ADVANCED PARALLEL COMPUTING 2017
LECTURE 02 - SHARED MEMORY ARCHITECTURES

Holger Fröning
holger.froening@ziti.uni-heidelberg.de
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg

ANNOUNCEMENTS

02.05.2017: No exercise, lecture at 4pm (c.t.)

09.05.2017: No lecture, but exercise

PROGRAMMING MODELS

PARALLEL COMPUTING

A collection of processing elements that communicate and cooperate to solve large problems fast." - Almasi & Gottlieb, 1989

Parallel architectures

- Extend the usual concepts of computer architecture
- Add a communication architecture

Computer & communication architecture consist of

- Definition of critical abstractions (ISA, ...)
- Organizational structure that realizes these abstractions
- Main goal: high performance

Programming model => user-level communication primitives (communication abstraction), similar to ISA

PROGRAMMING MODELS

Shared memory programming

- Posting information at known (shared) locations
- Orchestration by observation (arbitrary event)

Message passing

- Conveying information from a specific sender to a specific receiver
- Well-defined event when information is sent or received

Data-parallel processing

- Highly structured form of cooperation
- Multiple elements of a data set are processed simultaneously
- Information is exchanged globally before jointly continuing

SHARED MEMORY

Similarities to executing multiple processes by time-sharing on a single processor

Process: defined as a single (virtual) address space with one or more threads of control

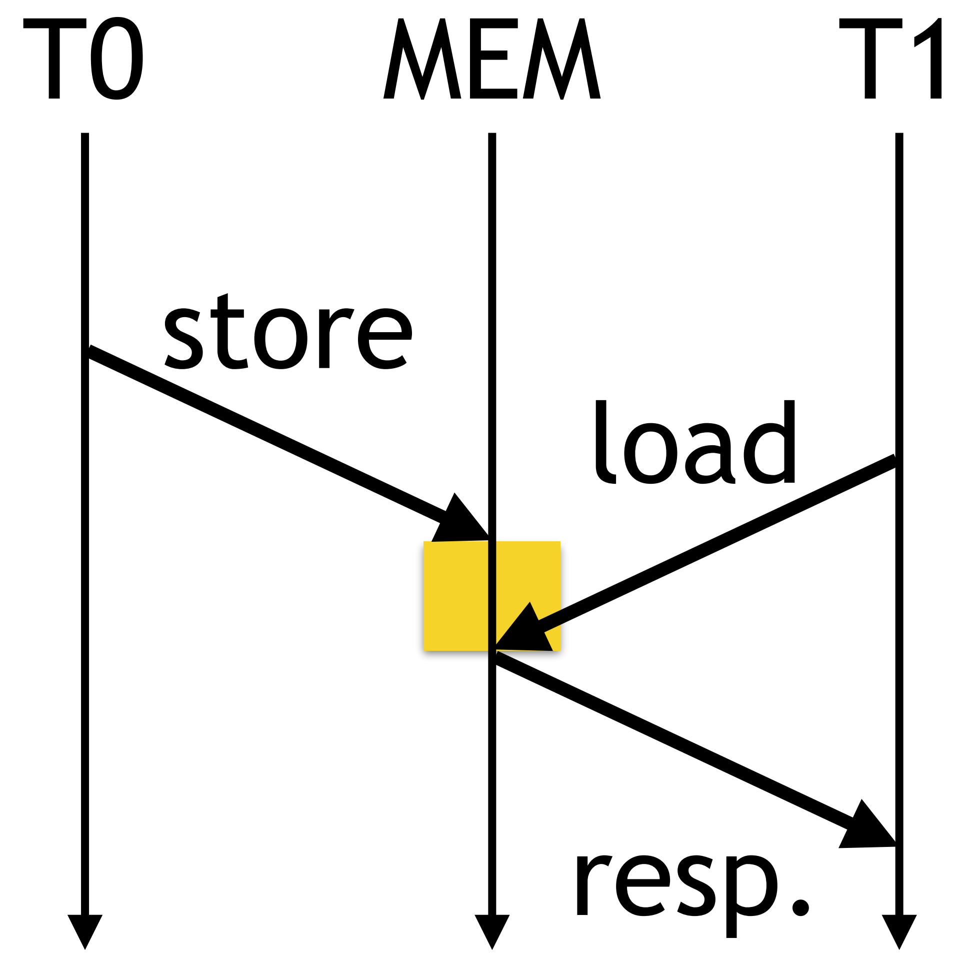
- Multiple threads share one address space by definition
- Portions of the address space can be shared, multiple virtual addresses (VA) map to a single physical address (PA)

Communication and Synchronization

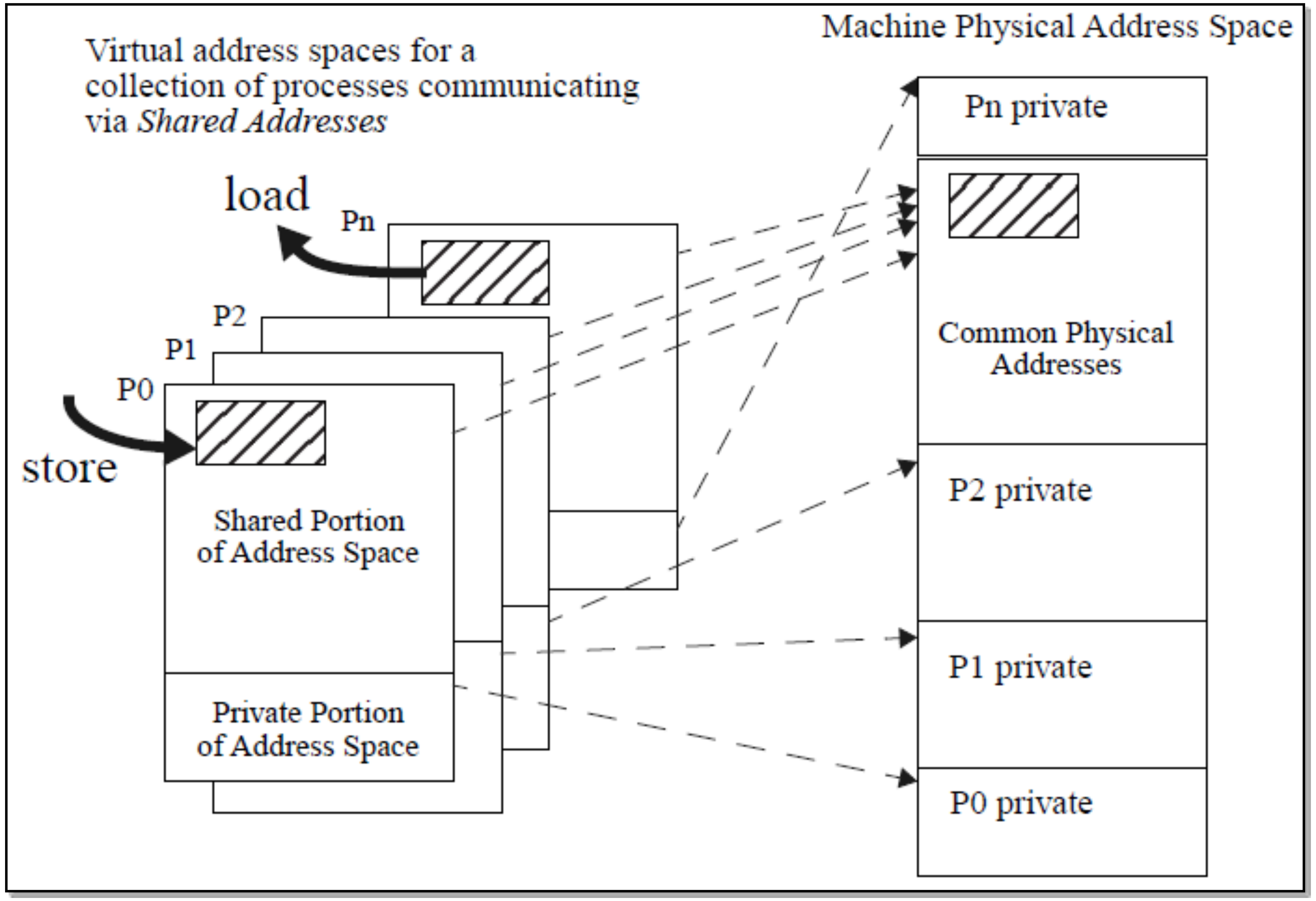
- Writes to a logically shared address by one thread are visible to reads of the other threads
- Rely on memory operations, including atomic operations

Virtual address space typically quite structured

- Private and shared segments



SHARED MEMORY



Culler et al, Parallel Computer Architecture, MK 1999

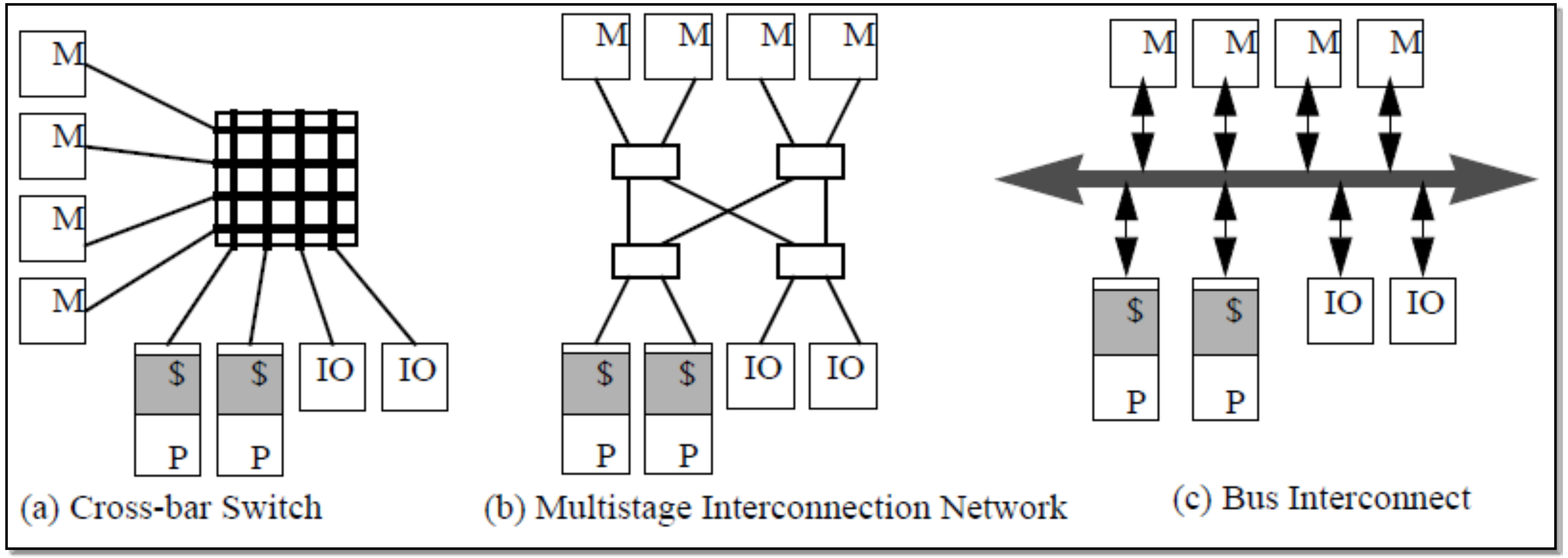
An address space defines a range of discrete addresses; each address may correspond to a different resource

SHARED MEMORY

Extending to a shared-memory multiprocessor by adding processors

- Typical shared memory multiprocessor interconnection scheme
- (Non-) Uniform Memory Access ((N)UMA)

Recent CPU architectures?



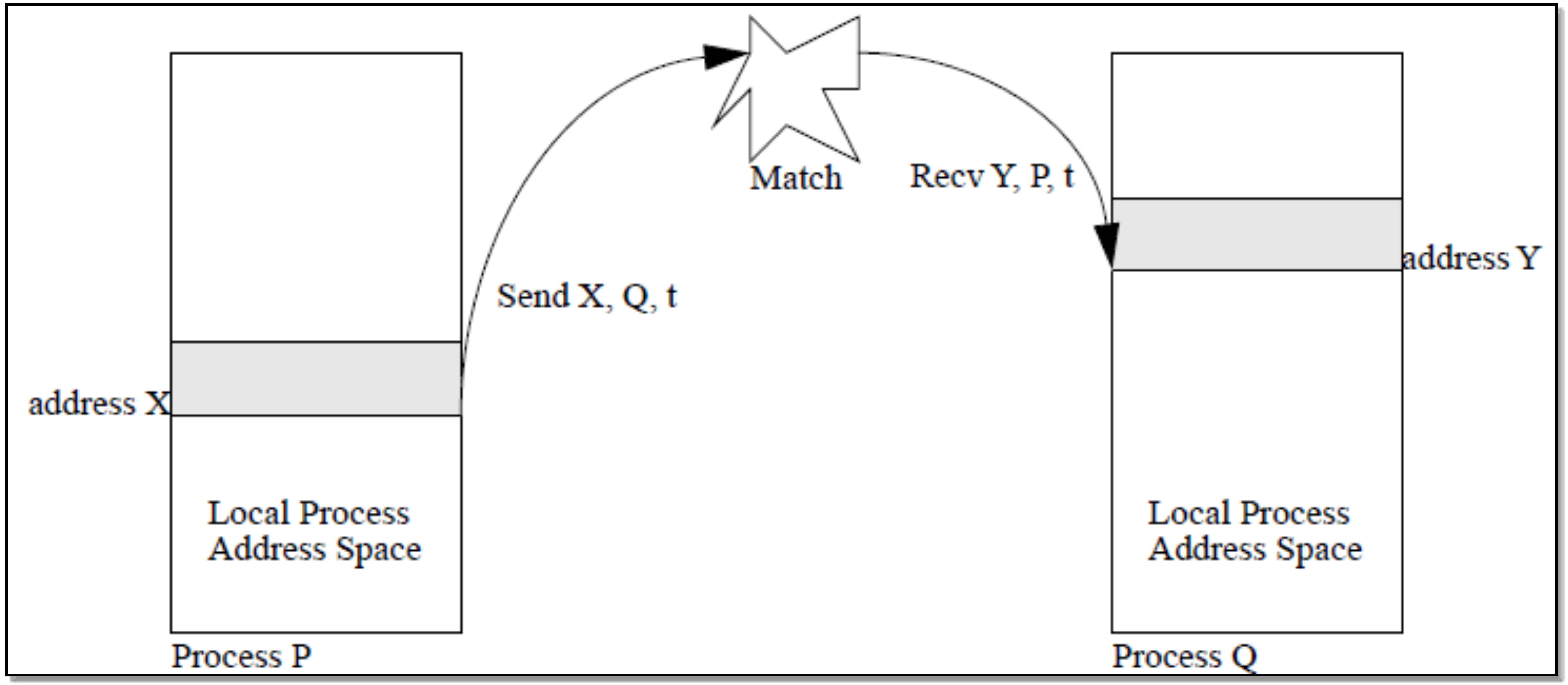
Culler et al, Parallel Computer Architecture, MK 1999

MESSAGE PASSING

No remote access, local processes control data placement

Matching to identify corresponding sends/writes

No Remote Memory Access (NORMA)



Culler et al, Parallel Computer Architecture, MK 1999

DATA-PARALLEL PROCESSING

Parallel processing on each element of a large regular data structure

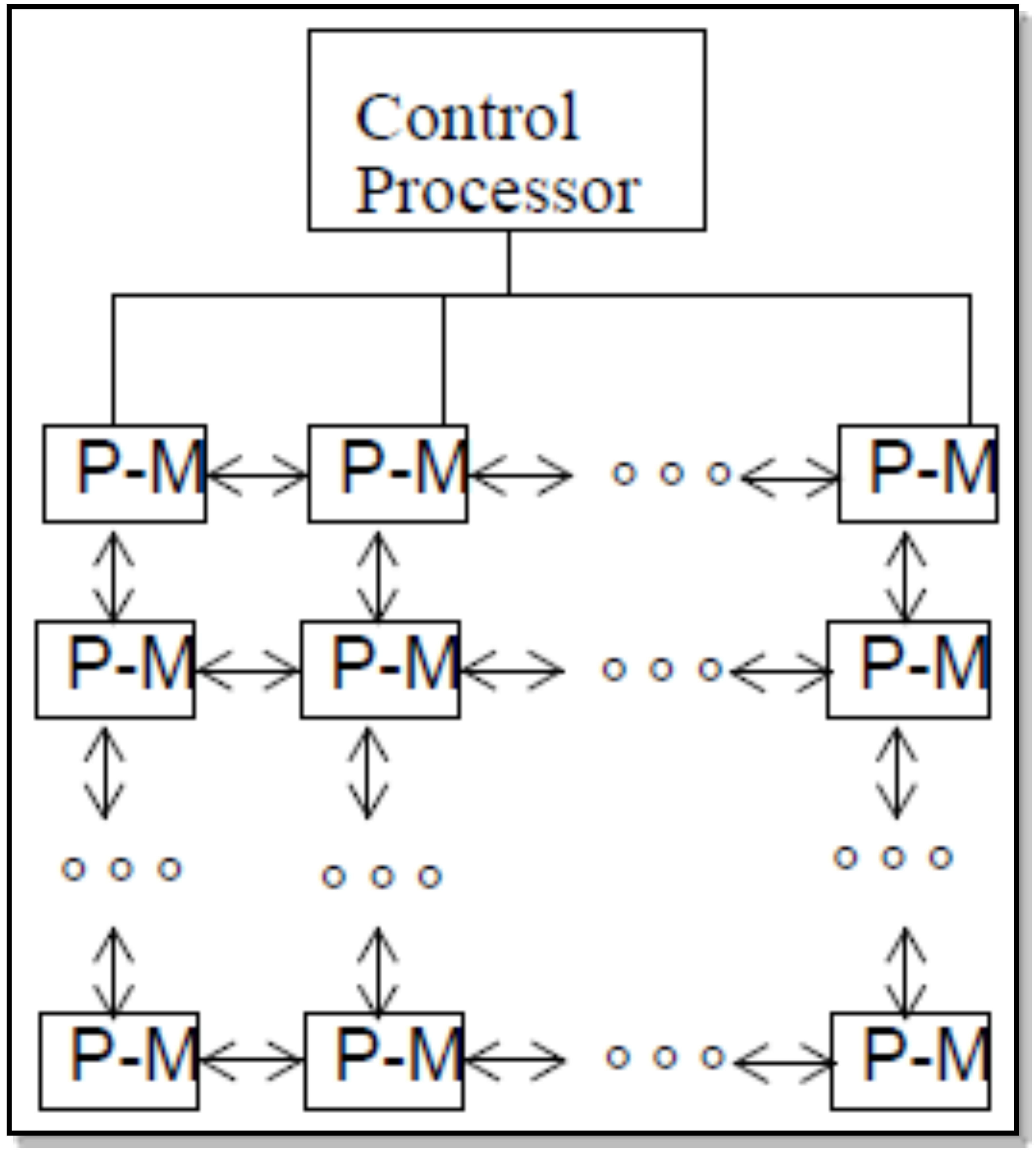
- Vectors/arrays, matrices, ...

Logical: single thread of control

- SIMD

Incarnations

- Vector processors (today: as extension only)
- GPU Computing (very alive)
- SSE ISA extensions (very alive)
- Dataflow architectures (dead)
  - Messages as tokens, tags/addresses



Culler et al, Parallel Computer Architecture, MK 1999

CONVERGENCE

Message passing over shared memory (library)

- Send: write data to a certain shared buffer
- Receive: read data from shared storage

Shared memory over message passing (compiler & library)

- (Logical) Read/Write: send request to process owning the object and receiving a response
- Messaging is hidden from the user, compiler-generated code to access a shared variable

Shared Virtual Address space over message passing at page level (OS)

- Use of page faults to migrate remote pages into the local system

Virtualization of data-parallel architectures

- Illusion of scalar execution (GPUs)

Distributed systems (message passing) typically rely on shared memory multiprocessors as building block

FUNDAMENTAL DESIGN ISSUES

FUNDAMENTAL DESIGN ISSUES

Communication abstraction

Contract, similar to ISA

1. Naming

   What data can be named?

2. Operations

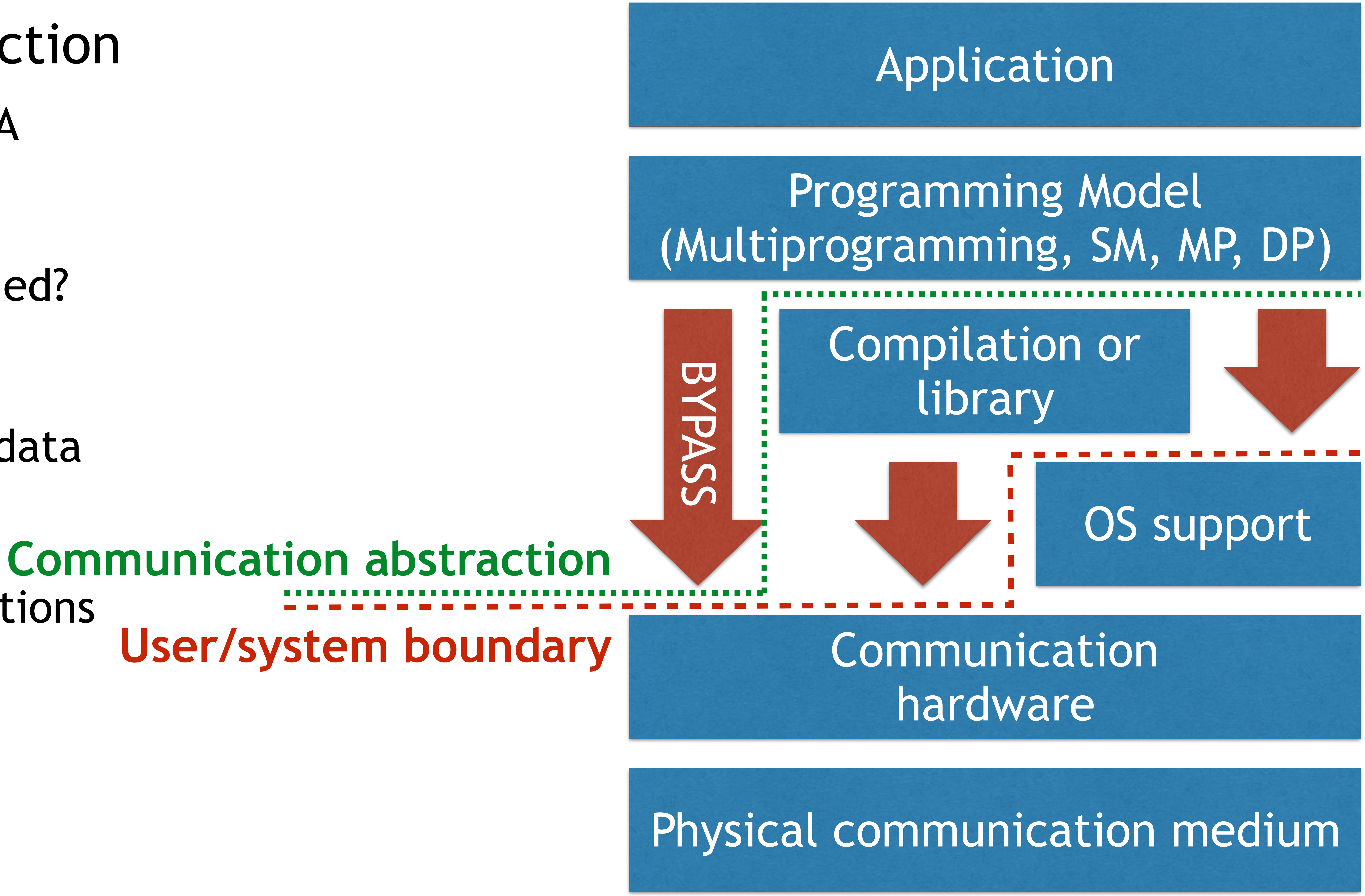
   Operations on named data

3. Ordering

   Ordering among operations

4. Communication/ Replication of data

5. Performance



FUNDAMENTAL DESIGN ISSUES: NAMING

- Shared memory
  - Naming: Thread can name locations in the register and the virtual address space
    - Segments for code, stack, heap
  - Access to shared variables mapped to load/store instructions on virtual addresses
  - Global physical address space: shared virtual addresses map to the same physical address
  - Independent local physical address spaces: page faults
- Message passing
  - Message passing in hardware, but matching/buffering better in software
- Issue of naming arises at each abstraction level of a parallel architecture

FUNDAMENTAL DESIGN ISSUES: OPERATIONS

- Shared memory
  - Loads and stores on addresses and registers (CISC), only registers (RISC)
  - Reading/writing shared variables
  - Atomic read-modify-write operations on shared variables
- Message passing
  - Sending/receiving on (private) local addresses and process identifiers
  - Collective operations
- Note the different complexity!

FUNDAMENTAL DESIGN ISSUES: ORDERING

- Shared memory
  - Threads operate independently, so which order to apply?
  - Among memory operations: sequential program order
  - Variables are read and modified: top-to-bottom, left-to-right order of the program
- Message passing
  - MPI guarantees strong ordering
  - Tag matching, matching results in linear search(es)
  - Receive any tag/sender will just return the first matched queue entry
- Ordering has big performance impact
  - Relaxed ordering models

FUNDAMENTAL DESIGN ISSUES

- Communication/Replication of Data
  - Data movement: explicit (message passing) vs. implicit (shared memory)
  - Memory hierarchy: explicit (registers, scratchpad memory) vs. implicit (caches)
- Performance issues (costs of data movement)
  - Latency: time taken for a certain operation
  - Bandwidth: rate at which operations are performed
  - Costs: impact of operations on the execution time of the program
  - Overhead: required processing time for communication
  - Relation:
    - Sequential execution: direct relation between those (no overlap)
    - Concurrency/parallelism: much more complex relation

SHARED MEMORY MULTIPROCESSORS

SHARED MEMORY MULTIPROCESSORS

Multiple execution contexts sharing a single address space

- Multiple processes/threads, multiple data (MIMD)
- Simplification: Single Program Multiple Data (SPMD)

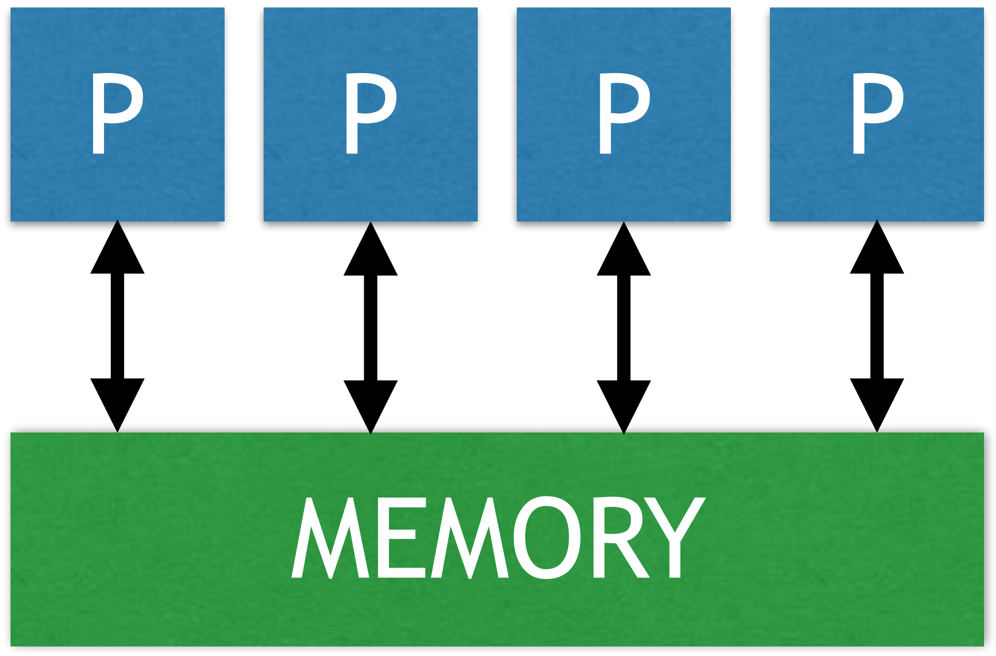
Parallelism type: TLP, DLP

Advantages:

- Applications: looks like multi-threaded uniprocessor
- OS: only evolutionary extensions required
- OS-bypass for communication
- Software development: first correctness, then performance

Disadvantages:

- Synchronization is very difficult
- Implicit communication is harder to optimize



Theoretical foundation:
Parallel Random Access Machine (PRAM)

Symmetric Multiprocessors (SMP) and Chip Multiprocessors (CMP) are the most successful parallel machines ever

SYSTEM ARCHITECTURE

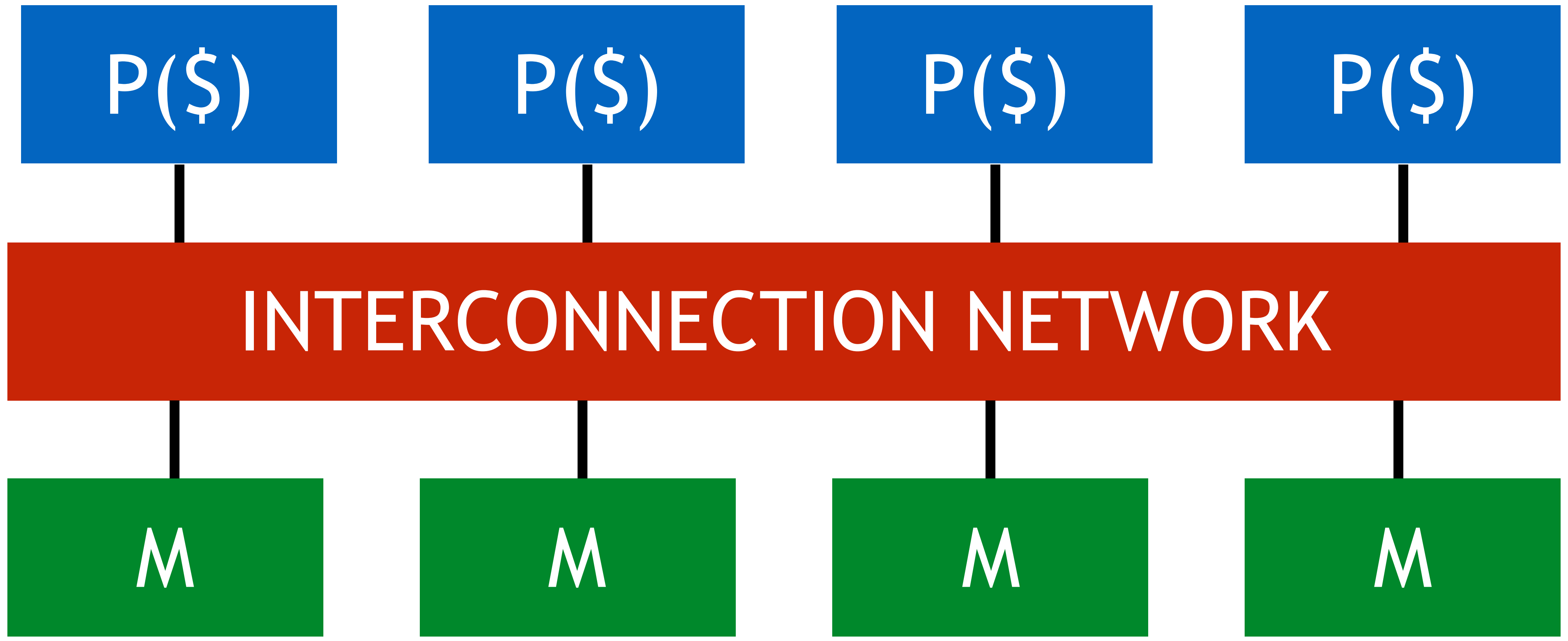
Separate processor / memory

Uniform memory access (UMA)

+ No locality effects

- Lower peak performance

Bus-based UMAs common



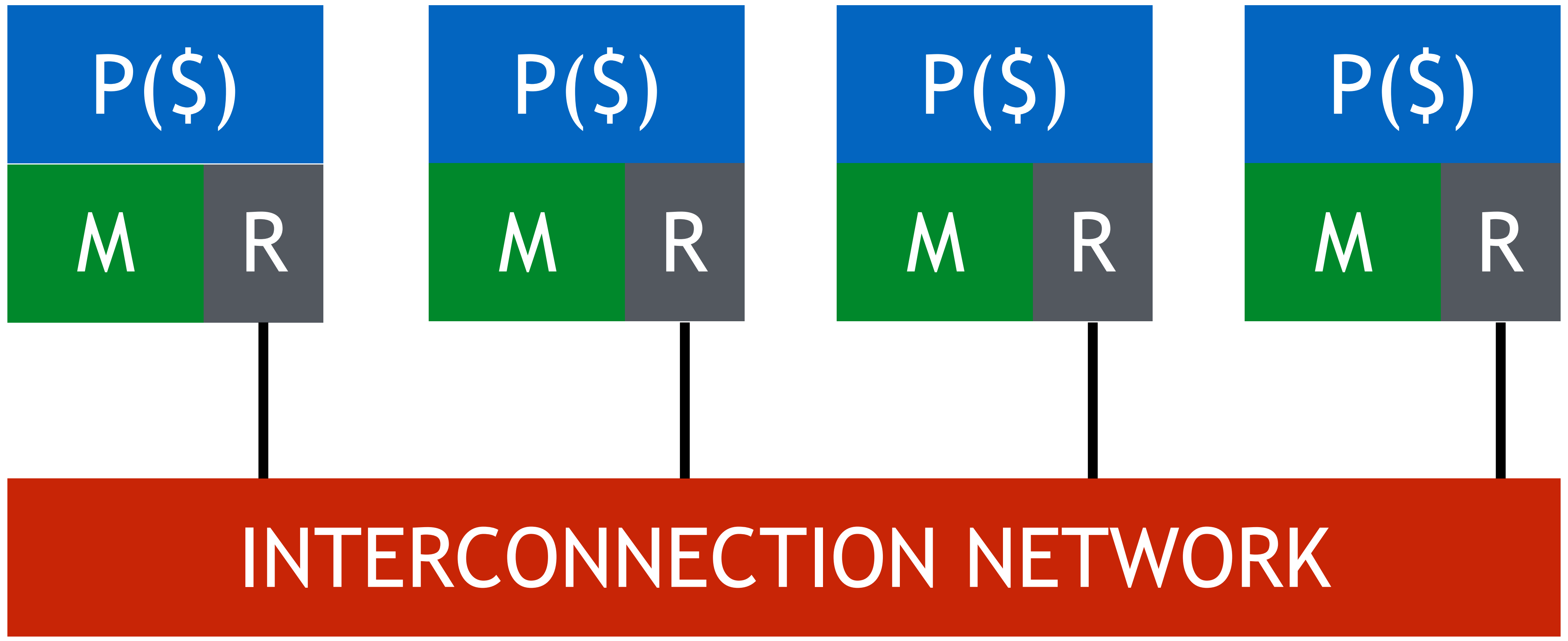
Paired processor / memory

Non-uniform memory access (NUMA)

+ Faster local memory access

- Locality matters!

Higher peak performance assuming proper data placement



SYSTEM ARCHITECTURE - NETWORK

Shared network

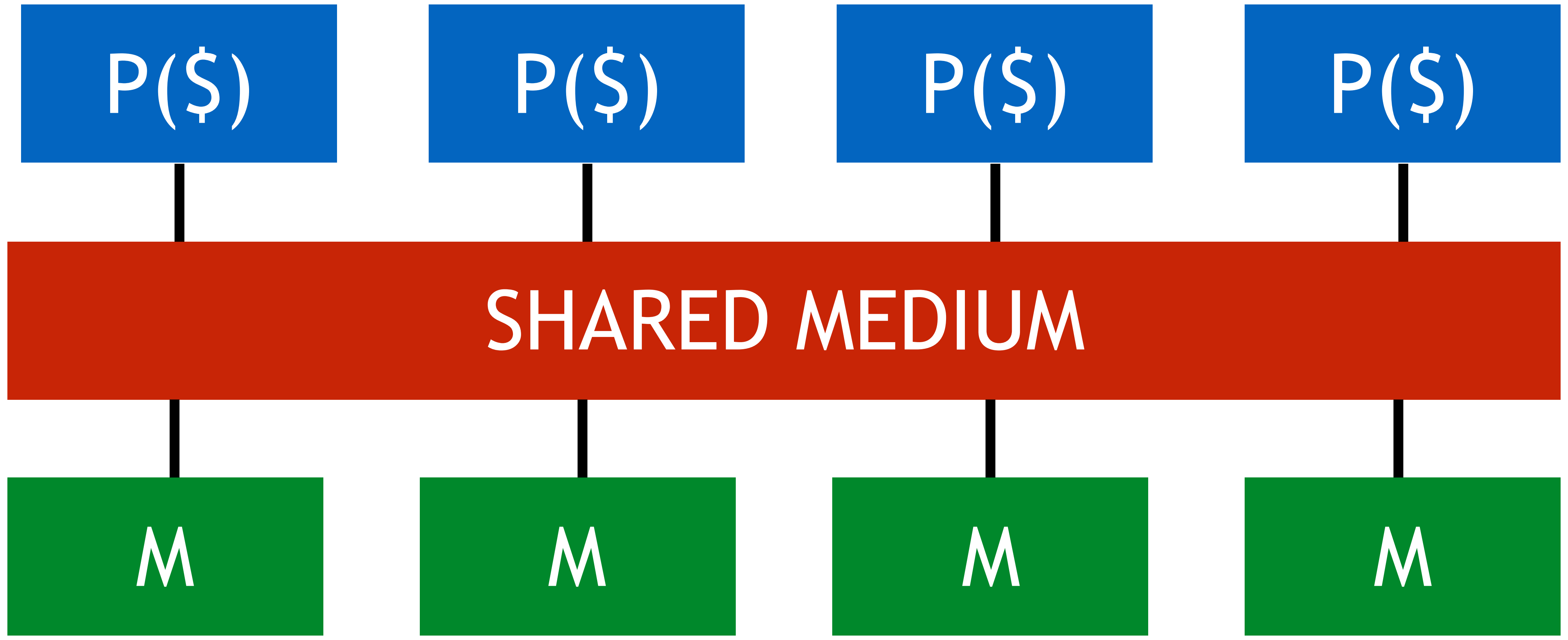
Low latency

Low bandwidth

Limited scalability

Simple coherence protocols

E.g., bus



Point-to-point network

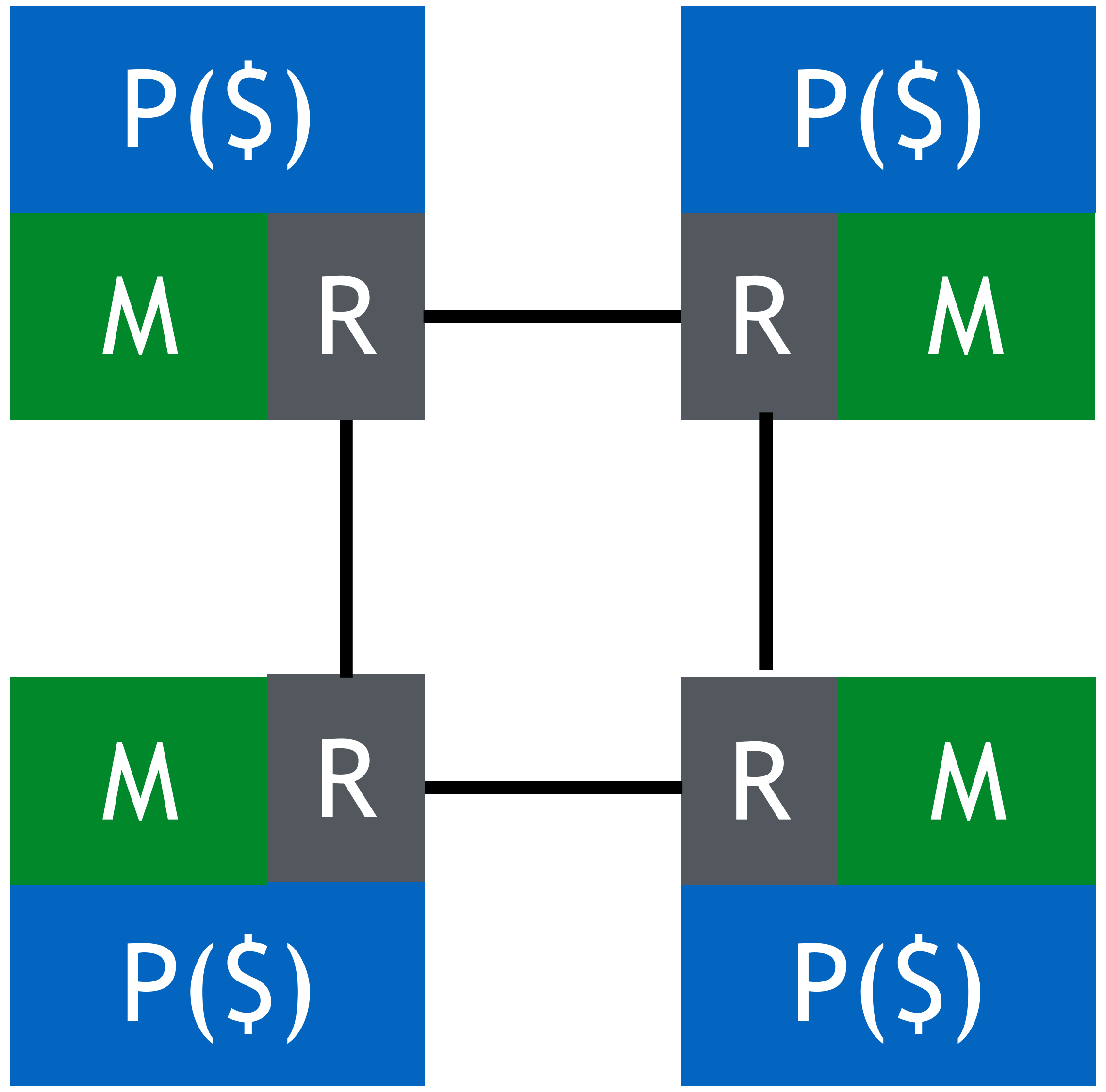
Higher latency, more hops

Higher bandwidth

High scalability

Complex coherence protocols

E.g., mesh, ring, multi-stage

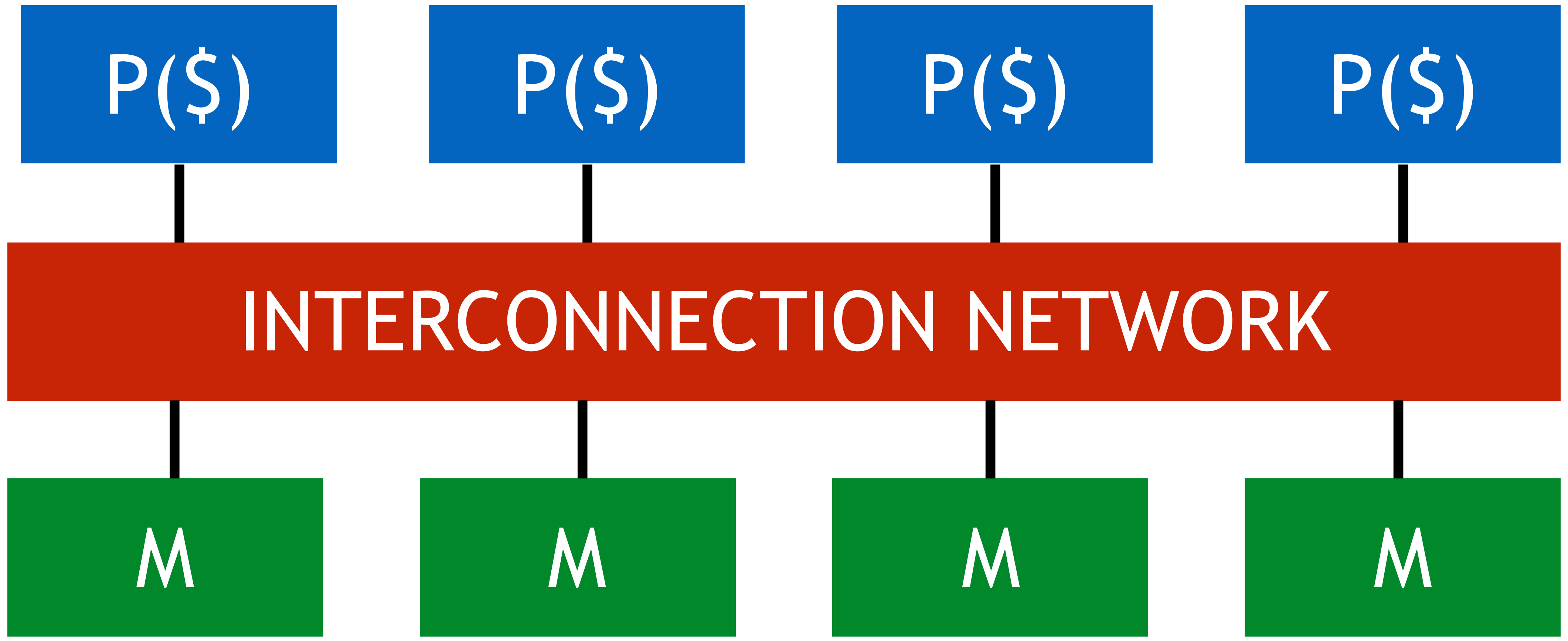


FUNDAMENTAL DIFFERENCE

Uniform Memory Access (UMA)

Memory is equidistant from processors

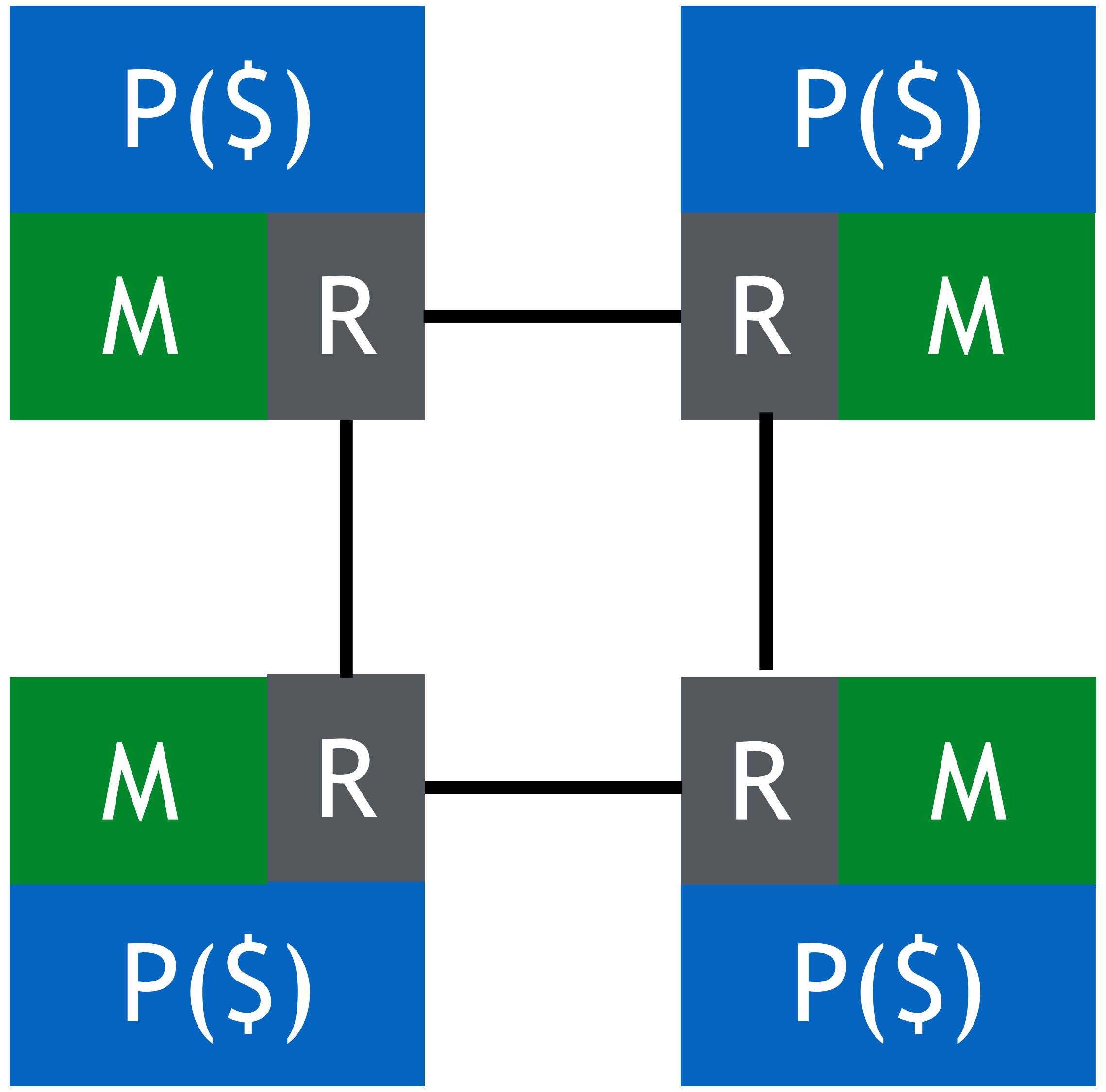
Data placement of no importance



Non-Uniform Memory Access (NUMA)

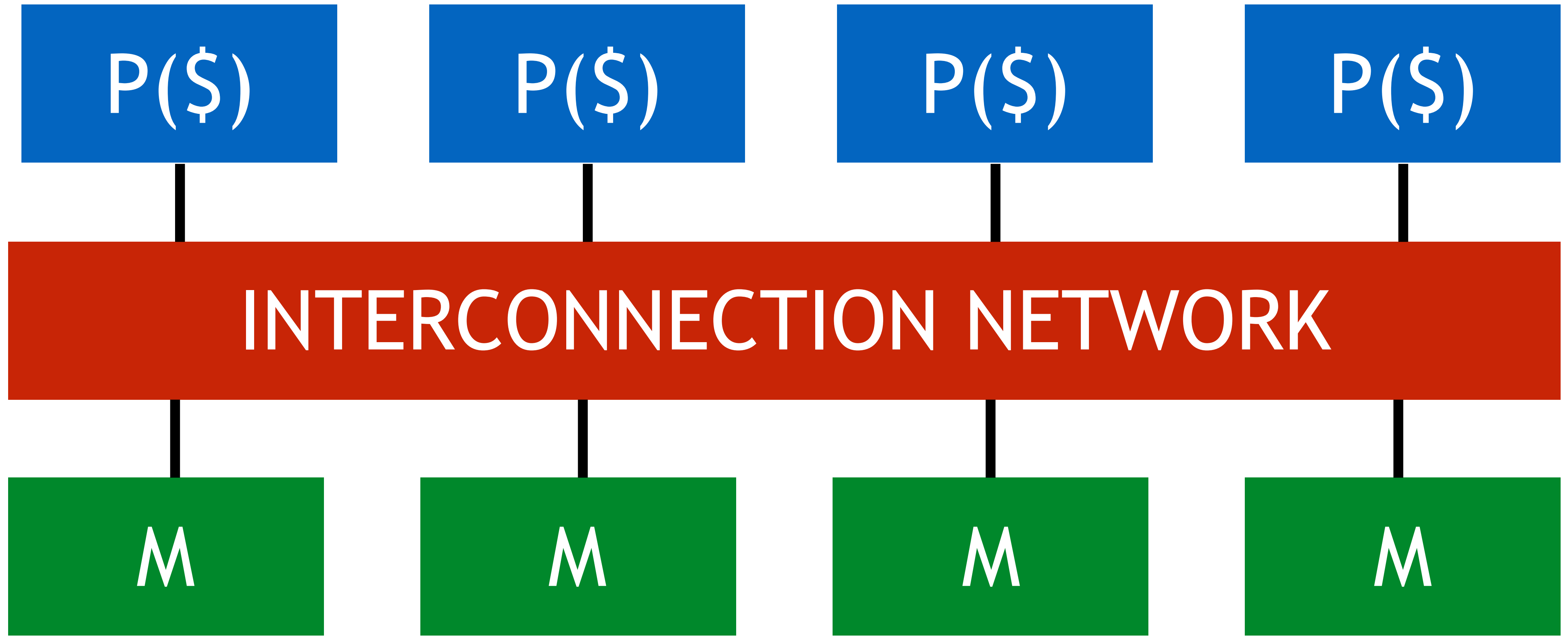
Different access costs for different memory modules

Data placement of high importance



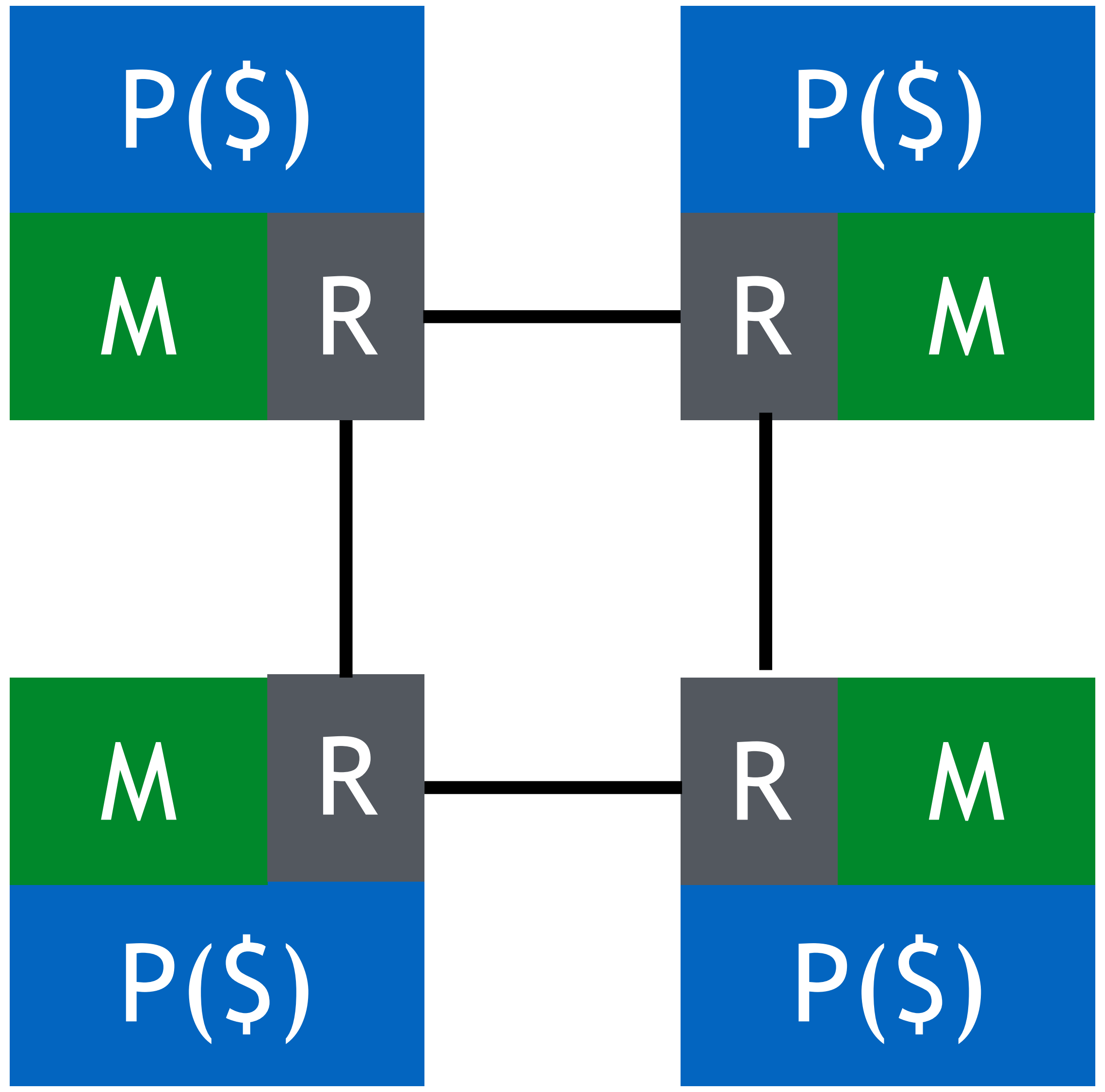
MOST IMPORTANT ISSUES

1. Cache coherence
2. Memory consistency model



Closely related to each other

Different solutions for UMA and NUMA



CONSISTENCY & COHERENCE

EXAMPLE APPLICATION

Example: database or web server

- Each query is a thread

Register allocation is the process of assigning a large number of target program variables onto a small number of CPU registers

- Shared variables can‘t be register allocated (accts)
- Private variables should be register allocated

```
struct account_t { int balance; }
shared struct account_t a[MAX];

int aID, int amount;

if ( a[aID].balance >= amount ) {
    a[aID].balance -= amount
    spew_cash ( amount );
}
```

```
0:  addi r1, a, r3
1:  ld 0(r3), r4
2:  blt r4, r2, 6
3:  sub r4, r2, r4
4:  st r4, 0(r3)
5:  call spew_cash
6:  noop
```

PARALLEL EXECUTION EXAMPLE

P0

```
0:   addi r1, a, r3
1:   ld 0(r3), r4
2:   blt r4, r2, 6
3:   sub r4, r2, r4
4:   st r4, 0(r3)
5:   call spew_cash
6:   noop
```

P1

```
40: addi r1, a, r3
41: ld 0(r3), r4
42: blt r4, r2, 6
43: sub r4, r2, r4
44: st r4, 0(r3)
45: call spew_cash
46: noop
```

P0 ($)	P1 ($)	MEM

Two (almost) concurrent withdrawals from account #42 at two ATMs

- Each transaction maps to a different thread on a different processor
- Track accts[42].bal (address in register r3)

PARALLEL EXECUTION EXAMPLE
NO CACHES

P0

```
0:  addi r1, a, r3
1:  ld 0(r3), r4
2:  blt r4, r2, 6
3:  sub r4, r2, r4
4:  st r4, 0(r3)
5:  call spew_cash
6:  noop
```

P1

```
40: addi r1, a, r3
41: ld 0(r3), r4
42: blt r4, r2, 6
43: sub r4, r2, r4
44: st r4, 0(r3)
45: call spew_cash
46: noop
```

	P0 ($)	P1 ($)	MEM
1:			500
4:			400
41:			400
44:			300

Two (almost) concurrent withdrawals ($100) from account #42 at two ATMs

- Each transaction maps to a different thread on a different processor
- Track accts[42].bal (address in register r3)

PARALLEL EXECUTION EXAMPLE
CACHE INCOHERENCE

P0
```
0:   addi r1, a, r3
1:   ld 0(r3), r4
2:   blt r4, r2, 6
3:   sub r4, r2, r4
4:   st r4, 0(r3)
5:   call spew_cash
6:   noop
```

P1
```
40:  addi r1, a, r3
41:  ld 0(r3), r4
42:  blt r4, r2, 6
43:  sub r4, r2, r4
44:  st r4, 0(r3)
45:  call spew_cash
46:  noop
```

	P0 ($)	P1 ($)	MEM
1:	V:500		500
4:	D:400		500
41:	D:400	V:500	500
44:	D:400	D:400	500

Two (almost) concurrent withdrawals ($100) from account #42 at two ATMs

- Each transaction maps to a different thread on a different processor
- Track accts[42].bal (address in register r3)

COHERENCE VS. CONSISTENCY

- Bus inherently provides synchronization point, serializing all accesses
    - Each cache controller snoops/listens to bus transactions
- Take action to ensure coherence, depending on state of the cache block (cache line) and the protocol: Invalidate/Update & Supply value
- Coherence vs. consistency: Intuition says loads should return latest value
    - Who is latest?
    - Coherence concerns only one memory location
    - Consistency concerns apparent ordering for all locations
- A memory system is coherent, if it:
    - can serialize all operations to that location such that,
    - operations performed by any processor appear in program order
      (order defined by program or assembler code)
    - value returned by a read is the value written by last write to that location

COHERENCE != CONSISTENCY

```
//initial A = B = flag = 0

Processor 0                    Processor 1
A = 1;                         while (flag == 0); // spin
B = 1;                         print A;
flag = 1;                      print B;
```

Intuition says: A = B = 1

- Implicit ordering by synchronizing using flag

Coherence doesn‘t say anything. Why?

- Coherence says nothing about the order in which writes to different locations become visible

Uniprocessor ordering mechanism (load/store queue) won‘t work. Why?

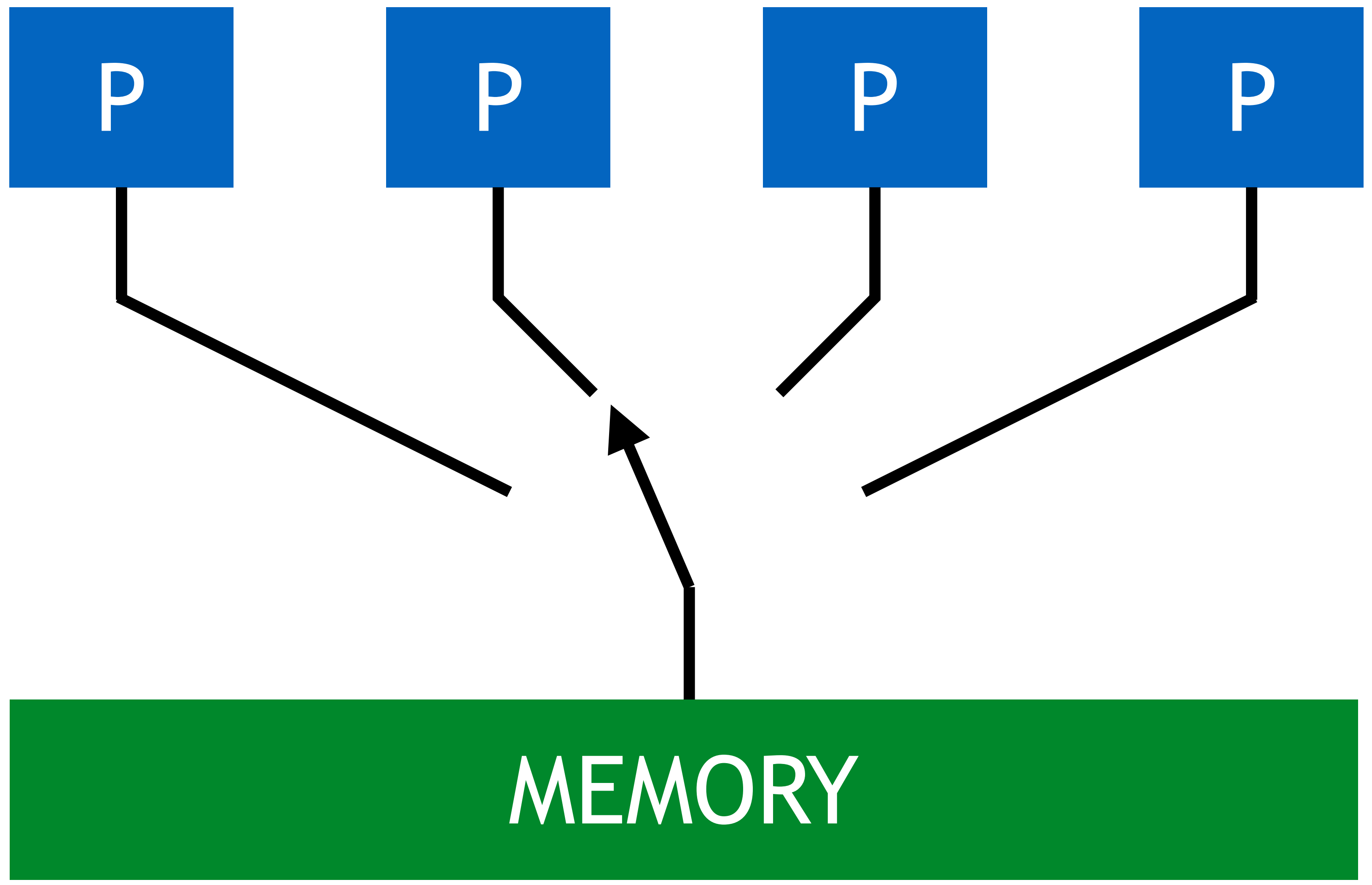
- P0 – Memory(A,B) – Memory(flag) - P1

SEQUENTIAL CONSISTENCY

Processors issue memory requests in program order

Switch set randomly after each memory operation

=> Provides sequential ordering among all operations



SEQUENTIAL CONSISTENCY

Sufficient condition for SC:

„A multiprocessor 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, 1979

Every processor issues memory requests in program order

Memory operations happen (start and end) atomically

Must wait for a store to complete before issuing next operation

After a load, issuing processor waits for load to complete, before issuing next operation

Easily implemented with a shared bus

Bus as **synchronization point**, serializing all accesses

COHERENCE != CONSISTENCY

```
//initial A = B = flag = 0

Processor 0                    Processor 1
A = 1;                         while (flag == 0); // spin
B = 1;                         print A;
flag = 1;                      print B;
```

Assume SC, which results for {A,B} are possible?

A = B = 1

PROBLEMS WITH SEQUENTIAL CONSISTENCY

Aspect 1: difficult to implement efficiently in hardware

- No concurrency among memory access
- Strict ordering of memory accesses at each processor (node)
- Essentially precludes out-of-order CPUs

Aspect 2: unnecessarily restrictive

- Most parallel programs won‘t notice out-of-order accesses

Aspect 3: conflicts with latency hiding techniques

- Which relies on many concurrent outstanding requests

Fixing SC performance

- Revert to a less strict consistency model (relaxed or weak consistency)
- Programmer specifies when ordering matters

PROBLEMS WITH SCALABLE CACHE COHERENCE

Aspect 1: Bandwidth

- Bus as a shared medium is not scalable at all
- Replace bus with a switched network (direct or indirect)

Aspect 2: Snooping overhead

- Interesting: most snoops result in no action
- Simply because no copy of the corresponding cache line is present
- Broadcast protocol is not scalable
- Revert to a directory protocol, only addressing processors that hold cache line copies (broadcast/multicast)

No caches, no coherence problem

- However: consistency problem sustains

SUMMARY

Shared-memory multiprocessors

+ Simple software: easy data sharing, exploits both TLP and DLP

- Complex hardware: must provide the illusion of a single global address space

Implementations

Symmetric Multiprocessors (UMA) - bus-based, simple protocols that rely on global ordering

Scalable Multiprocessors (NUMA) - switched point-to-point, unordered, scalable bandwidth, complex protocols

Two aspects to the global address space illusion

Coherence: consistent view of individual cache lines

Consistency: consistent global view of all memory locations

Programmers: intuitively expect sequential consistency (SC)