Cost-Effective and Efficient Parallel Computing

Sandhya Dwarkadas
Department of Computer Science
University of Rochester

Why Parallel Computing?

Fundamentally limited by the speed of a sequential processor

Need to use multiple processors to get further improvement

Flynn's Taxonomy of Parallel Architectures

Single instruction stream, single data stream (SISD)
- uniprocessor

Single instruction stream, multiple data streams (SIMD)

Multiple instruction streams, single data stream (MISD)
- no existing commercial implementation

Multiple instruction streams, multiple data streams (MIMD)

Types of Parallelism

Data

Functional (Task)

Master-Slave

Parallel Programming Models

Data Parallel

Shared memory

Message passing

Data Parallel Programming

High Performance Fortran (HPF)

Fortran-D

Power C/Fortran

Shared Memory Programming

Shared address space

- Processes/threads
- Shared memory allocation
- Synchronization
  - Locks
  - Barriers
  - Flags

Message Passing Programming

Multiple private address spaces

- Processes
- Communication via explicit messages (sends and recvs)
- Synchronization implicit in the message

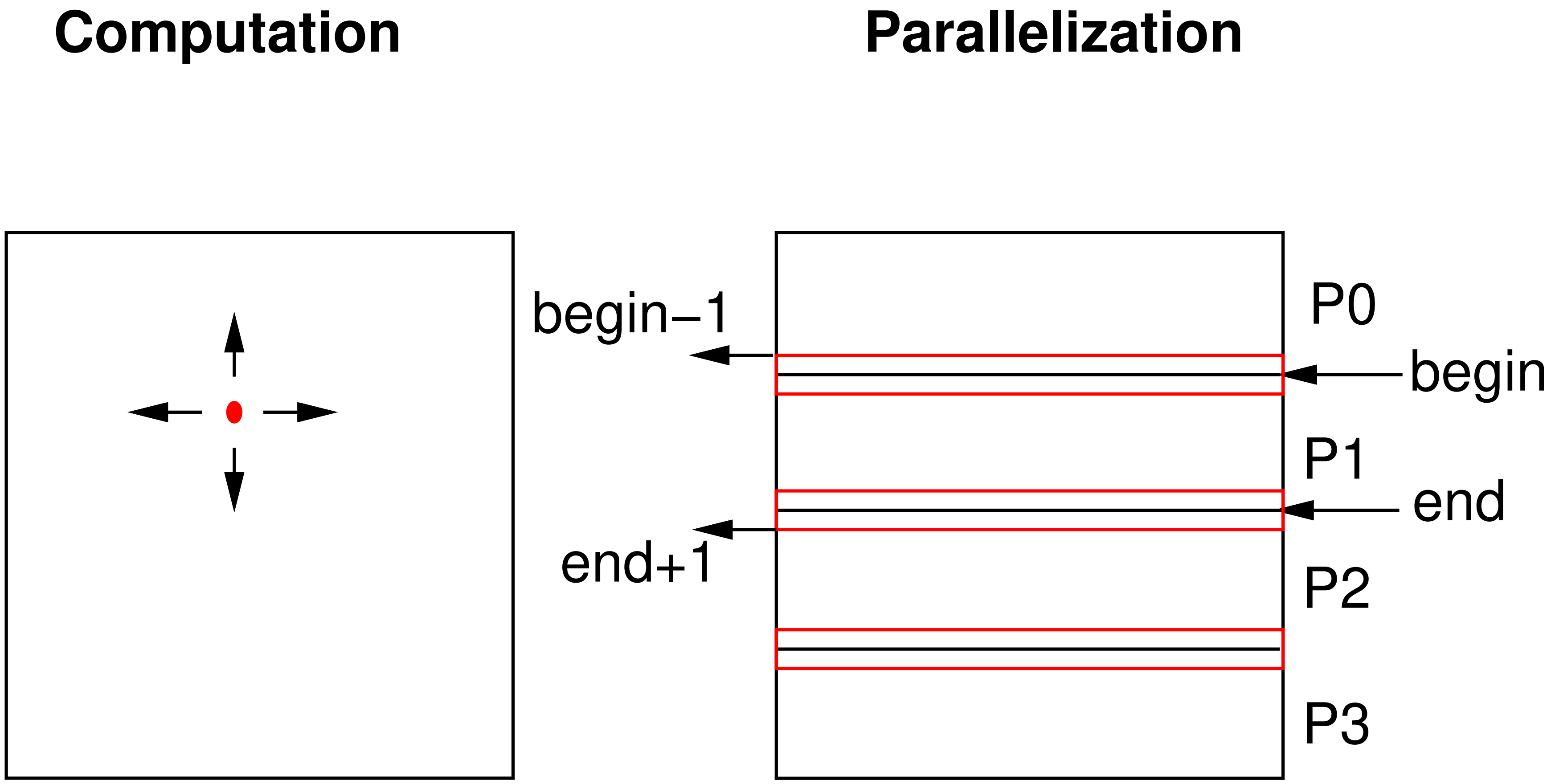
Example Program - Jacobi

```
for (k = 0; k < 100; k++) {

    for (j = 1; j < M-1; j++)
        for (i = 1; i < M-1; i++)
            a[j][i] = (b[j][i-1] + b[j][i+1] +
                       b[j-1][i] + b[j+1][i])/4;

    for (j = 1; j < M-1; j++)
        for (i = 1; i < M-1; i++)
            b[j][i] = a[j][i];
}
```

Jacobi Parallelization



a(i,j) = **F** (b(i−1,j),b(i+1,j),b(i,j−1),b(i,j+1))

Data Parallel Version of Jacobi (Power C)

```
for (k = 0; k < 100; k++) {

#pragma parallel shared(a, b) local(i,j)
{
    #pragma pfor
    for (j = 1; j < M-1; j++)
        for (i = 1; i < M-1; i++)
            a[j][i] = (b[j][i-1] + b[j][i+1] +
                       b[j-1][i] + b[j+1][i])/4;
}

#pragma parallel shared(a, b) local(i, j)
{
    #pragma pfor
    for (j = 1; j < M-1; j++)
        for (i = 1; i < M-1; i++)
            b[j][i] = a[j][i];
}
}
```

Shared Memory Version of Jacobi

```
for (k = 0; k < 100; k++) {

    for (j = begin; j < end; j++)
        for (i = 1; i < M-1; i++)
            a[j][i] = (b[j][i-1] + b[j][i+1] +
                       b[j-1][i] + b[j+1][i])/4;
    barrier();

    for (j = begin; j < end; j++)
        for (i = 1; i < M-1; i++)
            b[j][i] = a[j][i];
    barrier();
}
```

Message Passing Version of Jacobi

```
for (k = 0; k < 100; k++) {

    for (j = begin; j < end; j++)
        for (i = 1; i < M-1; i++)
            a[j][i] = (b[j][i-1] + b[j][i+1] + b[j-1][i] + b[j+1][i])/4;

    for (j = begin; j < end; j++)
        for (i = 1; i < M-1; i++)
            b[j][i] = a[j][i];

    if (p != (nprocs - 1)) {
        send(p+1, b[end-1]);
        recv(p+1, b[end]);
    }
    if (p != 0) {
        send(p-1, b[begin]);
        recv(p-1, b[begin-1]);
    }
}
```

Why Is Parallel Computing Hard?

Amdahl's law - insufficient available parallelism

$$Speedup = \frac{1}{\frac{Fraction_{enhanced}}{Speedup_{enhanced}} + (1 - Fraction_{enhanced})}$$

Overhead of communication and coordination

Portability - knowledge of underlying architecture often required

Parallel Processing Issues

Data sharing - single versus multiple address spaces

Process coordination - synchronization using lock or messages

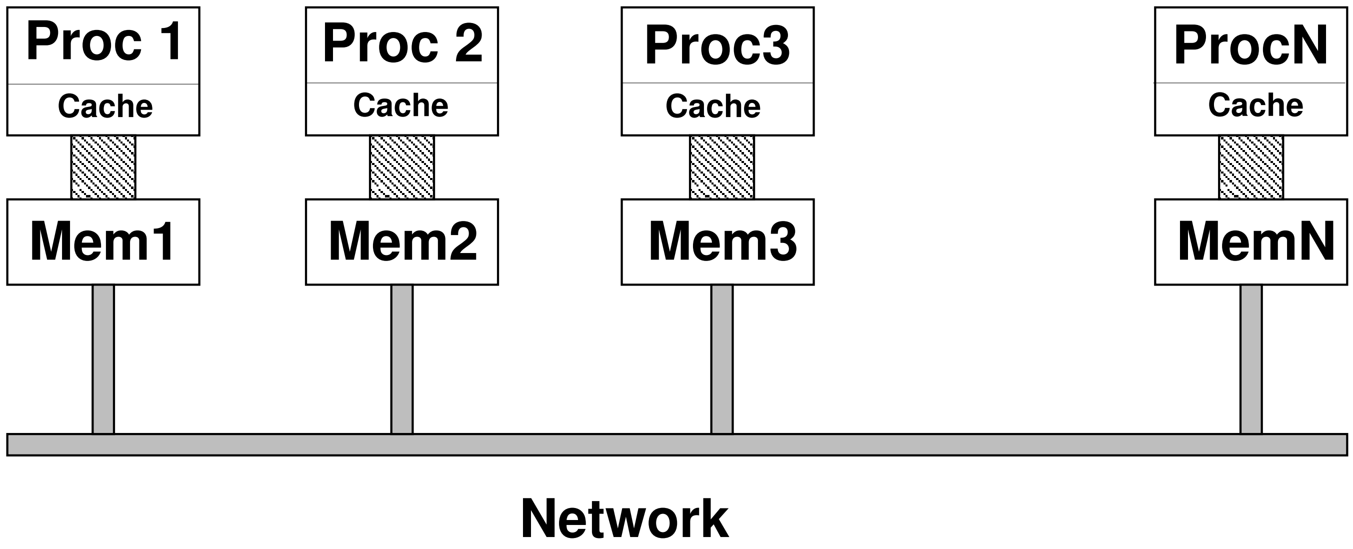
Distributed (non-uniform access - NUMA) versus centralized (uniform access - UMA) memory

Connectivity - single shared bus versus network with many different topologies

Fault tolerance

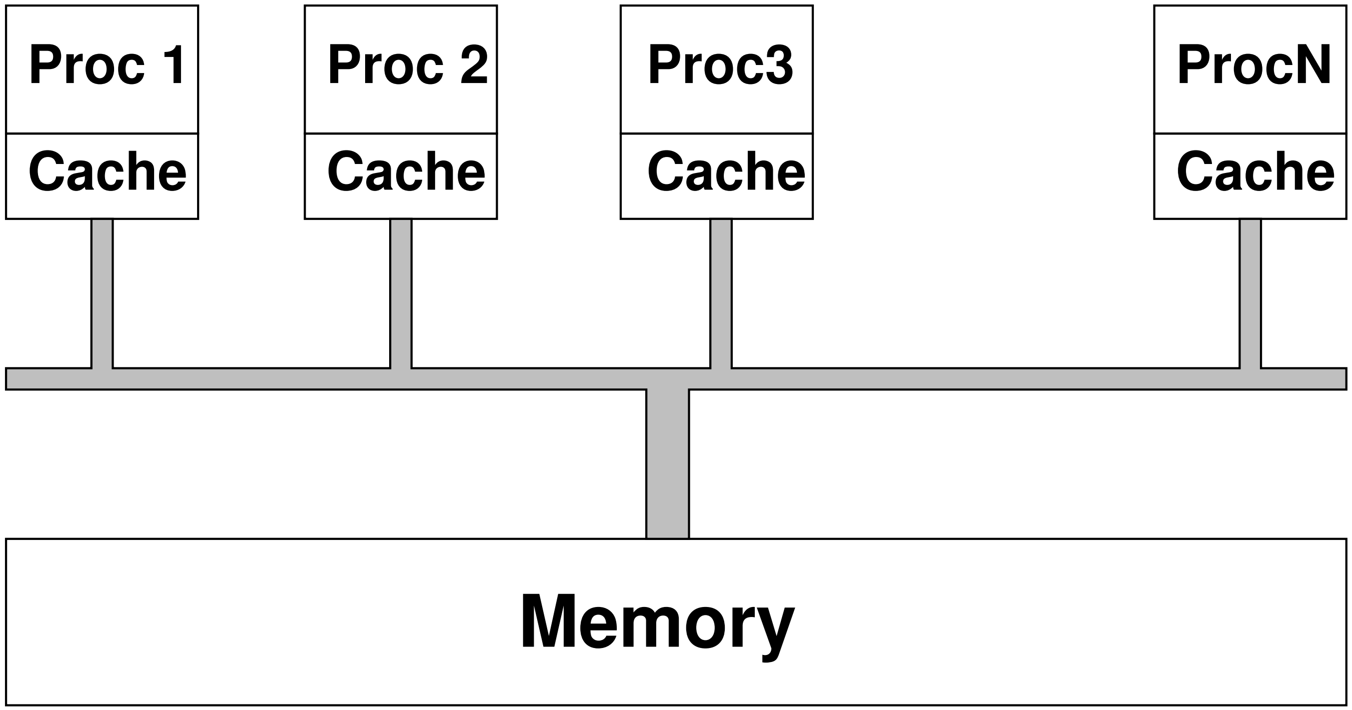
Distributed Memory Hardware (DM)

Parallel computing on networks of workstations



All commodity technology (including network), thus cheap

Shared Memory Hardware (SM)



Shared Memory Implementation

Coherence - ensuring that modifications made by a processor propagate to all (cached) copies of the data

- program order preserved
- Writes to the same location by different processors serialized
- defines the behavior of reads and writes to the same memory location

Synchronization - coordination mechanism

Consistency - defines when and in what order modifications are propagated to other processors

- defines the behavior of reads and writes with respect to accesses to other memory locations

Cache Coherence Protocols

Snooping - sharing status broadcast to all copies

Directory-based - sharing status of each block of data kept in one location

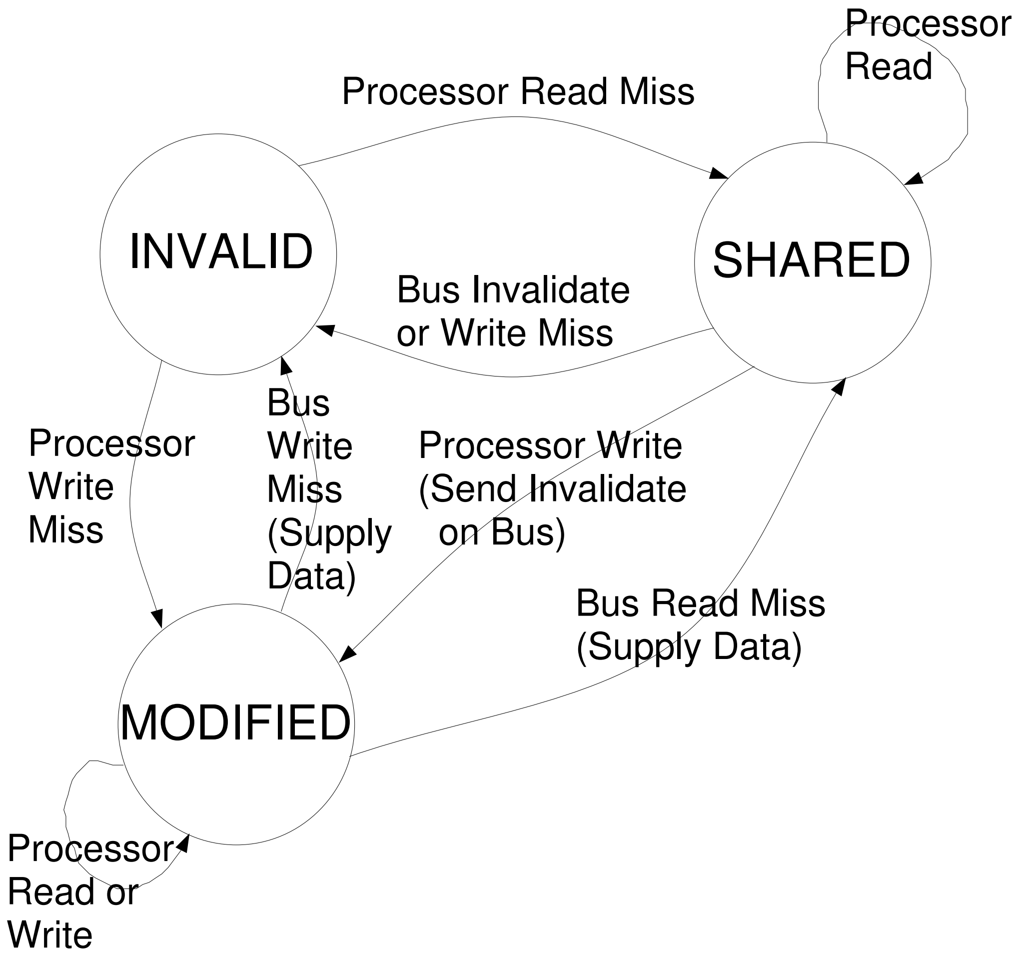
Snoopy-Based Protocols

Write Invalidate Protocol

Write Update Protocol

A Simple Invalidate-Based Snoopy Protocol

Finite State Machine with three states (MSI) -



Basic Hardware Mechanisms for Synchronization

Must provide the ability to atomically read and modify a memory location

- Test-and-set - atomic exchange
- Fetch-and-increment - returns value and atomically increments it
- Load-locked/store conditional - pair of instructions - deduce atomicity if second instruction returns correct value

Using ll/sc for Atomic Exchange

Swap the content of R4 with the memory location specified by R1

```
try:    mov   R3, R4      ; mov exchange value
        ll    R2, 0(R1)   ; load linked
        sc    R3, 0(R1)   ; store conditional
        beqz  R3, try     ; branch if store fails
        mov   R4, R2      ; put load value in R4
```

Implementing Locks Using Coherence

On a SPARC: *ldstub* moves an unsigned byte into the destination register, and rewrites the same byte in memory to all 1s.

```
_Lock:
        ldstub [%o0], %o1
        addcc %g0, %o1, %g0
        bne _Lock
        nop
fin:    jmpl %r15+8, %g0
        nop
_UnLock:
        st %g0, [%o0]
        jmpl %r15+8, %g0
        nop
```

Memory Consistency Models

When must a processor see a value that has been updated by another processor?

```
P1:    A = 0;                  P2:    B = 0;
       ...                            ...
       A = 1;                         B = 1;
L1:    if (B == 0) ...         L2:    if (A == 0) ...
```

Consistency Model Classification

Models vary along the following dimensions

- Local order - order of a processor's accesses as seen locally
- Global order - order of a single processor's accesses as seen by each of the other processors
- Interleaved order - order of interleaving of different processor's accesses on other processors

Sequential Consistency

"A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport 79]

In practice, this means that every write must be seen on all processors before any succeeding read or write can be issued

Why Shared Memory?

Easier to go sequential $\rightarrow$ parallel

Easier to utilize multi-paradigm platform

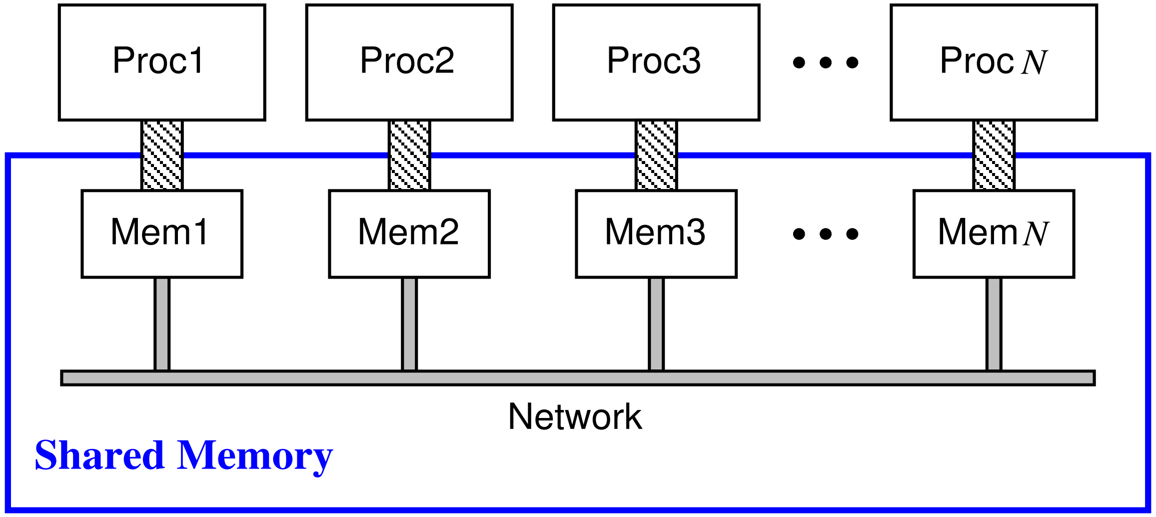
Sequential =
  single thread + single address space

Shared memory =
  multiple threads + single address space

Message passing =
  multiple threads + multiple address spaces

Software Distributed Shared Memory (DSM)

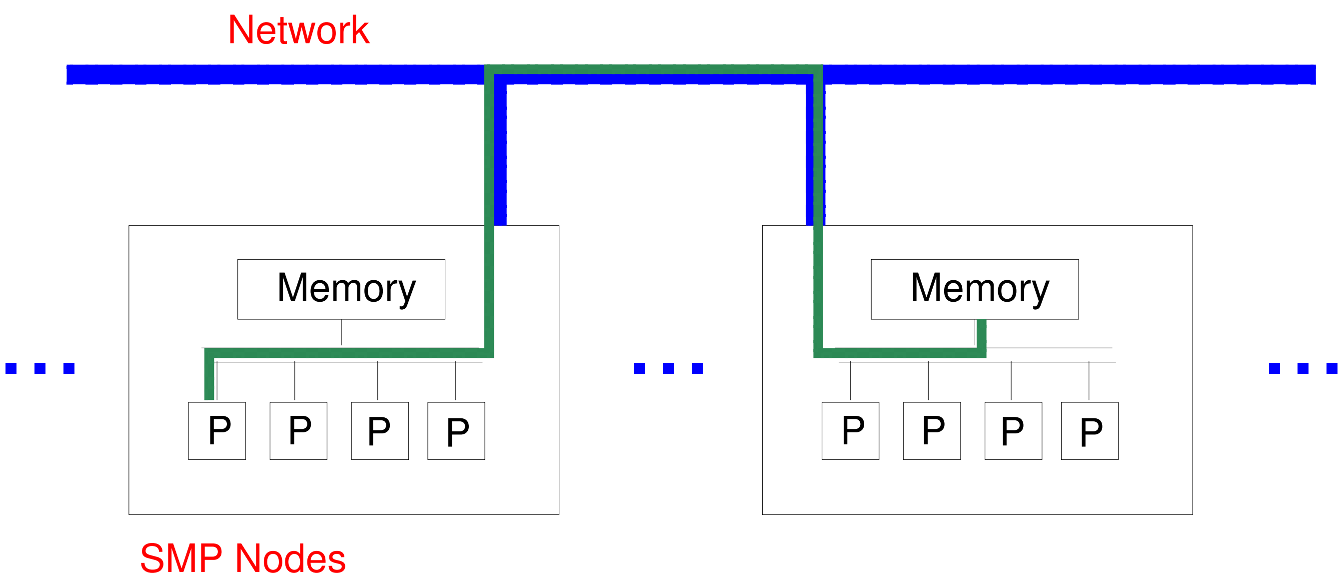
- Software provides shared memory image
- Global address space



Motivation:

Easier to program than message passing

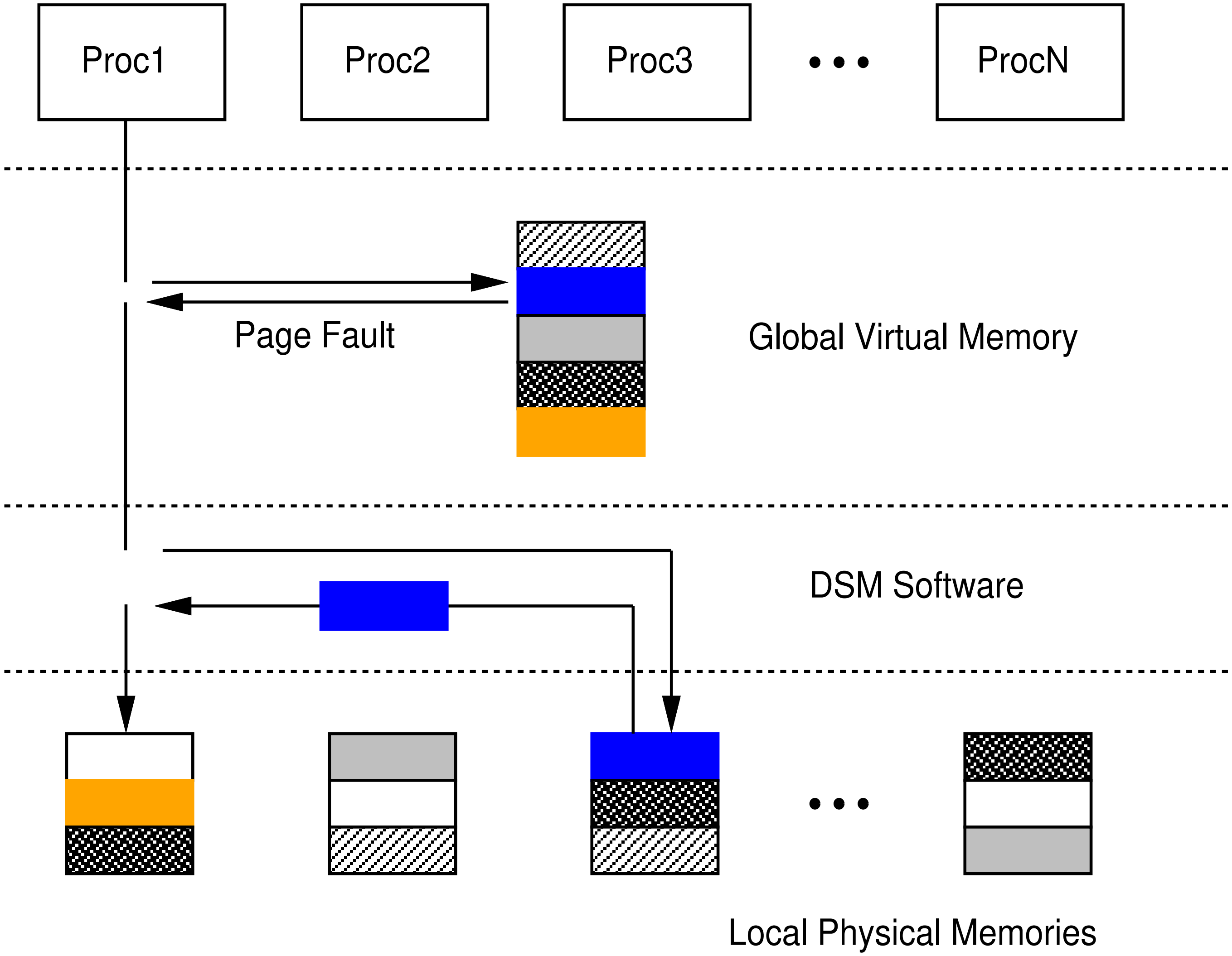
Cost-Effective Parallel Systems



Motivation: Need for a uniform programming model

Shared memory easier to program and exploits hardware coherence within node

Conventional DSM Implementation [Li 86]



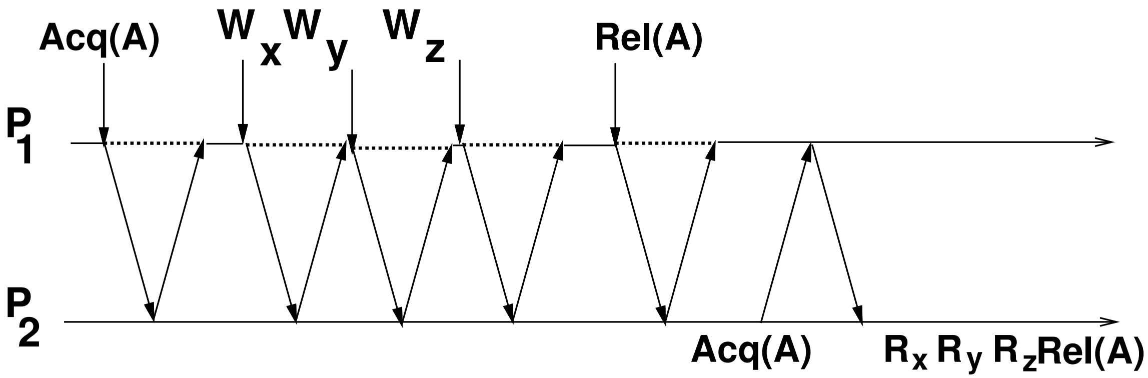
Page-based Consistency

Utilizes underlying virtual memory hardware to detect shared memory modifications

Advantages: No software overheads in the absence of sharing

Performance Problem: Sequential Consistency

Every write visible "immediately"

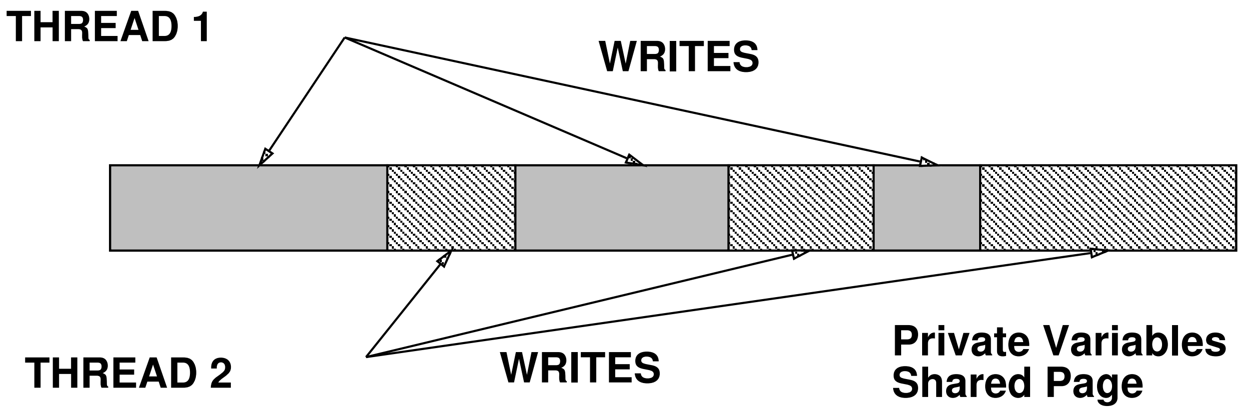


Problems:

- Number of messages
- Latency

Performance Problem: False Sharing

Pieces of the same page updated by different processors



Leads to “ping-pong” effect

Techniques

Virtual memory based software coherence

Multiple writer protocol

Delayed consistency with directories and home nodes

Reduce protocol synchrony in order to maximize hardware coherence exploitation

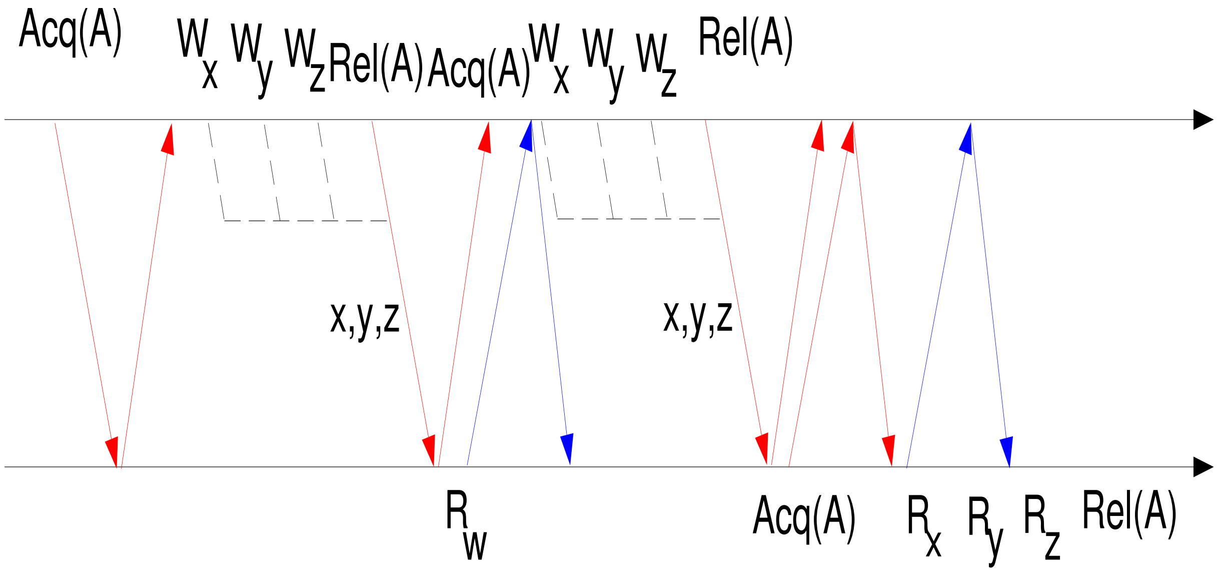
Release Consistency (RC) [Ghara. et al. 90]

Distinguish ordinary accesses and synchronization (acquire and release)

Need to guarantee consistency only after synchronization

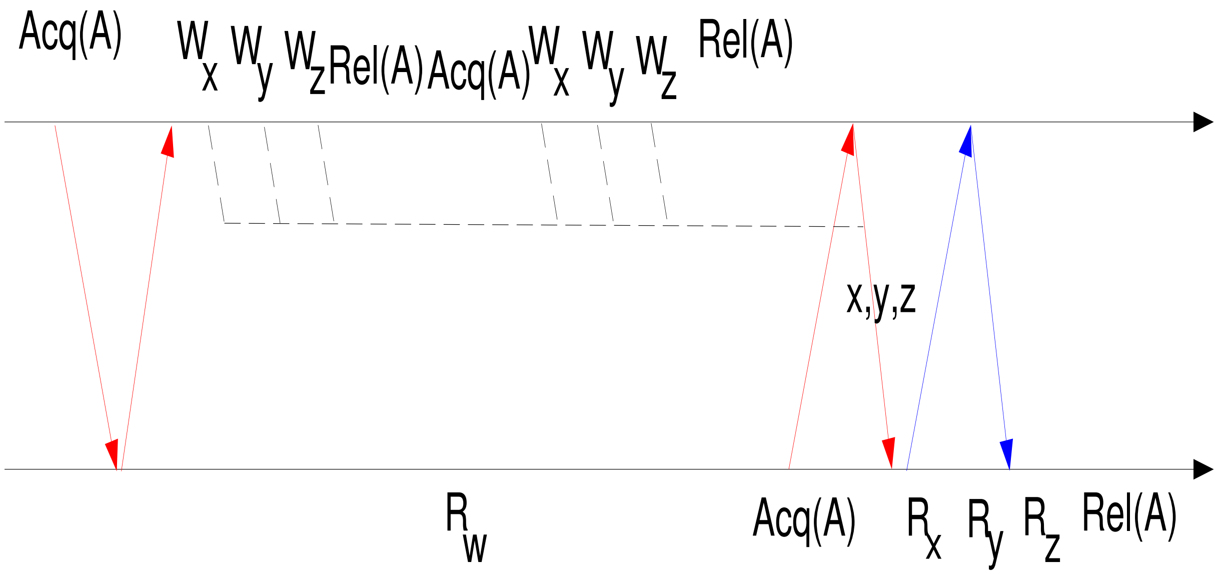
$\leadsto$ Enforce consistency only at synchronization points

Eager Release Consistency



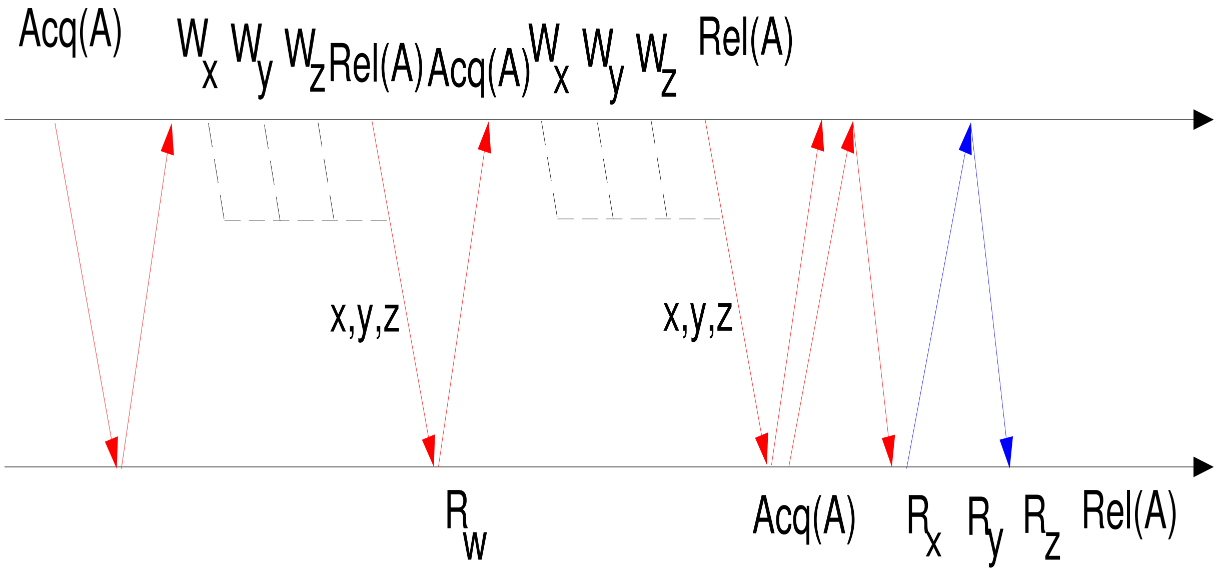
Push (and apply) modifications at release

Lazy Release Consistency [KCZ92]



Pull and apply modifications at acquire

Delayed Consistency



Push modifications at release

Apply modifications at acquire

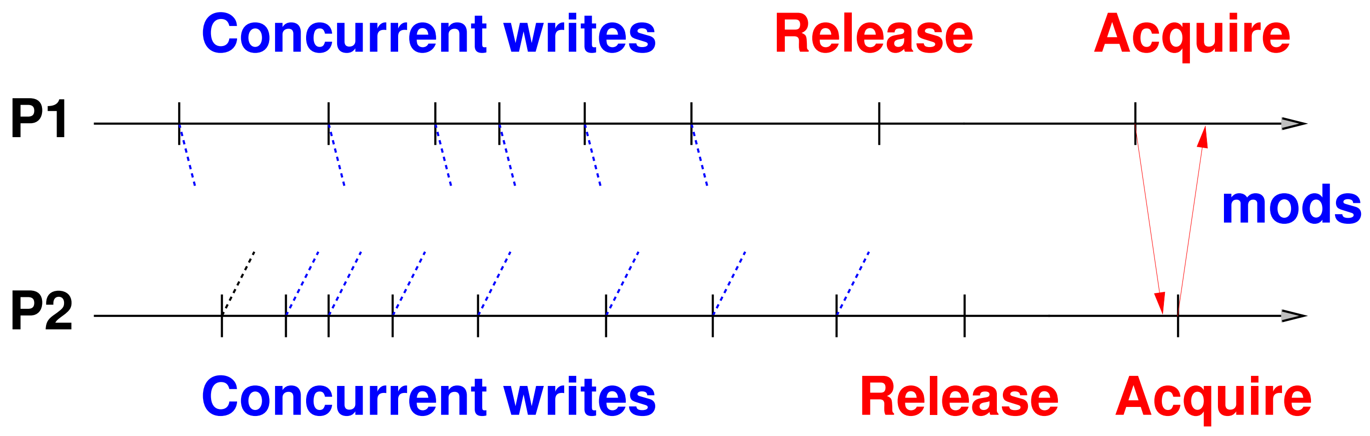
Multiple Writer Protocol [CBZ91]

Reduces false sharing overhead

Buffer writes until release

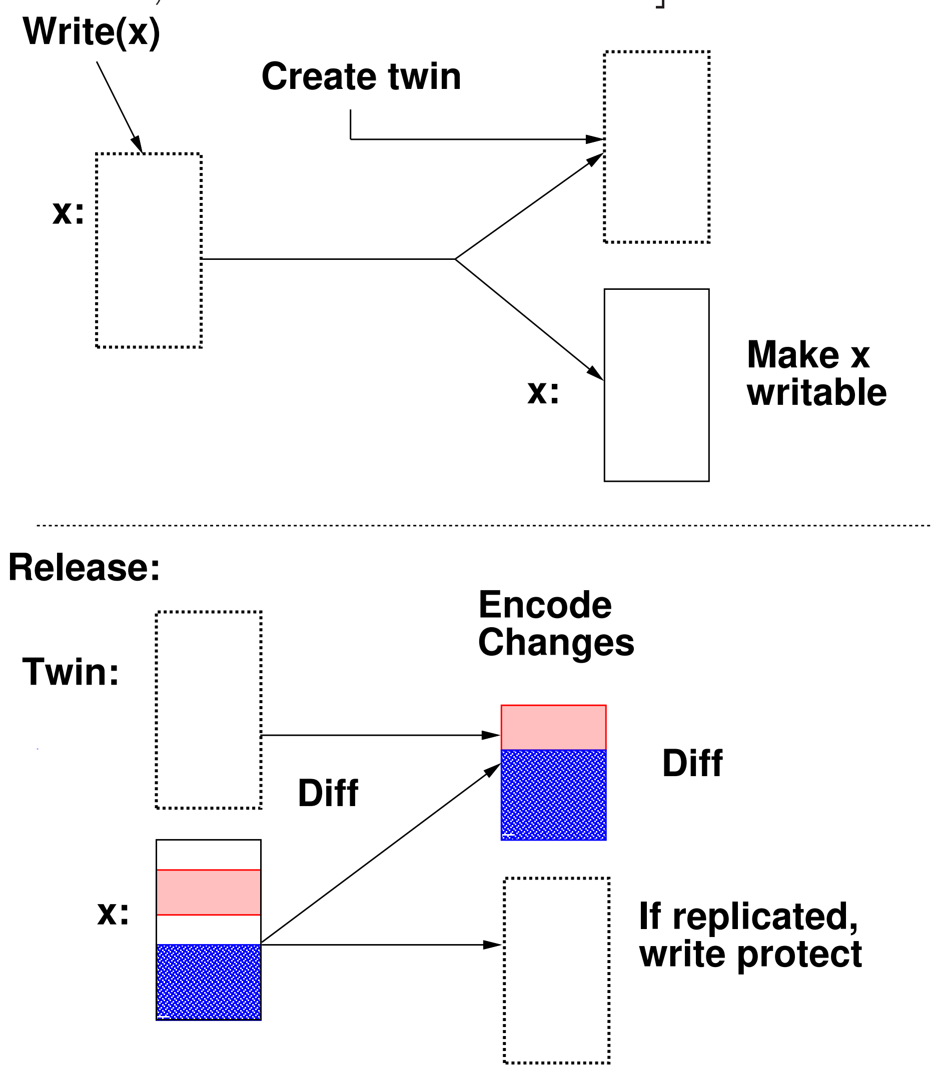
Create diffs and propagate to home node at release

Acquire lock $\rightarrow$ apply modifications



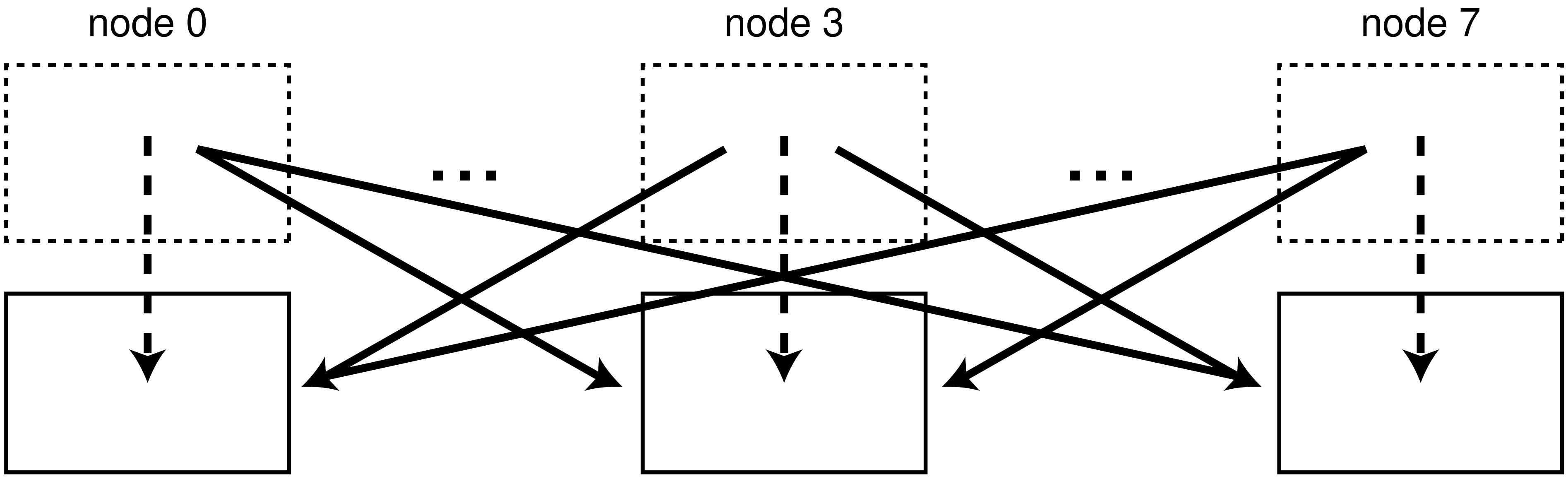
Outgoing Diffs

[Munin, TreadMarks, Home-based LRC]



Directory-Based Coherence

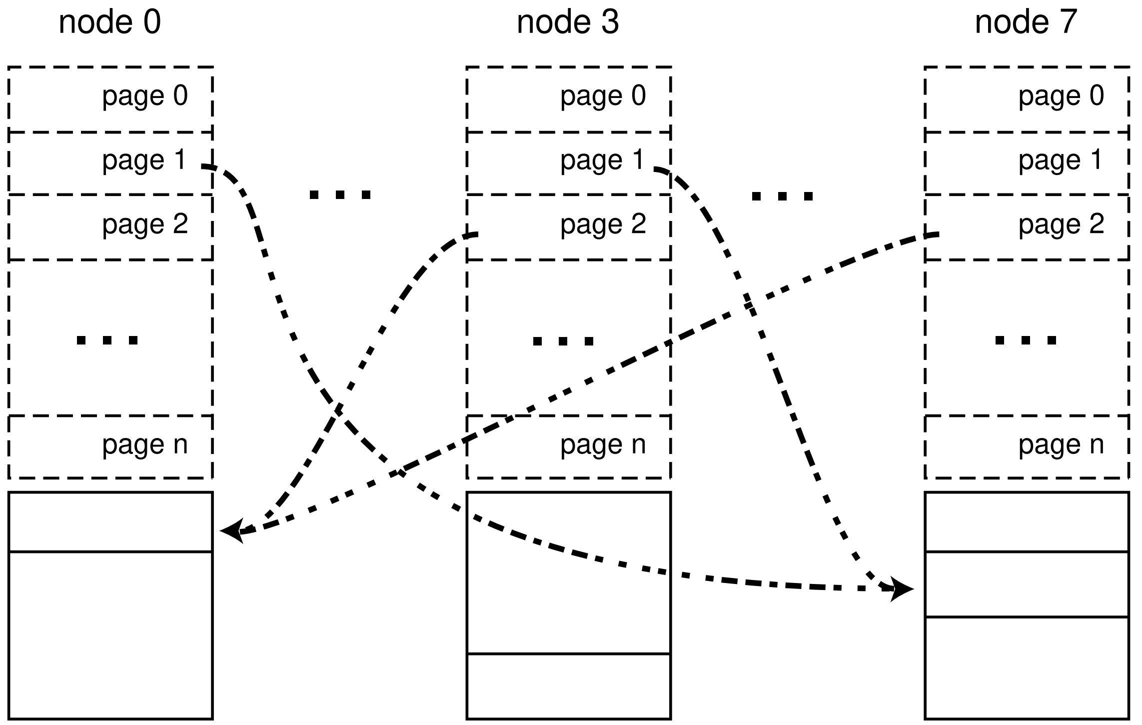
Global directory tracks per-SMP accesses to pages



Synchronization objects also employ broadcast

Home-Node Based Coherence

Data kept up-to-date at the home node



Exclusive and Migratory mode to avoid unnecessary communication

Protocol Actions

Page Faults

- Update global page state information
- Obtain current data on a read fault, create twin on a write fault

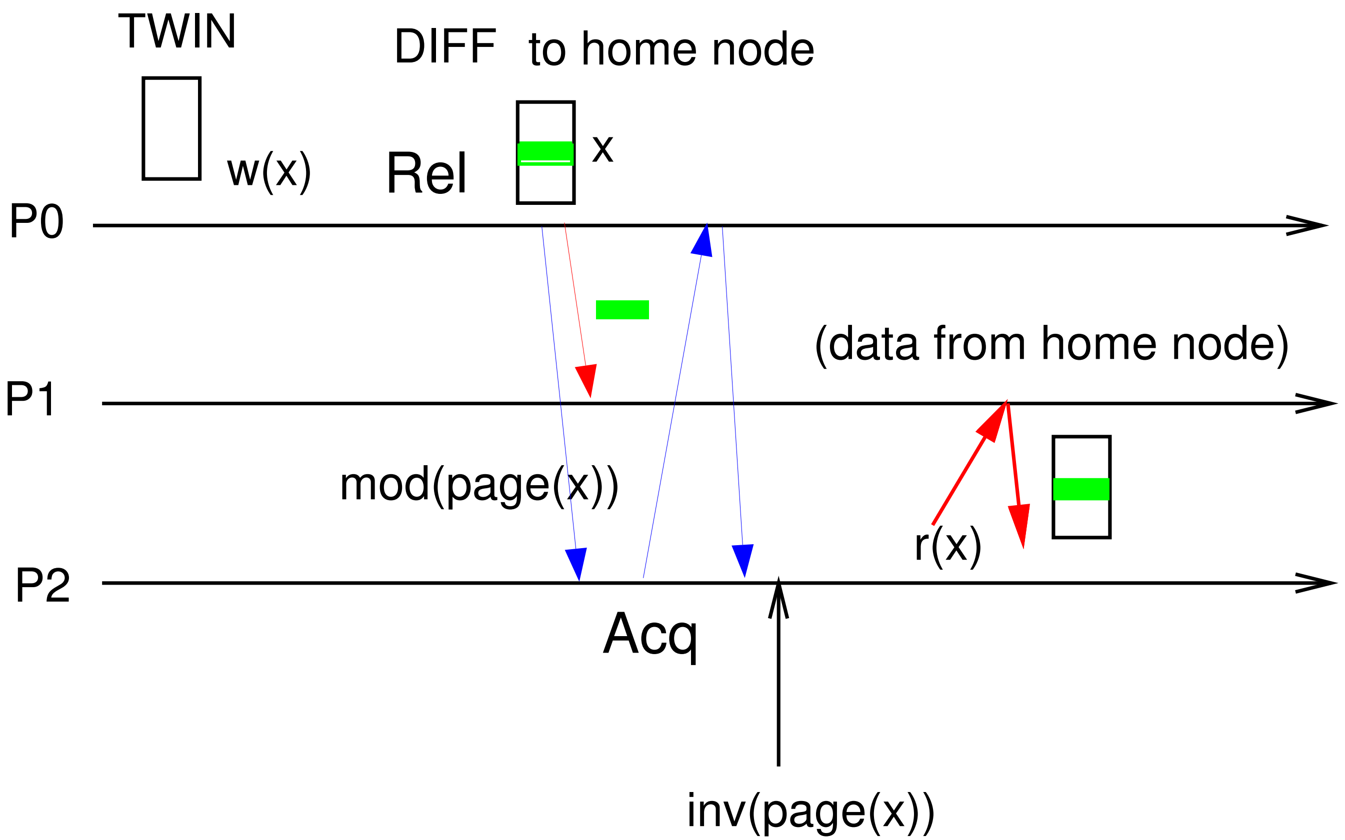
Releases

- Use outgoing diffs to send modifications to the home node
- Send write notices to sharing processors

Acquires

- Invalidate pages named by write notices
- Update global page state information

Protocol Actions for Cashmere-2L



SMP Design Optimizations

Each processor in a node shares same page frame

Coalesce global meta-data operations

Avoid redundant data fetches and home-node updates

Design protocol for asynchronous operation of processes

Hardware/Software Interaction

Software coherence operations performed for entire node

- Coalesce fetches of data to validate pages
- Coalesce updates of modified data to home node

Per-page timestamps within SMP to track remote-SMP events

- Last write notice received
- Last home-node update
- Last data fetch

Avoiding Redundant Updates

Reducing Synchronization Requirements within a Node

Need to provide mutually exclusive update capabilities with multiple writers on a node

Possible solutions

- Shootdown TLB entries of all concurrent copies
- Cashmere-2L - Incoming diffs

Incoming Diffs

Experimental Platform

32 233 MHz 21064A processors (8 AlphaServer 2100 4/233 SMPs)

- 16KByte icache, 16KByte dcache on-chip caches
- 1MByte board-level cache
- 256MBytes primary memory per SMP

DEC Memory Channel I network

- One-way latency: 5.2μsecs
- Bandwidth - 29 MBytes/sec per link, 60 MBytes/sec aggregate

TreadMarks

State-of-the-art Software Distributed Shared Memory System

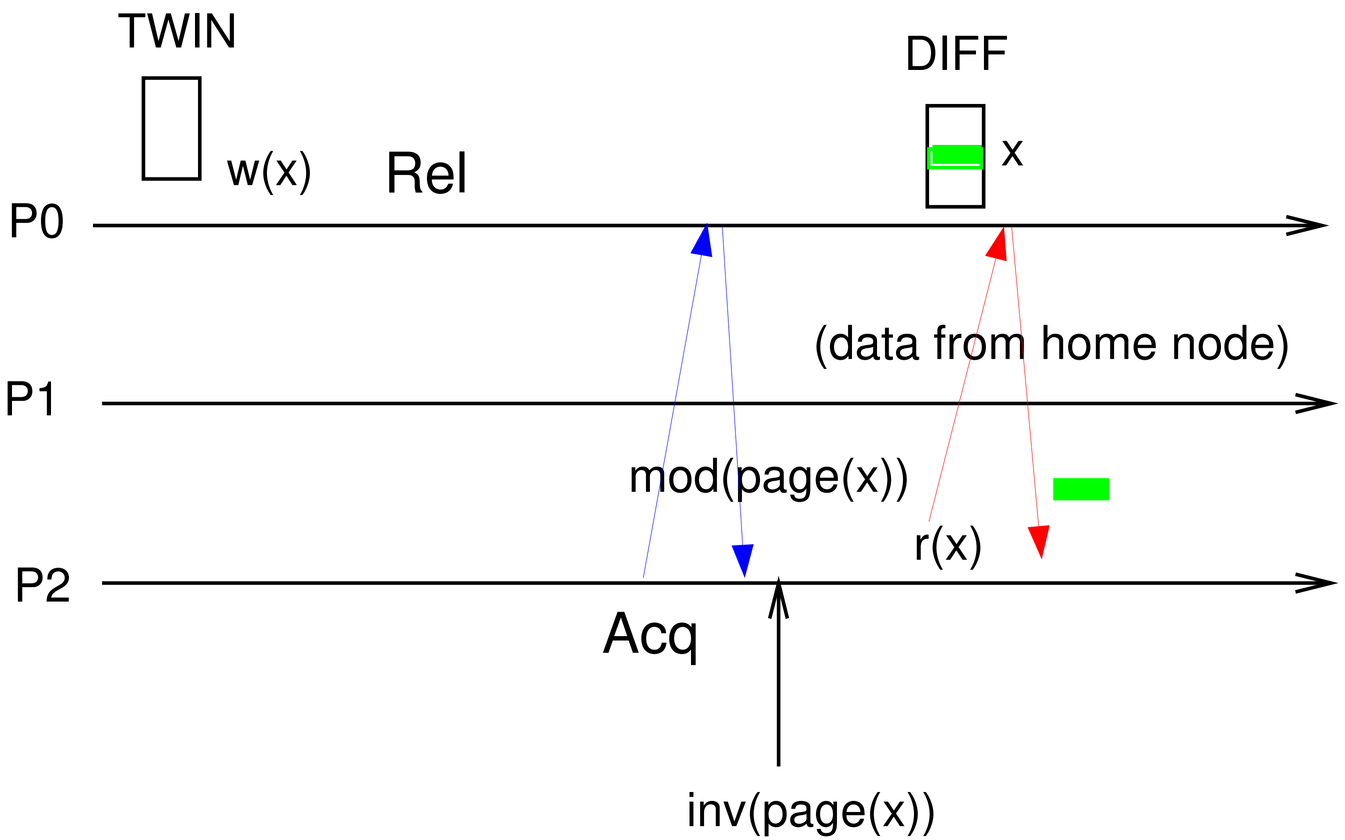
Page-based

Distributed vector timestamps

Lazy release consistency (LRC - [Keleher et al. 92])

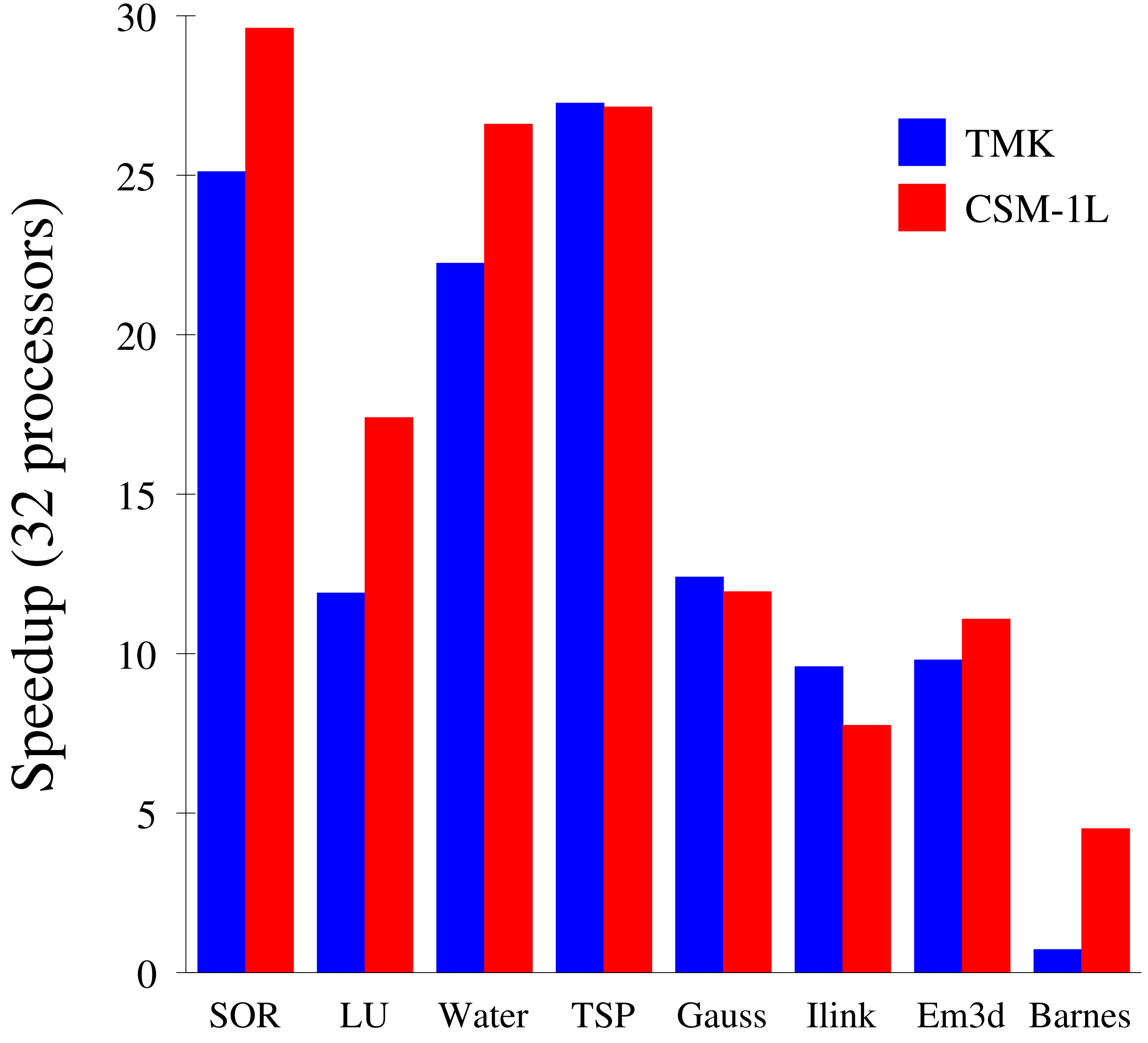
Multiple writer protocol [Carter et al. 91]

Protocol Actions for TreadMarks

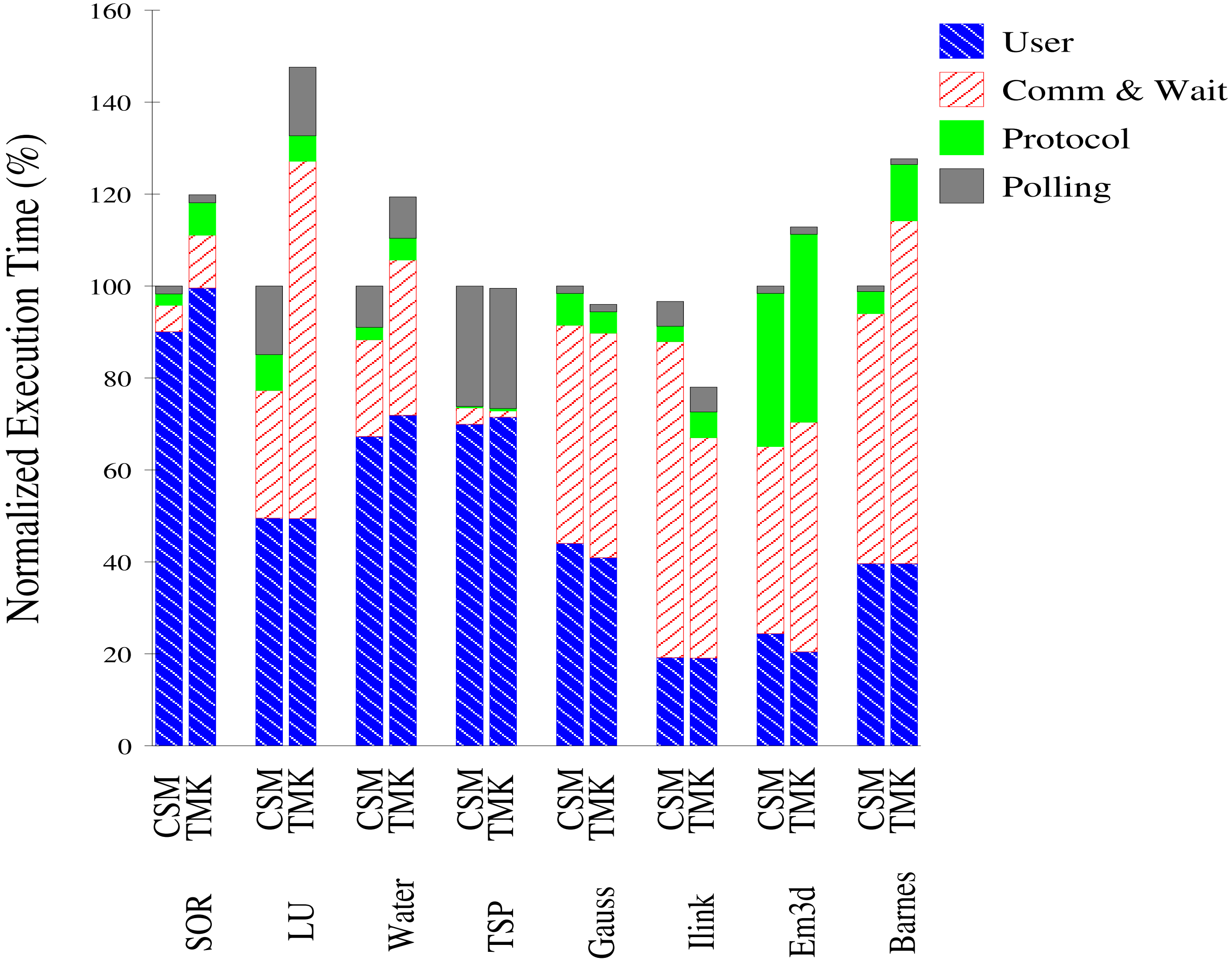


$\rightsquigarrow$Must communicate all write notices to all processors

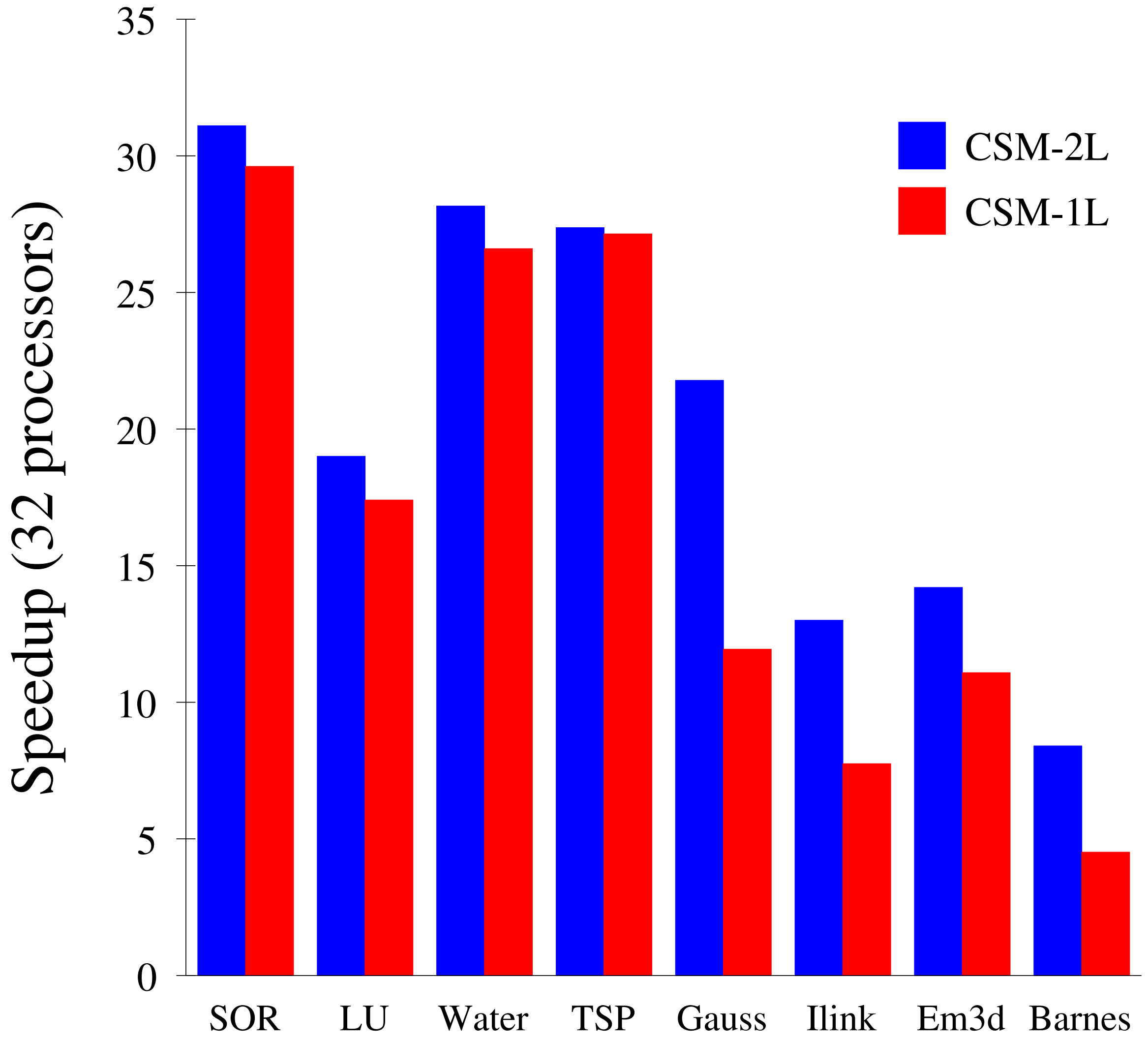
Performance: TreadMarks Vs. Cashmere-1L



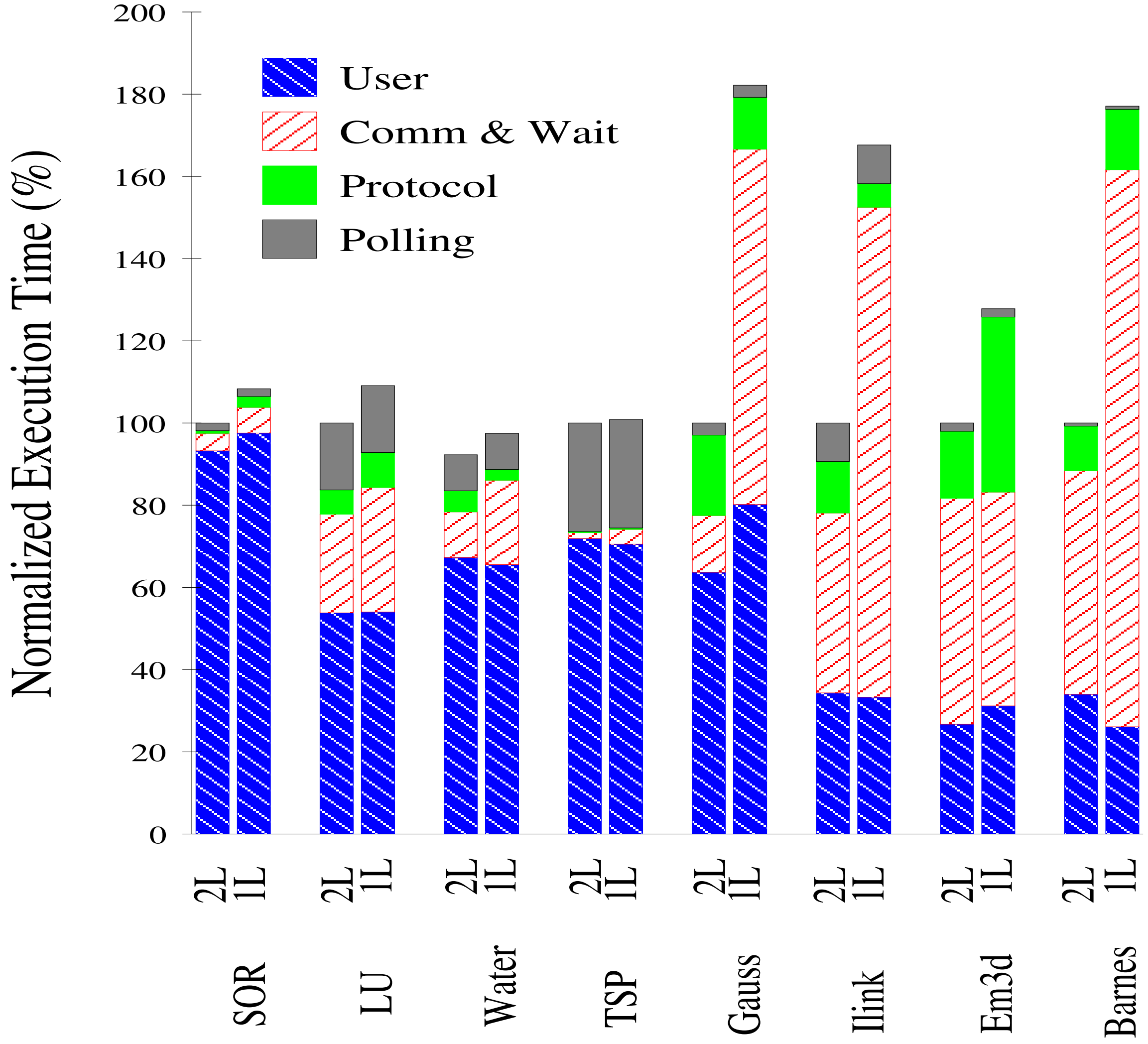
Execution Breakdown: TreadMarks Vs. Cashmere-1L

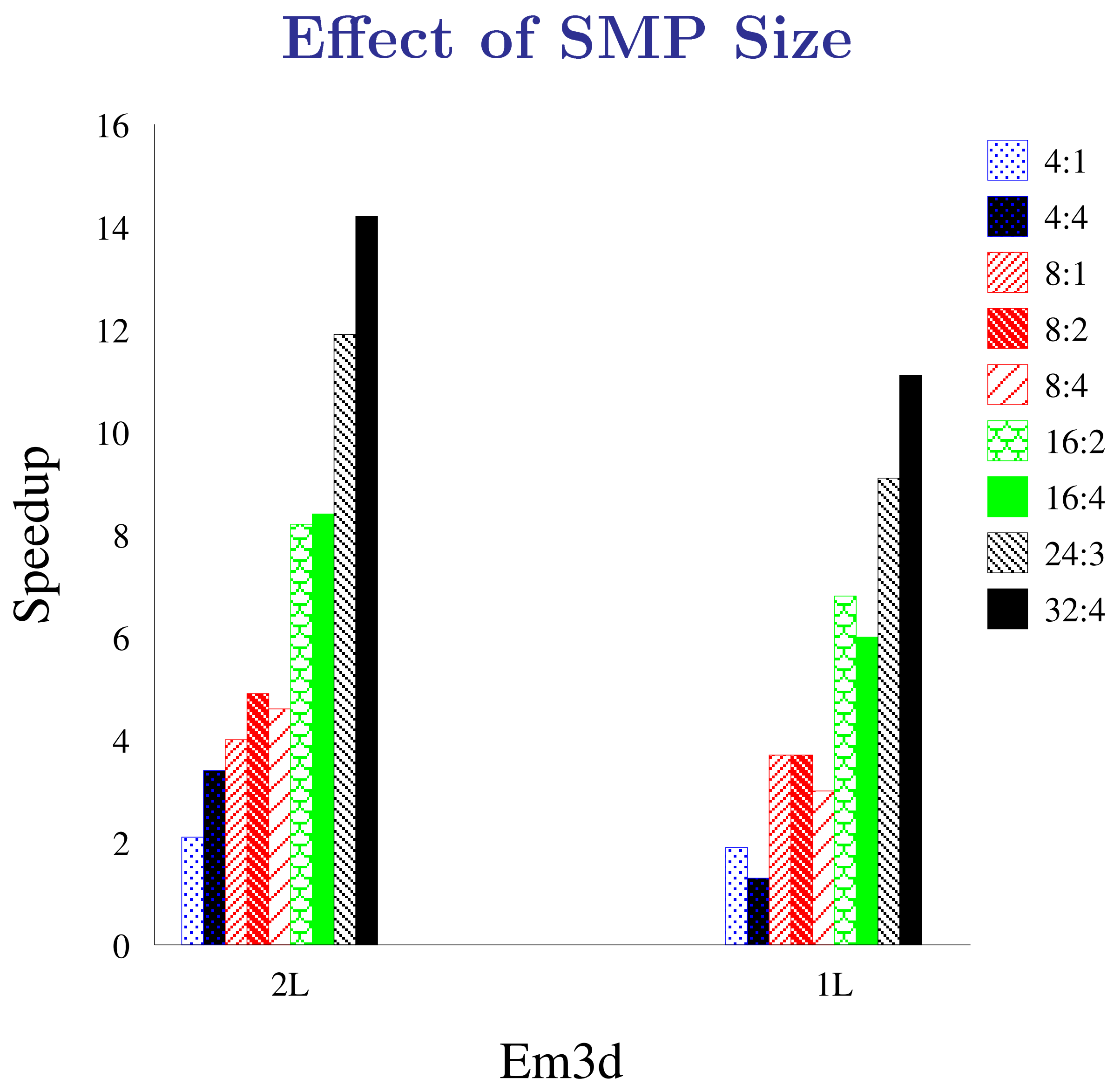


Performance: Cashmere-2L Vs. Cashmere-1L

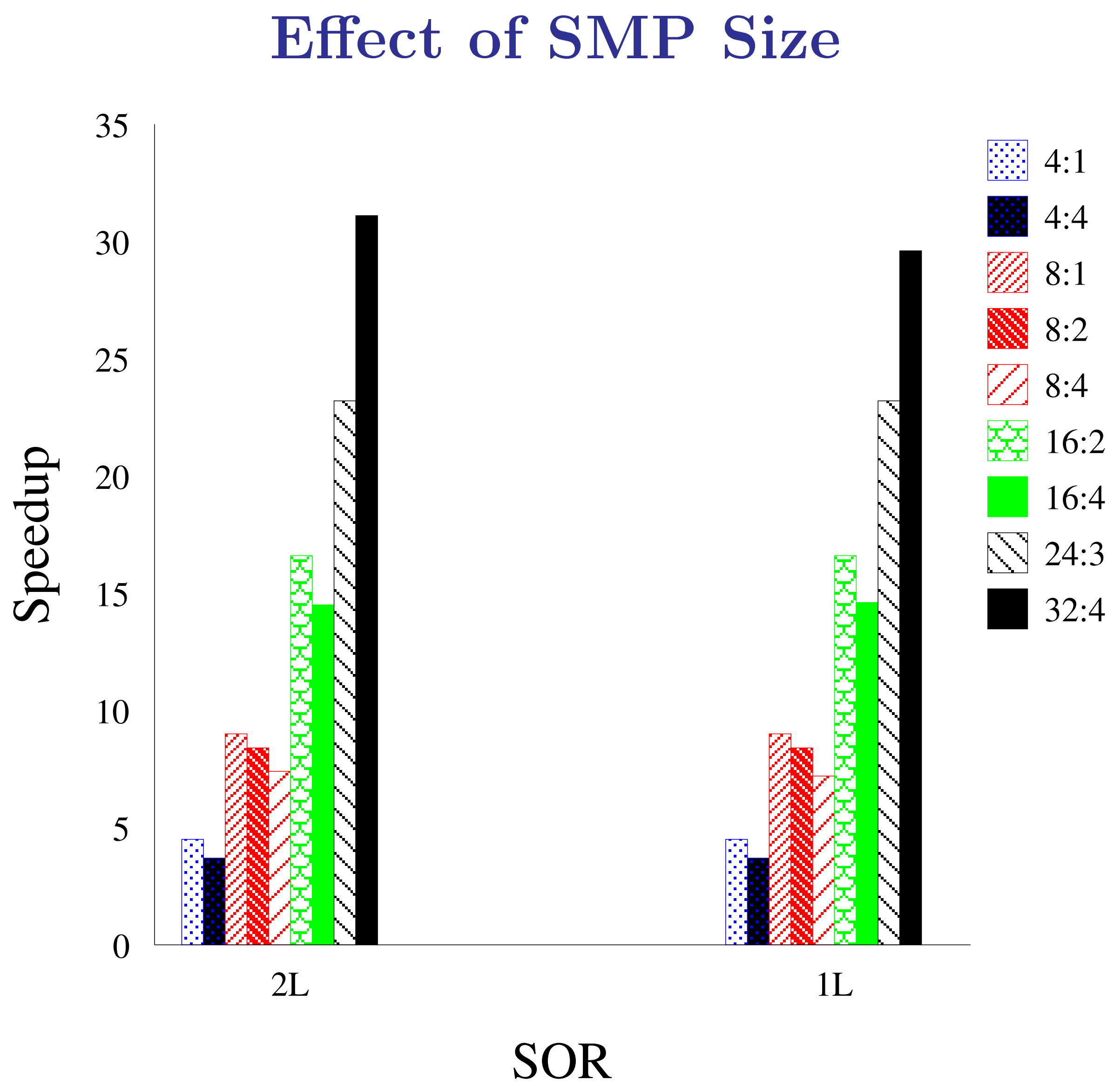


Execution Breakdown: 2L Vs. 1L

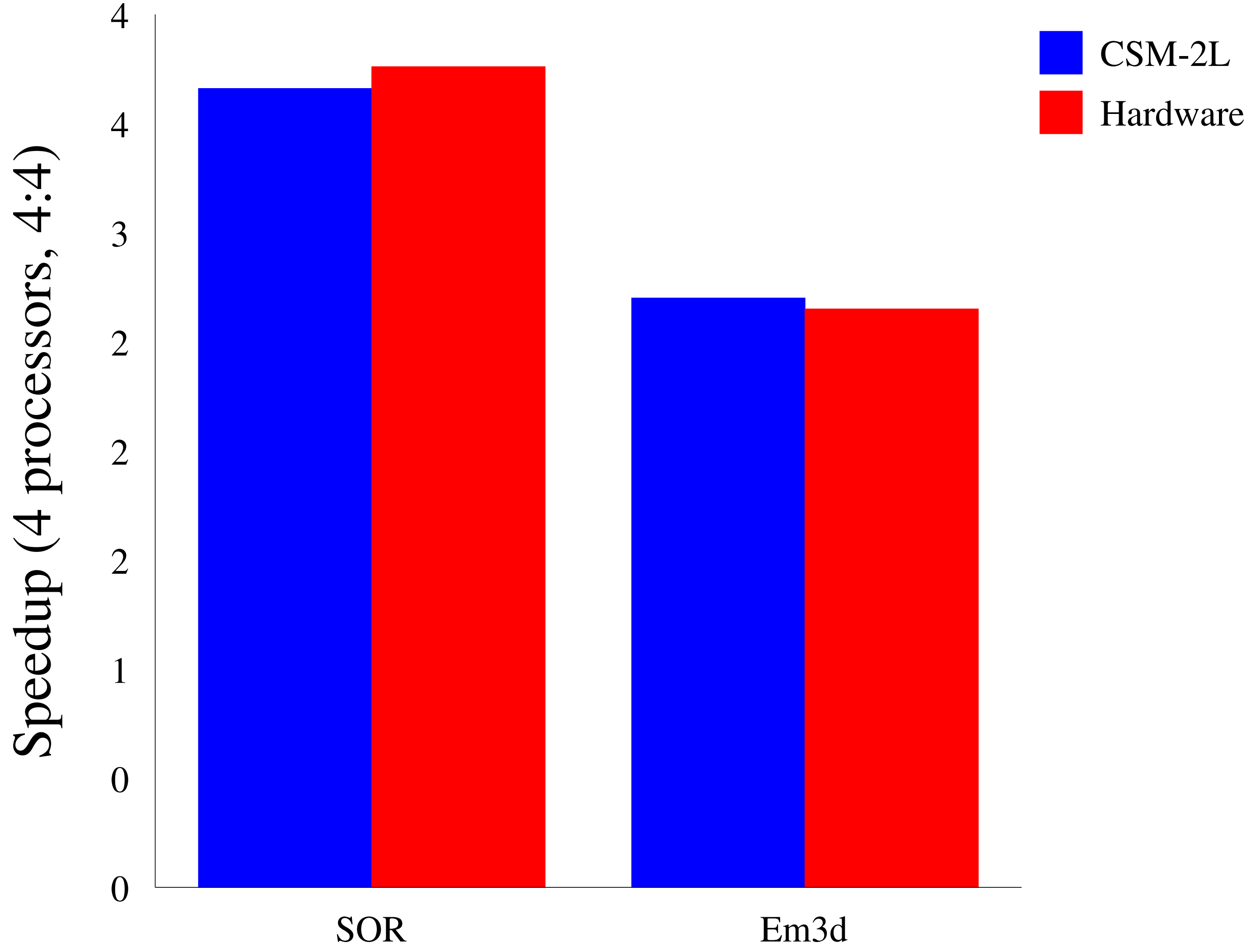




Effect of SMP Size
Speedup
16
14
12
10
8
6
4
2
0
2L
1L
Em3d
4:1
4:4
8:1
8:2
8:4
16:2
16:4
24:3
32:4


Effect of SMP Size
Speedup
35
30
25
20
15
10
5
0
2L
1L
SOR
4:1
4:4
8:1
8:2
8:4
16:2
16:4
24:3
32:4

Cashmere-2L Vs. Hardware



Summary and Conclusions

Parallel programming models - shared memory and message passing

MIMD implementation issues

Hardware shared memory implementation

Software shared memory implementation

Software shared memory can be both cost-effective and efficient

Two-level software design exploits hardware coherence within SMPs