Chapter 5: Thread-Level Parallelism – Part 1

Introduction

     What is a parallel or multiprocessor system?

     Why parallel architecture?

     Performance potential

     Flynn classification

Communication models

Architectures

Centralized shared-memory

Distributed shared-memory

Parallel programming

Synchronization

Memory consistency models

What is a parallel or multiprocessor system?

Multiple processor units working together to solve the same problem

Key architectural issue: Communication model

Why parallel architectures?

Absolute performance

Technology and architecture trends

    Dennard scaling, ILP wall, Moore's law

$\Rightarrow$ Multicore chips

    Connect multicore together for even more parallelism

Performance Potential

Amdahl's Law is pessimistic

 Let s be the serial part

 Let p be the part that can be parallelized n ways

```
Serial:          SSPPPPPP
6 processors:    SSP
                   P
                   P
                   P
                   P
                   P
```

 Speedup = 8/3 = 2.67

 $T(n) = \dfrac{1}{s+p/n}$

 As $n \to \infty$, $T(n) \to \dfrac{1}{s}$

Pessimistic

Performance Potential (Cont.)

Gustafson's Corollary

    Amdahl's law holds if run same problem size on larger machines

    But in practice, we run larger problems and "wait" the same time

Performance Potential (Cont.)

Gustafson's Corollary (Cont.)

    Assume for larger problem sizes

        Serial time fixed (at s)

        Parallel time proportional to problem size (truth more complicated)

```
Old Serial:     SSPPPPPP
6 processors:   SSPPPPPP
                  PPPPPP
                  PPPPPP
                  PPPPPP
                  PPPPPP
                  PPPPPP
Hypothetical Serial:
   SSPPPPPP PPPPP PPPPP PPPPP PPPPP PPPPP
```

    Speedup = (8+5*6)/8 = 4.75

    $T'(n) = s + n*p; T'(\infty) \rightarrow \infty$!!!!

How does your algorithm "scale up"?

Flynn classification

Single-Instruction Single-Data (SISD)

Single-Instruction Multiple-Data (SIMD)

Multiple-Instruction Single-Data (MISD)

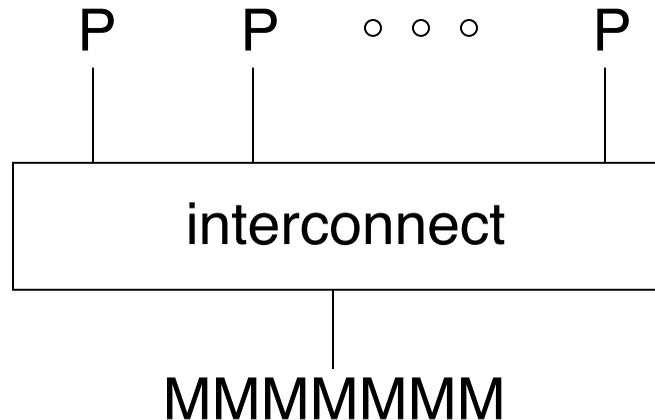Multiple-Instruction Multiple-Data (MIMD)

Communication models

Shared-memory

Message passing

Data parallel

Communication Models: Shared-Memory

P   P   ○ ○ ○   P

interconnect

MMMMMMM

Each node a processor that runs a process

One shared memory
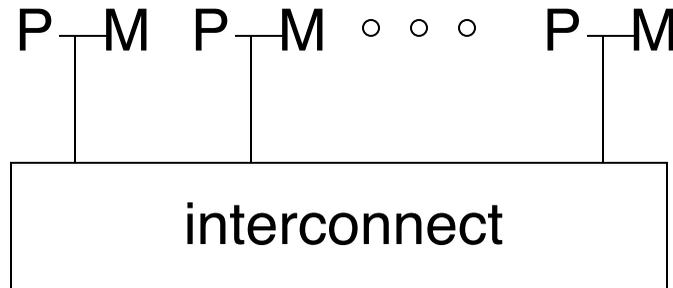
    Accessible by any processor

    The same address on two different processors refers to the same datum

Therefore, write and read memory to

    Store and recall data

    Communicate, Synchronize (coordinate)

Communication Models: Message Passing

P—M P—M ○ ○ ○   P—M

```
┌──────────────────────────────┐
│          interconnect         │
└──────────────────────────────┘
```
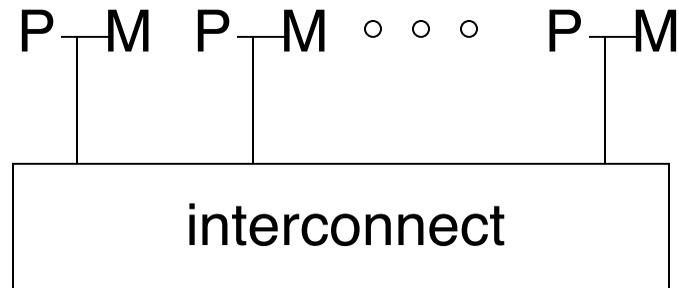
Each node a computer

    Processor – runs its own program (like SM)

    Memory – local to that node, unrelated to other memory

Add messages for internode communication, send and receive like
    mail

Communication Models: Data Parallel

$$P - M \quad P - M \quad \circ \circ \circ \quad P - M$$

interconnect

Virtual processor per datum

Write sequential programs with "conceptual PC" and let parallelism
   be within the data (e.g., matrices)

C = A + B

Typically SIMD architecture, but MIMD can be as effective

Architectures

All mechanisms can usually be synthesized by all hardware

Key: which communication model does hardware support best?

Virtually all small-scale systems, multicores are shared-memory

Which is Best Communication Model to Support?

Shared-memory

    Used in small-scale systems

    Easier to program for dynamic data structures

    Lower overhead communication for small data

    Implicit movement of data with caching
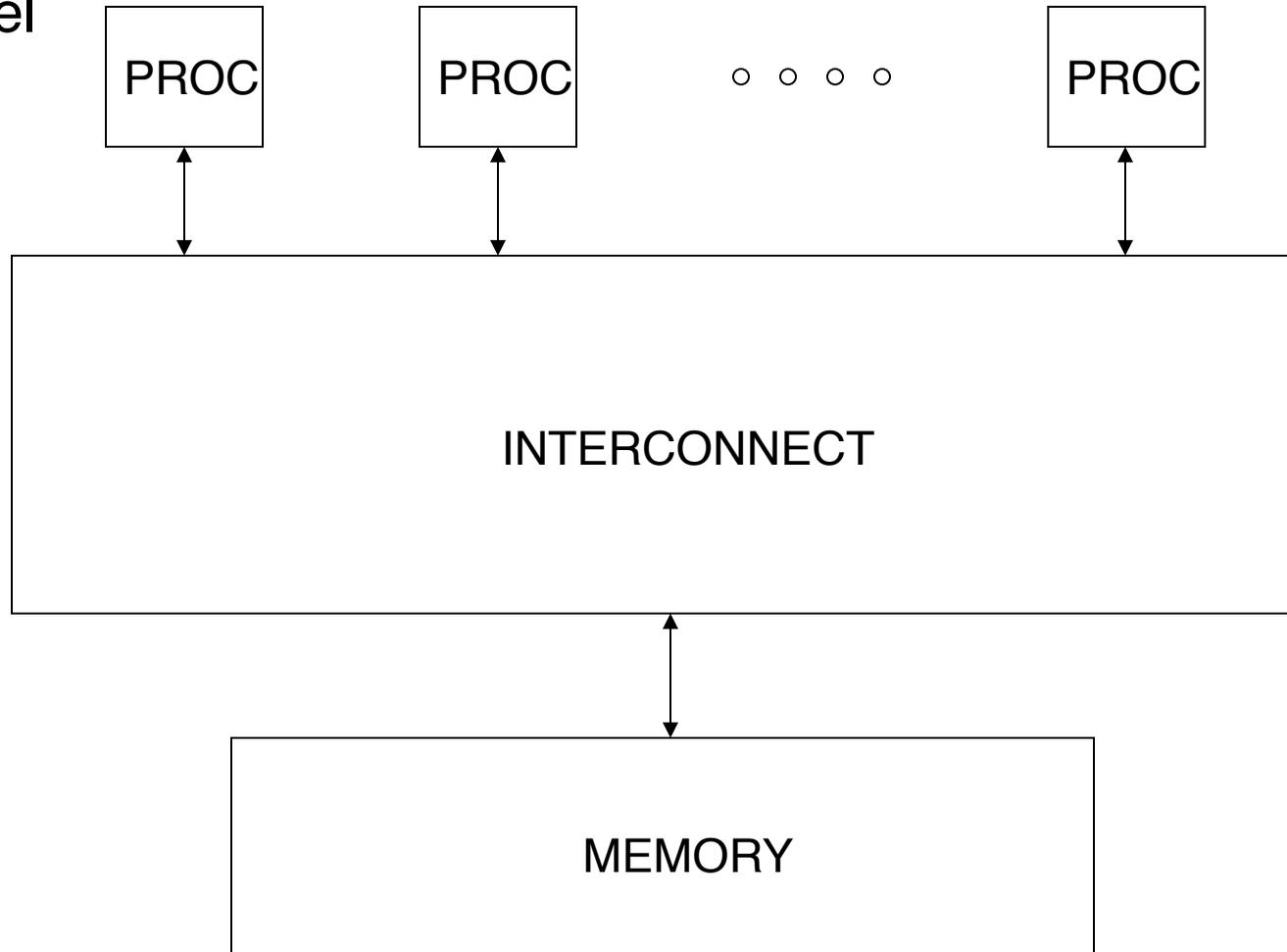
    Hard to build?

Message-passing

    Communication explicit harder to program?

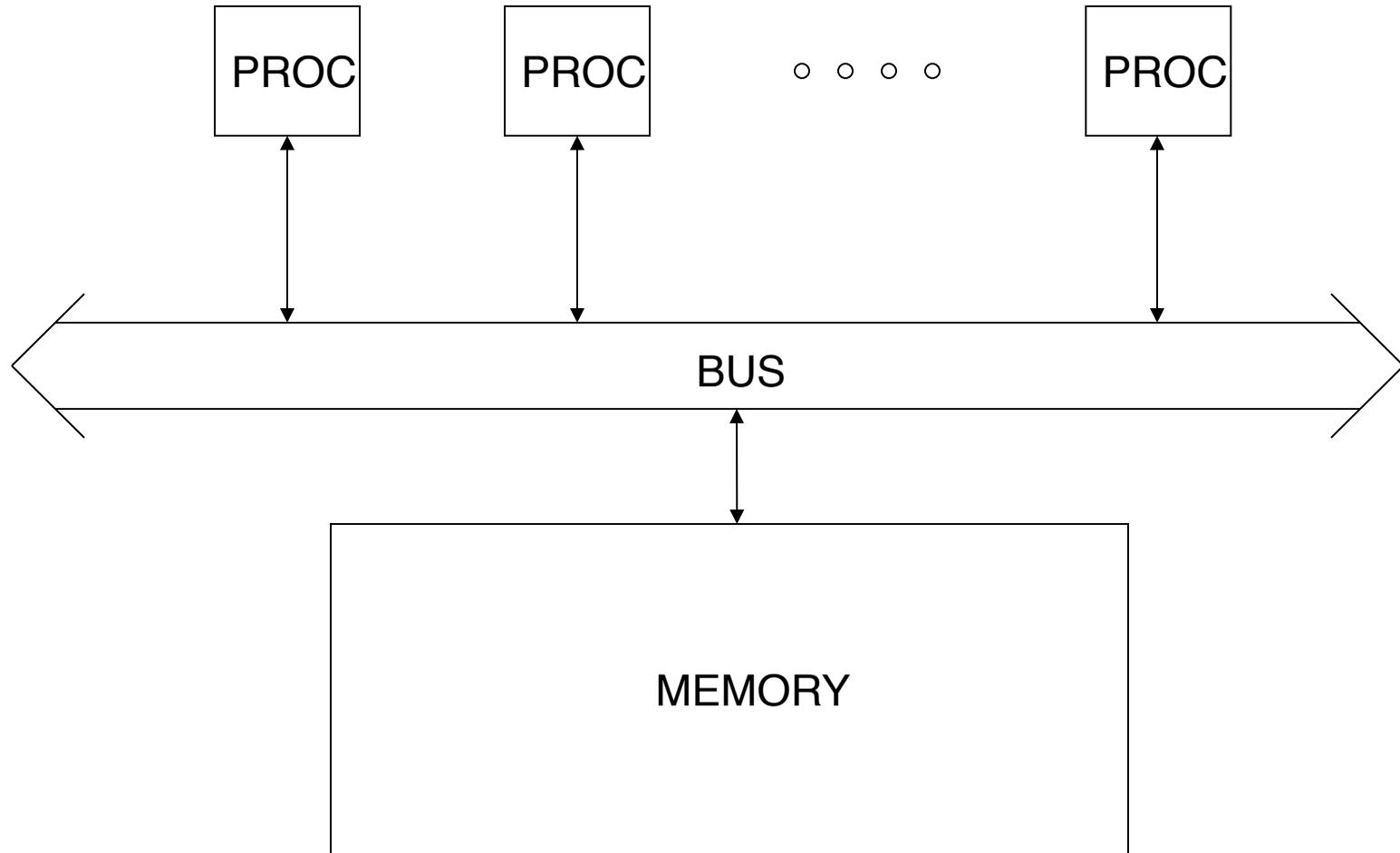   Larger overheads in communication OS intervention?

    Easier to build?

Shared-Memory Architecture

The model

For now, assume interconnect is a bus – *centralized architecture*
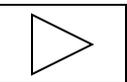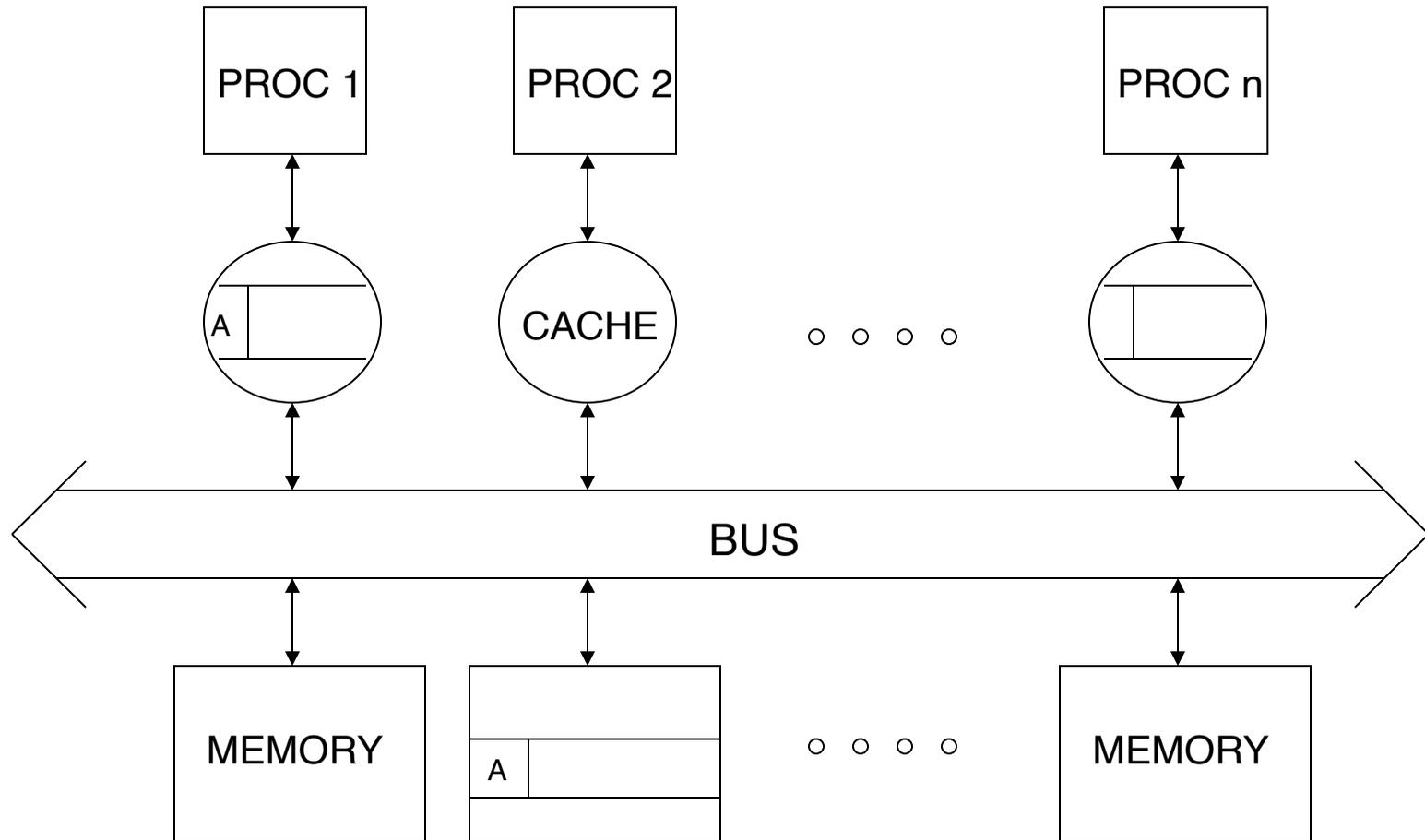
Centralized Shared-Memory Architecture

For higher bandwidth (throughput)

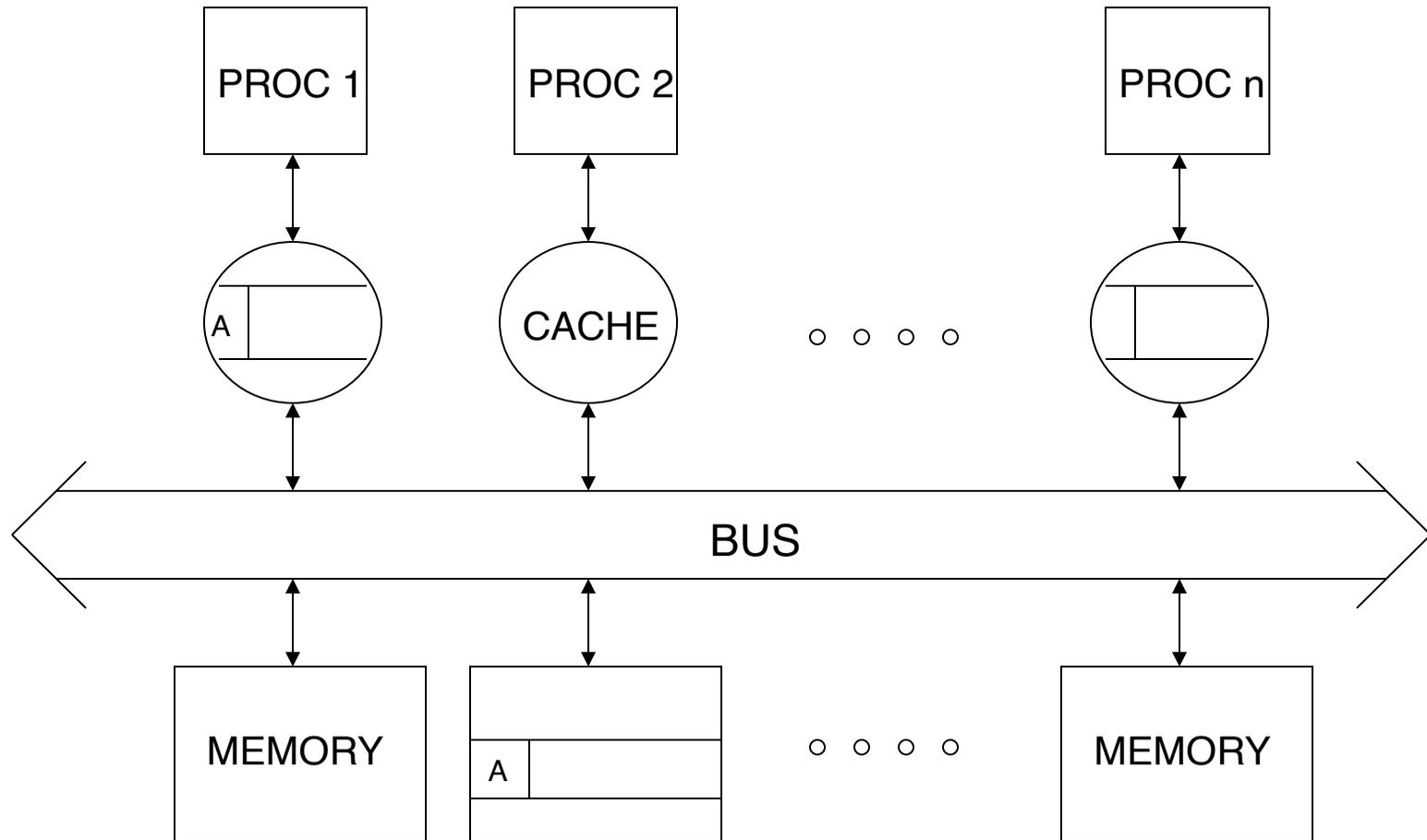For lower latency

Problem?

Cache Coherence Problem

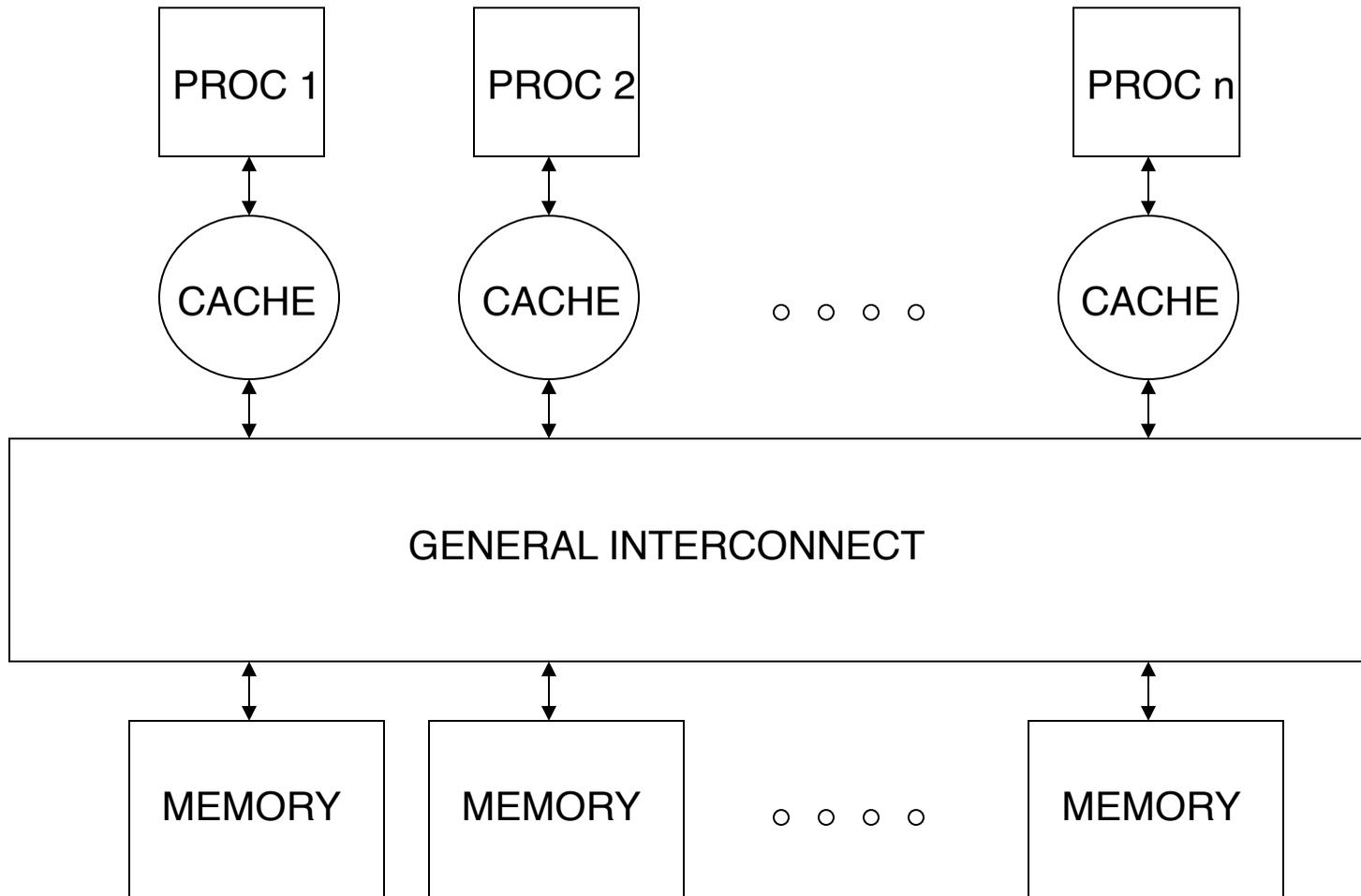# *Cache Coherence Solutions*

Snooping

Problem with centralized architecture

MSI Coherence Protocol

MSI Coherence Protocol
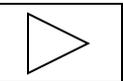
Distributed Shared-Memory (DSM) Architecture

Use a higher bandwidth interconnection network

```
  ┌────────┐   ┌────────┐                    ┌────────┐
  │ PROC 1 │   │ PROC 2 │                    │ PROC n │
  └────────┘   └────────┘                    └────────┘
      ↕            ↕                              ↕
   ( CACHE )    ( CACHE )      o  o  o  o      ( CACHE )
      ↕            ↕                              ↕
  ┌──────────────────────────────────────────────────────┐
  │              GENERAL INTERCONNECT                     │
  └──────────────────────────────────────────────────────┘
      ↕            ↕                              ↕
  ┌────────┐   ┌────────┐                    ┌────────┐
  │ MEMORY │   │ MEMORY │    o  o  o  o      │ MEMORY │
  └────────┘   └────────┘                    └────────┘
```

Uniform memory access architecture (UMA)

Distributed Shared-Memory (DSM) - Cont.

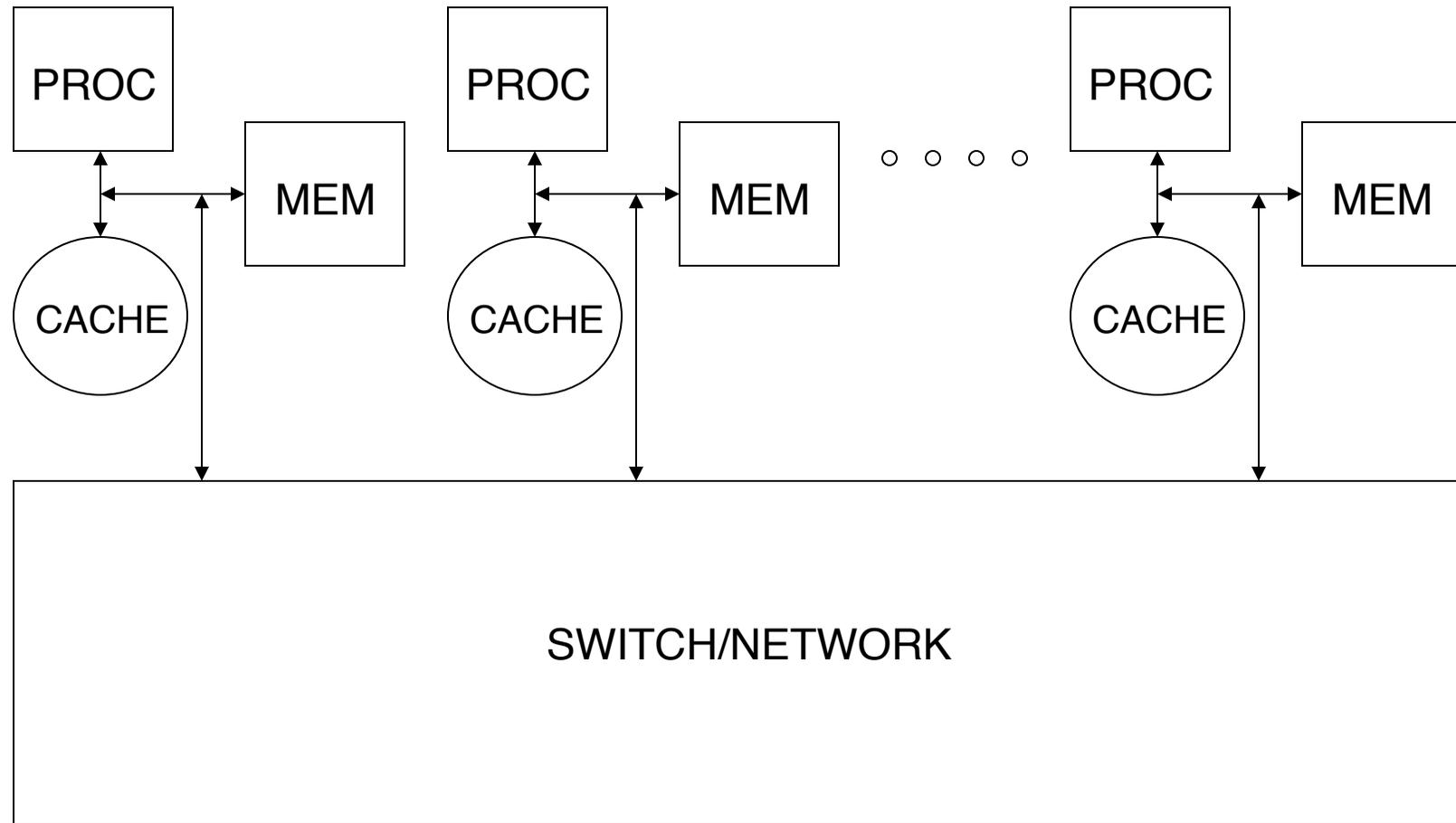For lower latency: Non-Uniform Memory Access architecture (NUMA)

Non-Bus Interconnection Networks

Example interconnection networks

Distributed Shared-Memory - Coherence Problem

Directory scheme

Level of indirection!

Parallel Programming Example

Add two matrices: C = A + B

Sequential Program

```
main(argc, argv)
int argc; char *argv;
{
   Read(A);
   Read(B);
   for (i = 0; i ! N; i++)
       for (j = 0; j ! N; j++)
           C[i,j] = A[i,j] + B[i,j];
   Print(C);
}
```

The Parallel Programming Process

Synchronization

Communication – Exchange data

Synchronization – Exchange data to order events

    Mutual exclusion or atomicity

    Event ordering or Producer/consumer

        Point to Point

            Flags

        Global

            Barriers

Mutual Exclusion

Example

Each processor needs to occasionally update a counter

Processor 1	Processor 2
Load reg1, Counter	Load reg2, Counter
reg1 = reg1 + tmp1	reg2 = reg2 + tmp2
Store Counter, reg1	Store Counter, reg2

Mutual Exclusion Primitives

Hardware instructions

Test&Set

Atomically tests for 0 and sets to 1

Unset is simply a store of 0

```
while (Test&Set(L) != 0)  {;}
Critical Section
Unset(L)
```

Problem?

Mutual Exclusion Primitives – Alternative?

Test&Test&Set

Mutual Exclusion Primitives – Fetch&Add

Fetch&Add(var, data)

    { /* atomic action */

    temp = var

    var = temp + data

    }

    return temp

E.g., let X = 57

    P1: a = Fetch&Add(X,3)

    P2: b = Fetch&Add(X,5)

      If P1 before P2, ?

      If P2 before P1, ?

      If P1, P2 concurrent ?

Point to Point Event Ordering

Example

Producer wants to indicate to consumer that data is ready

Processor 1   Processor 2

A[1] = …   … = A[1]

A[2] = …   … = A[2]

  .   .

  .   .

A[n] = …   … = A[n]

Global Event Ordering – Barriers

Example

   All processors produce some data

   Want to tell all processors that it is ready

   In next phase, all processors consume data produced previously

   Use barriers