to 7.6
Architectural Support for Parallel Reduction in Scalable Shared-Memory Multiprocessors

María Jesús Garzarán, M. Prvulovic, Y. Zhang, A. Jula, H. Yu, L. Rauchwerger, and J. Torrellas

garzaran@posta.unizar.es
http://iacoma.cs.uiuc.edu
Parallelizing Reductions

- Two steps
  - Recognizing the reduction variable
    - syntactically pattern-matching
    - verify that the operator is commutative & associative
    - verify reduction variable is not used anywhere else
  - Apply a parallelization transformation
Reduction

- Reduction operation:
  
  ```
  for (...) 
  x = x op expression
  ``
  
  - `op`: associative and commutative operator
  - `x`: does not occur in expression or anywhere else

- Parallelization of reductions needs special transformations
  
  ```
  for (i=0; i<Nodes; i++)
  w[x[i]] += expression;
  ```
Recognizing Reduction Data (I)

- Naïve approach

  Special load and store for reduction accesses
  (load&addint, load&addfloat …)

  - Special messages on cache-miss
  - Bring the data into the cache in a special state
  - Special displacement message
Recognizing Reduction Data (I)

Advanced Mechanism

Shadow addresses [Impulse]
- Use a *shadow array* during the reduction
- Shadow array is mapped to shadow physical addresses
- Directory controller
  - Recognizes shadow physical addresses
  - Translates them into the physical address corresponding to the original reduction array.
Recognizing Reduction Data

Advanced Mechanism

Shadow addresses [Impulse]
- Use a\textit{ shadow array} during the reduction
- Shadow array is mapped to shadow physical addresses
- Directory controller
  - Recognizes shadow physical addresses
  - Translates them into the physical address corresponding to the original reduction array.
Reduction

- Reduction operation

\[
\text{for (i=0;i<Nodes;i++)} \\
\quad w[x[i]] += \text{expression};
\]

- \(+: \) associative and commutative operator
- \(w: \) does not occur in expression or anywhere else

- Parallelization of reductions needs special transformations
  - There may be flow dependences between iterations
Additional Use of PCLR

- Dynamic Last Value assignment
  - Loop parallelized through privatization
  - The privatized variable is used after the loop
  - The compiler cannot determine the last writing iteration

```c
for (i =0; i<N; i++)
    if (f(i)){
        A[g[i]]= ...;
        ... = A[g[i]];
    }
```

Dynamic last Value assignment
- Identify the private array with the last value
- Copy the value from the private var. to the shared var.
Drawbacks of the Privatization Method

- **Merge phase**

  # processors increase

  \[
  \begin{cases}
  \text{portion of the array to merge decreases} \\
  \text{# private arrays to merge increases}
  \end{cases}
  \]

  **Work of Merging is always proportional to array size**

  ```
  for (i=MyBegin;i<MyEnd;i++)
      for (p=0; p<NumProcs; p++)
          w[i]+=w_priv[p][i];
  barrier();
  ```

  **This method is not scalable**
Removing Initialization

- CPU
- Cache
- Directory
- Memory
- Neutral Element
- Network

miss
Accumulating on Cache Lines
Removing the Merge
Atomicity Concerns

Solution:
- Special load and store instructions
  - load\&pin and store\&unpin
- Small number of Cache Pin Registers (CPR). Each has:
  - tag of the pinned line
  - counter

Operation
- **Load\&pin**: Allocate CPR; Set tag; Set counter = 1.
- **Store\&unpin**: Decrease counter. Deallocate CPR if counter eq 0.
- Displacement: Prevented for the lines with a match in any CPR