Slide 483

Multiprocessor Systems

Slide 484

Multiprocessor - computer system containing more than one processor.

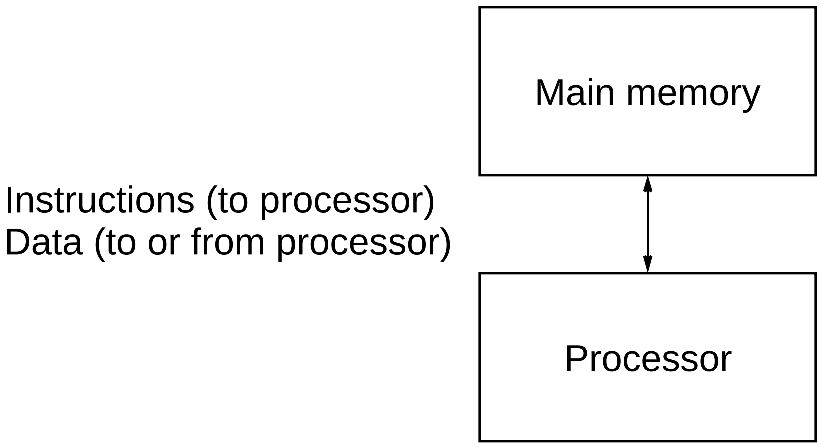
Principal motive is to increase the speed of execution of the system.

Sometimes other motives, such as fault tolerance.

Apparent that increased speed should result when more than one processor operates simultaneously.

Slide 485

Conventional Computer



Main memory

Instructions (to processor)
Data (to or from processor)

Processor

Each main memory location located its *address*. Addresses start at 0 and extend to $2^n - 1$ when there are *n* bits in address.

Slide 486

Types of multiprocessor systems

(where each processor executes its own program)

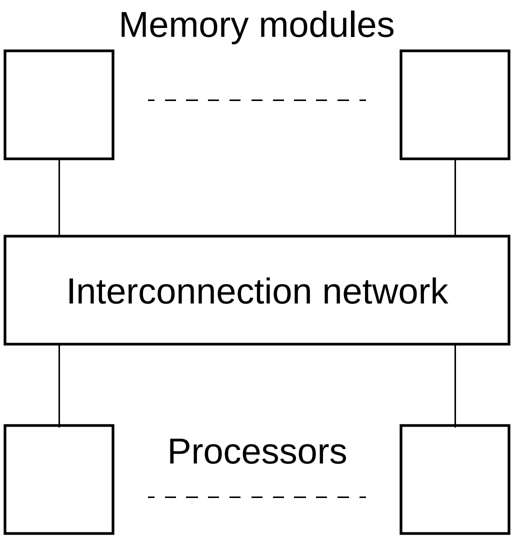
Shared memory multiprocessor system - a natural extension of a single processor system in which all the processors can access a common memory.

Distributed memory multicomputer system -multiple interconnected computers where each computer has its own memory.

Slide 487

Shared Memory Multiprocessor System

Each processor can access any memory location. One address space.



Memory modules

- - - - - - - - - - - -

Interconnection network

Processors

- - - - - - - - - - - -

Slide 488

Interconnection networks

Various possible networks:

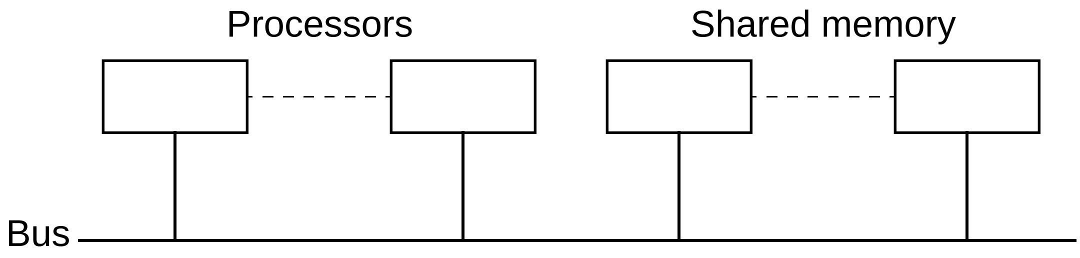
- Single bus
- Multiple buses (not much used)
- Rings
- Mesh
- Hypercube (popular in the 1980’s, not any more)
- Multistage interconnection networks (MINs)

Here, we will only consider the single bus approach used in small multiprocessor systems, for example quad Pentium systems.

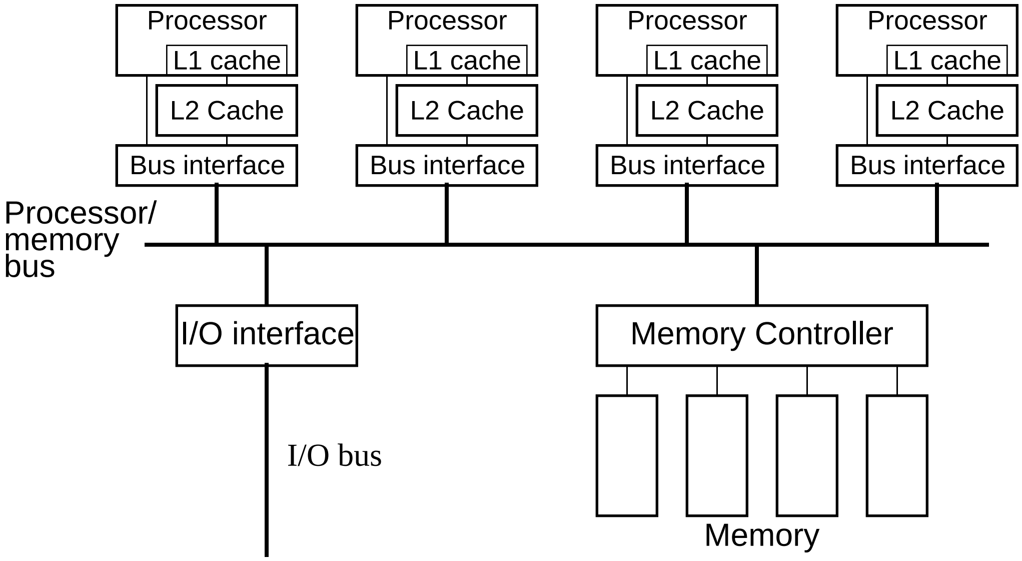
Shared bus multiprocessor system

A natural extension to a single bus microprocessor systems.

Simplistic view:

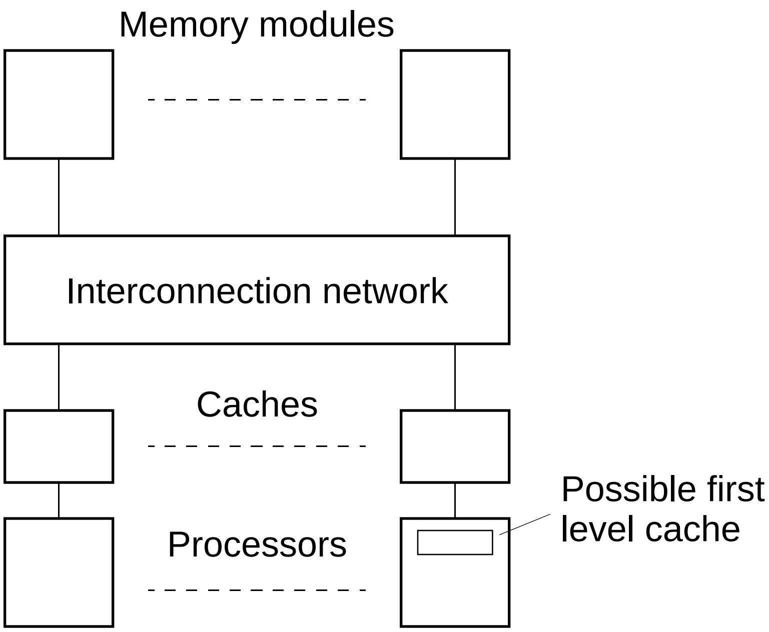


Quad Pentium shared memory multiprocessor system



Slide 491

General model of a shared memory multiprocessor system with caches



Slide 492

Cache Coherence

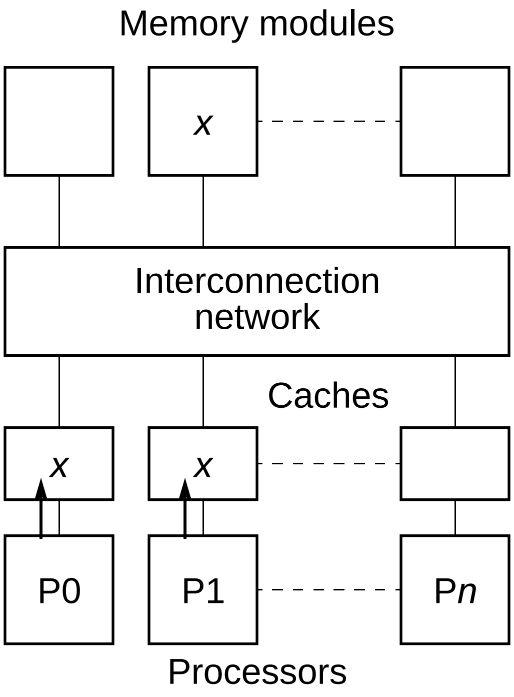
Significant additional factors to consider in using cache memory in a multiprocessor environment, in particular maintaining accurate copies of data in the multiple caches in the system.

Maintaining copies in all the caches the same is known as *cache coherence* Any read should obtain the most recent value written. (Actually more complicated that this.)

Write policy

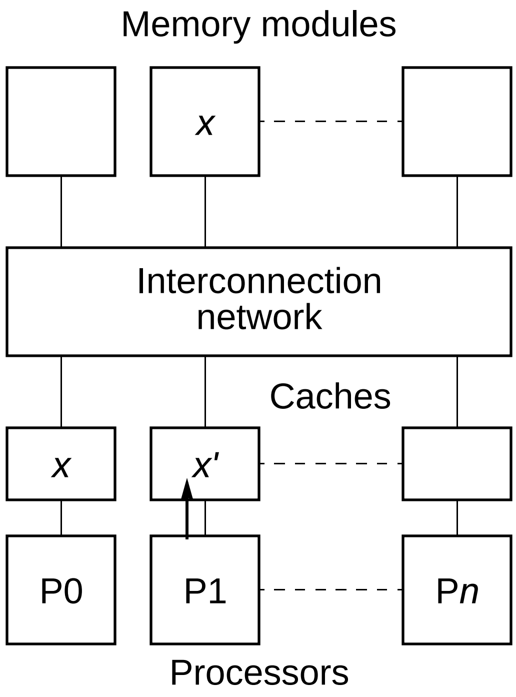
Write-through is not sufficient, or even necessary, for maintaining cache coherence, as more than one processor writing-through the cache does not keep all the values the same and up to date.

Inconsistency with write through policy



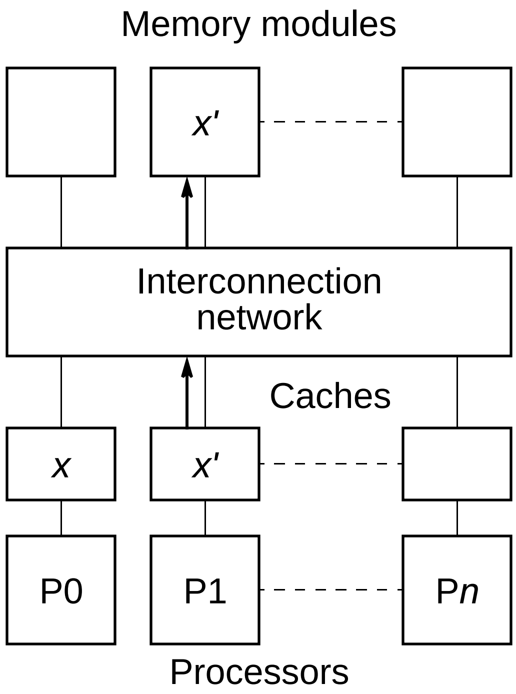
(a) Processors accessing *x*

Slide 495



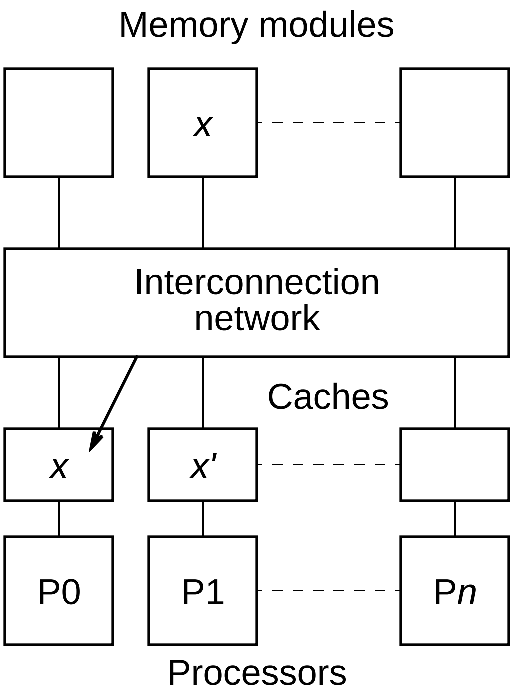
(b) Processor 1 updating x

Slide 496



(c) After write-through

Slide 497



(d) Invalidating or updating copy

Slide 498

Two possible solutions

1. Update copy in the cache of processor 0, or

2. Invalidate copy in the cache of processor 0

both of which require access to the cache of processor 0.

Slide 499

Update

Update writes all cached copies with the new value of *x*.

Not usually implemented because of the overhead of the update.

In any event, it may be not completely necessary because not all processors may access the location again.

Slide 500

Invalidation

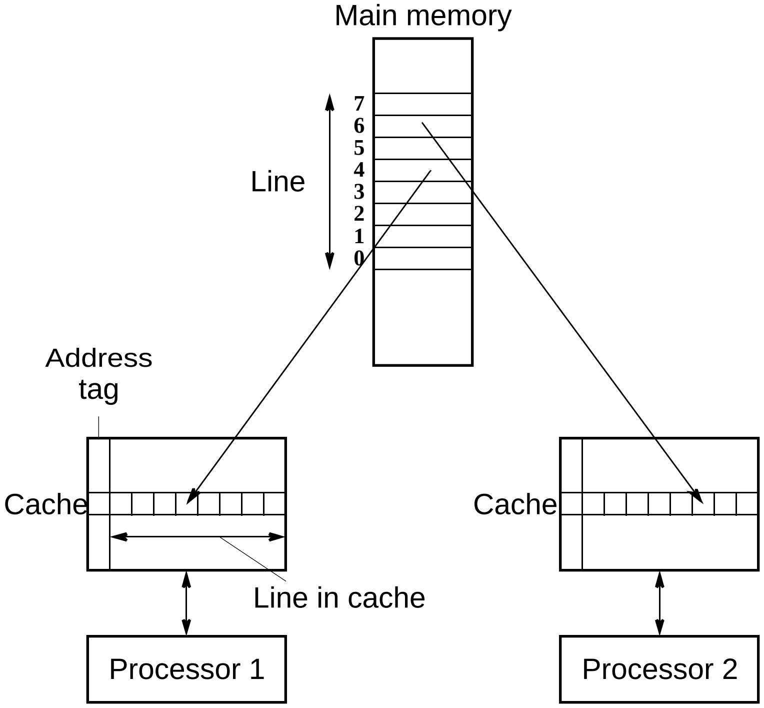
Done by resetting the valid bit associated with *x* in the cache. Now processor 0 must access the main memory if it references *x* again, to bring a new copy of *x* back into its cache. If copies existed in caches apart from the cache of processor 1, these copies would also need to be invalidated.

With invalidation, write back may be practiced rather than write-through to reduce the memory traffic. Then there is only one valid copy in one cache, and one processor has *ownership* of this copy.

Slide 501

False sharing

When more than one processor accesses different parts of a line but not the actual data items.



Slide 502

False sharing can result in significant reduction in performance because, in maintaining cache coherence, the smallest unit considered is the line.

False sharing can be reduced by distributing the data into different lines if sharing is expected.

A task for the compiler, and requires both knowledge of the use of the data and the architectural arrangements of the caches.

Alter the layout of the data stored in the main memory, separating data only altered by one processor into different blocks.

Slide 503

May be difficult to satisfy in all situations.

Example

```
forall (i = 0; i < 5; i++)
   a[i] = 0;
```

is likely to create false sharing as the elements of **`a`**, **`a[0]`**, **`a[1]`**, **`a[2]`**, **`a[3]`**, and **`a[4]`**, likely to be stored in consecutive locations in memory.

Would need to place each element in a different block, which would create significant wastage of storage for a large array.

`forall` is a high level language construct that says do the body with each value of **`i`** simultaneously.

Slide 504

Methods of Achieving Cache Coherence

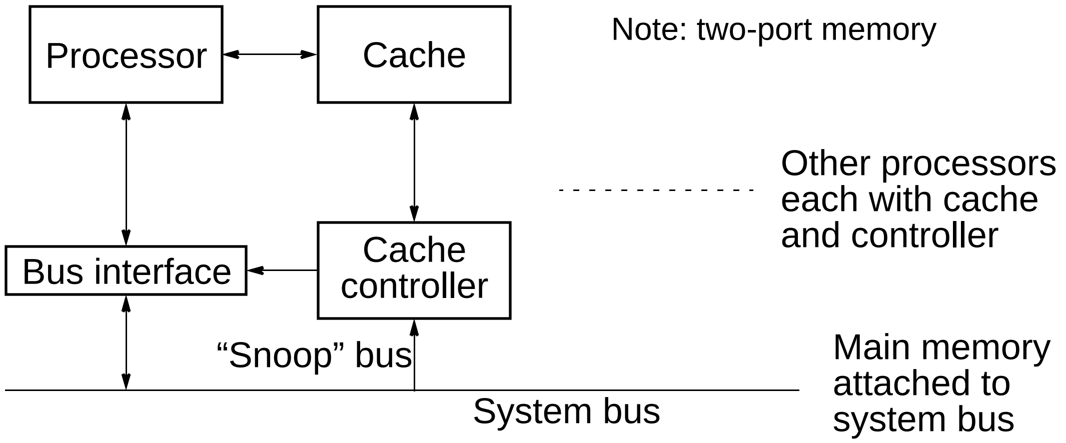
For a single bus structure, snoop bus mechanism often used.

Snoop bus mechanism

In the *snoop bus* mechanism, a “bus watcher” unit with each processor/cache observes the transactions on the bus and in particular monitors all memory write operations. If a write is performed to a location which is cached locally, this copy is invalidated - needs a protocol -see later. Could invalid based upon only index (not compare tags).

Slide 505

Snoop bus mechanism



Processor

Cache

Note: two-port memory

Other processors
each with cache
and controller

Bus interface

Cache
controller

“Snoop” bus

System bus

Main memory
attached to
system bus

Slide 506

Four-state MESI (Modified/Exclusive/Shared/ Invalid) invalidate cache coherence protocol

Can be found in the internal data cache of Intel Pentium, the second level Pentium cache controller, the Intel 82490 (Intel, 1994c), the Intel i860 processor (Intel, 1992b), and Motorola MC88200 cache controller (Motorola, 1988b), among others - with variations.

Slide 507

Each line in the cache can be in one of four states:

1. Modified (exclusive) – The line is only in this cache and has been modified (written) with respect to memory. Copies do not exist in other caches or in memory.

2. Exclusive (unmodified) – The line is only in this cache and has not being modified. It is consistent with memory. Other copies do not exist in other caches.

3. Shared (unmodified) – This line potentially exists in other caches. It is consistent with memory. To stay in this state, access to line can only be for reading.

4. Invalid – This line has been invalidated and does not contain valid data.

Slide 508

Two bits can be associated with each line to indicate state of line.

Modified (exclusive) and exclusive (unmodified) states are used to indicate that the processor has the only copy of the cache line.

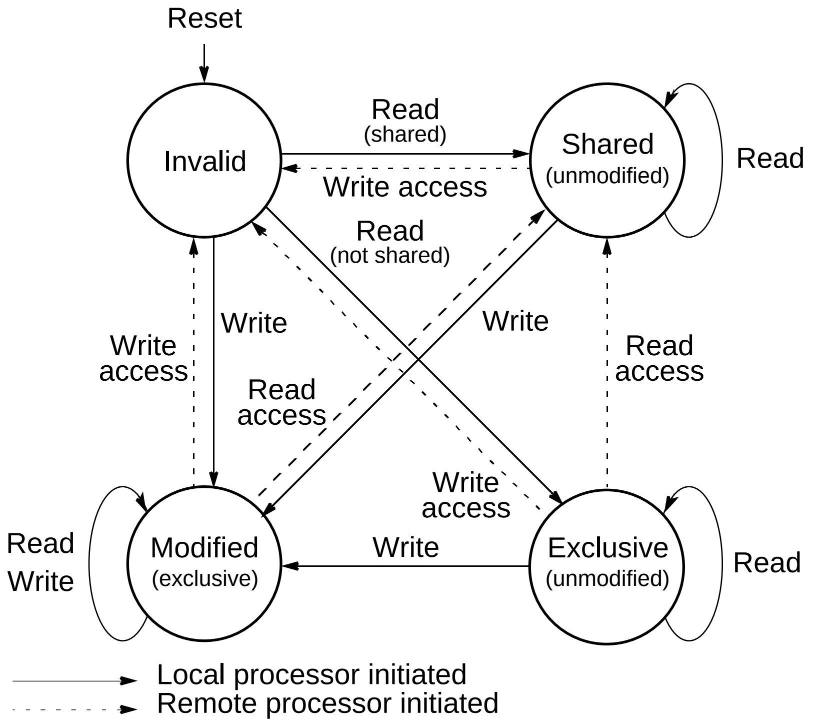
In modified (exclusive) state, processor has altered the contents of the line from that kept in the main memory and hence a valid copy does not even exist in the main memory. It will be necessary to write back the line before any other cache can use the line.

Lines enter the invalid state by being invalidated by other processors, i.e. this is an invalidate protocol.

Slide 509

MESI protocol – major transitions

(Note: there are variations)



Reset

Invalid

Shared
(unmodified)

Modified
(exclusive)

Exclusive
(unmodified)

Read
(shared)

Write access

Read
(not shared)

Write

Write

Write
access

Read
access

Read
access

Write
access

Write

Read

Read

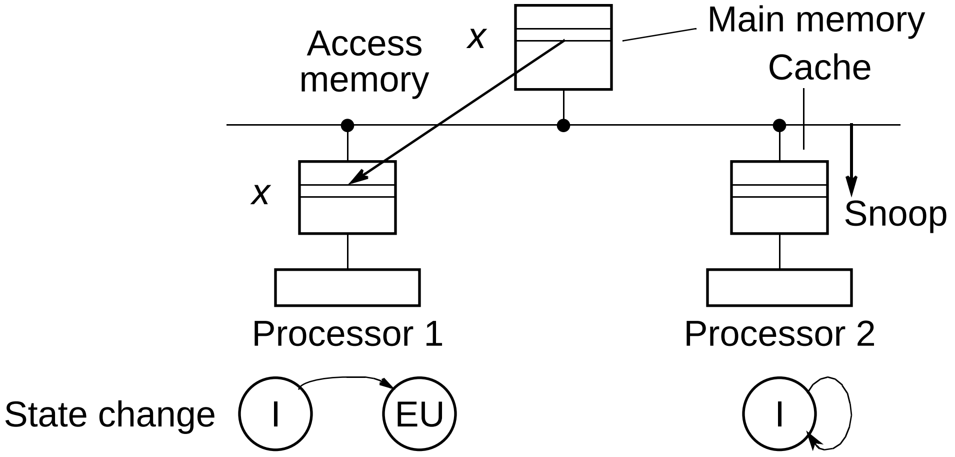
Read
Write

Local processor initiated

Remote processor initiated

Slide 510

Example Sequence of MESI Protocol State Changes



Access
memory

x

Main memory

Cache

x

Snoop

Processor 1

Processor 2

State change

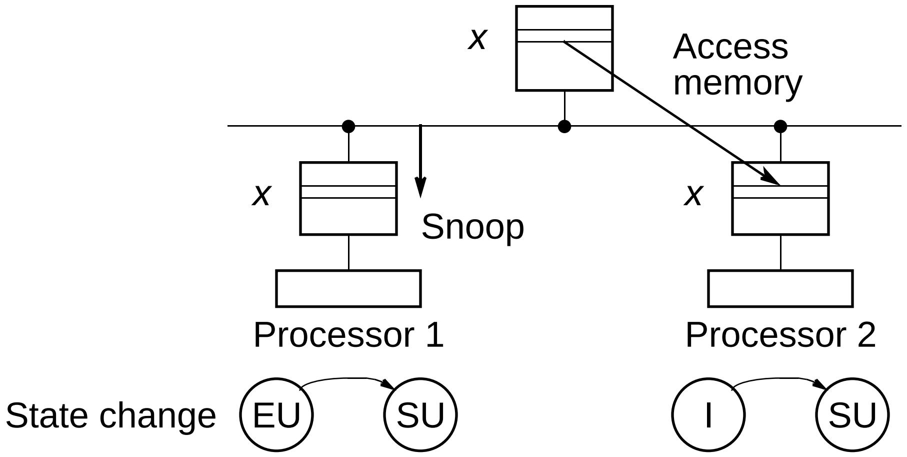
I

EU

I

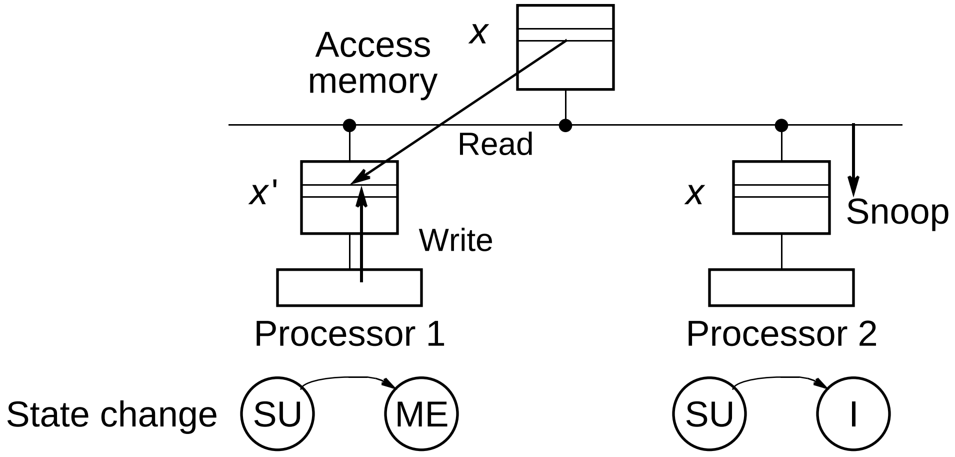
(a) Processor 1
reads *x*

Slide 511



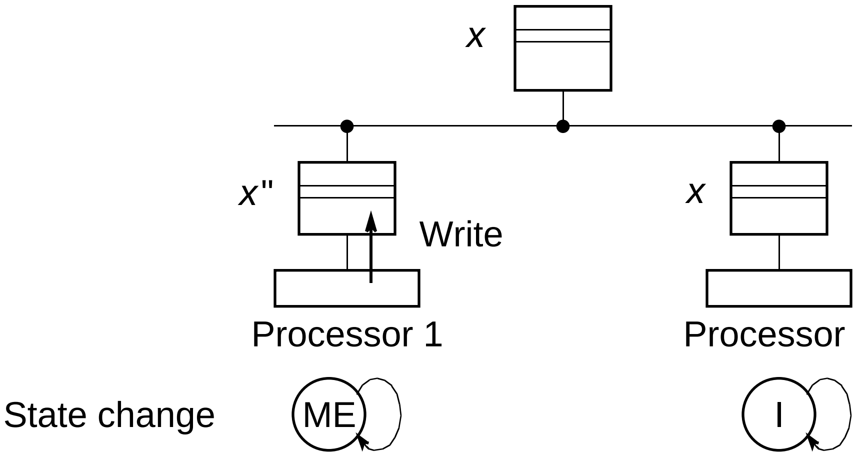
(b) Processor 2 reads *x*

Slide 512



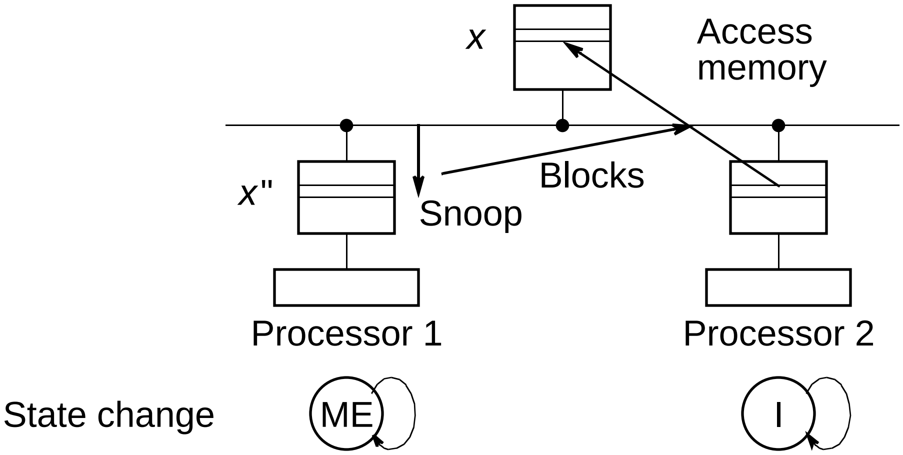
(c) Processor 1 writes to *x*

“Read with intent to modify” (RWITM) sequence initiated.



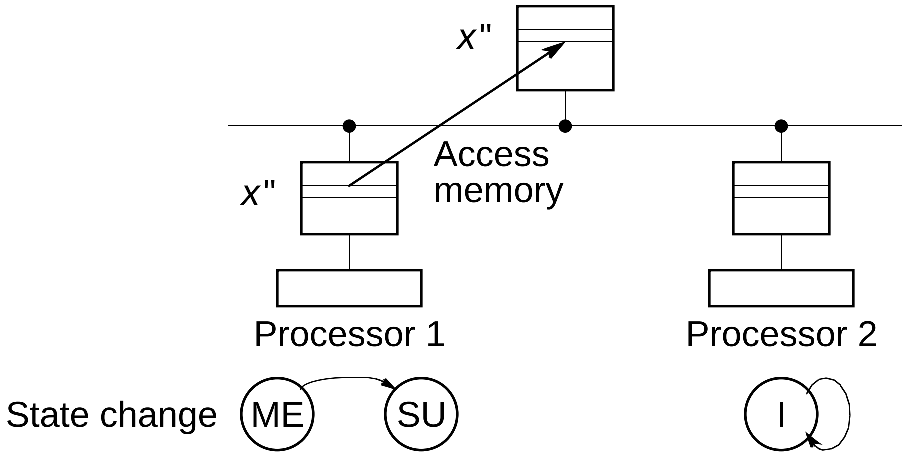
(d) Processor 1
writes to *x*

MESI protocol state changes from exclusive ownership to shared



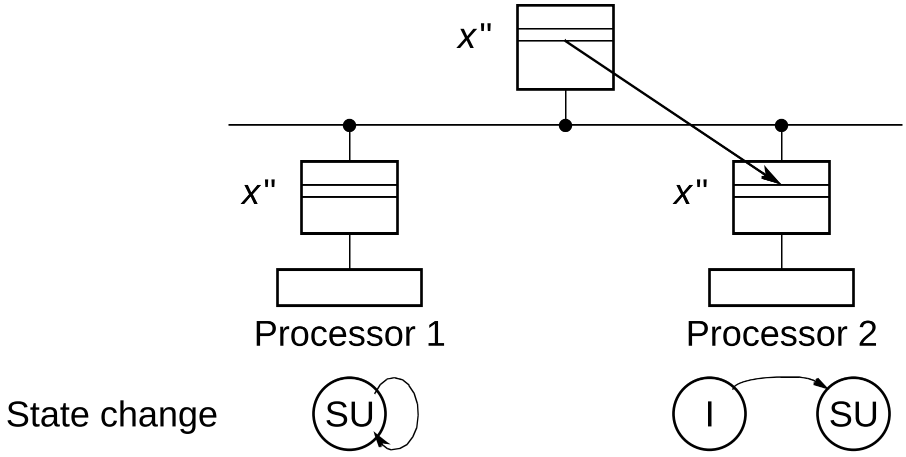
(a) Processor 2
reads *x*

Slide 515



(b) Processor 1
writes back x"

Slide 516



(c) Processor 2
reads *x*" from memory

Slide 517

Performance of Single Bus Network

A key factor in any interconnection network is the bandwidth - the average number of requests accepted in a bus cycle.

Bandwidth and other performance figures can be found by one of four basic techniques:

1. Using analytical probability techniques.
2. Using analytical Markov queuing techniques.
3. By simulation.
4. By measuring an actual system performance.

Slide 518

Probabilistic Techniques

Principal assumptions:

1. The system is synchronous and processor requests are only generated at the beginning of a bus cycle.

2. All processor requests are random and independent of each other.

3. Requests which are not accepted are rejected, and requests generated in the next cycle are independent of rejected requests generated in previous cycles.

Slide 519

Assumption 2

Ignores characteristic that programs normally exhibit referential locality. However, requests from different processors normally independent.

Slide 520

Assumption 3

Rejected requests are ignored and not queued for the next cycle. This assumption is not generally true. Normally when a processor request is rejected in one cycle, the same request will be resubmitted in the next cycle. However, the assumption substantially simplifies the analysis and makes very little difference to the results.

Slide 521

Bandwidth

Probability that processor makes (random) request for memory = r.

Probability that the processor does not make a request = $1 - r$.

Probability that no processors make a request for memory = $(1 - r)^p$ where there are p processors.

Probability that one or more processors make a request= $1-(1-r)^p$.

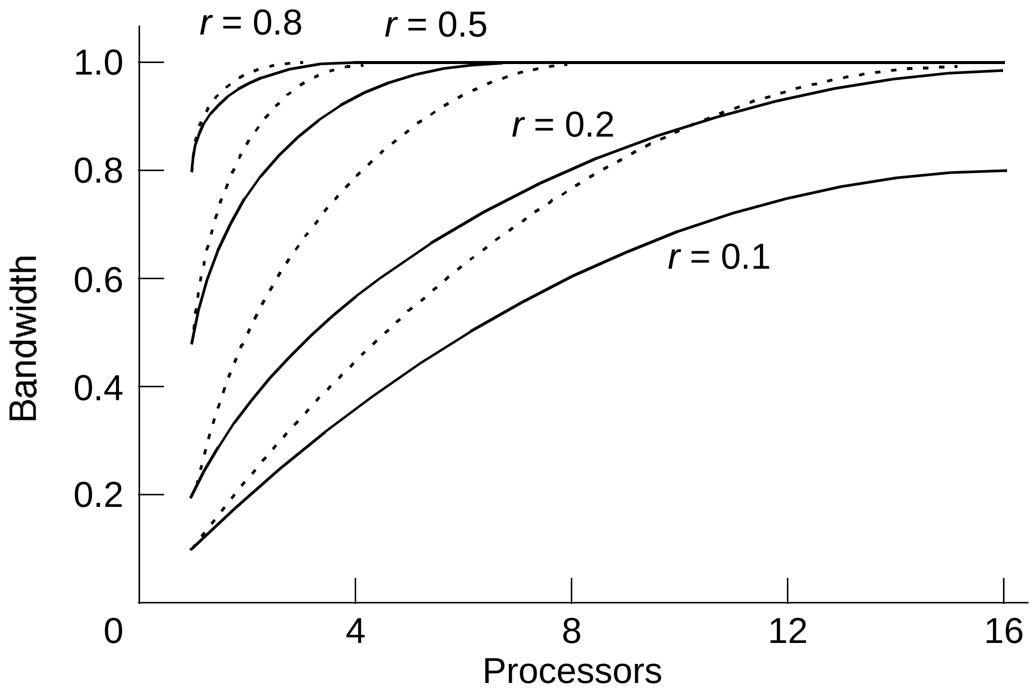
Since only one request can be accepted at a time in a single bus system, the average number of requests accepted in each arbitration cycle (the bandwidth, BW) is given by:

$$BW = 1 - (1 - r)^p$$

Slide 522

Bandwidth of a single bus system

(· · · using more accurate rate adjusted equations, see textbook)



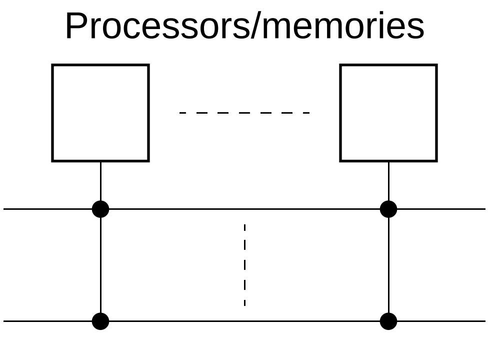
Key Observation,

Bus saturates - at about 8 processors with r = 0.5.

Not be that bad with cache memory as then r is much less.

Still, a single bus is only suitable for a small system.

Multiple buses



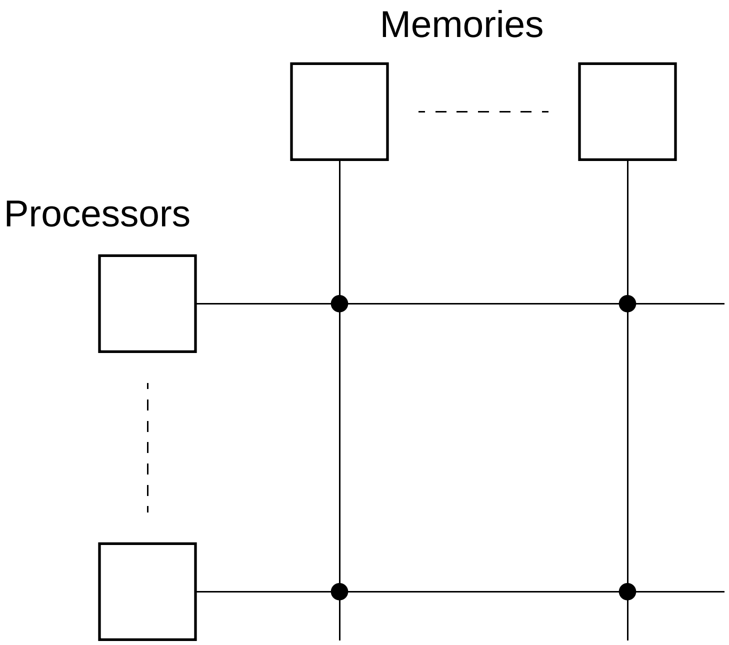
To improve the performance over a single bus system.

Costly

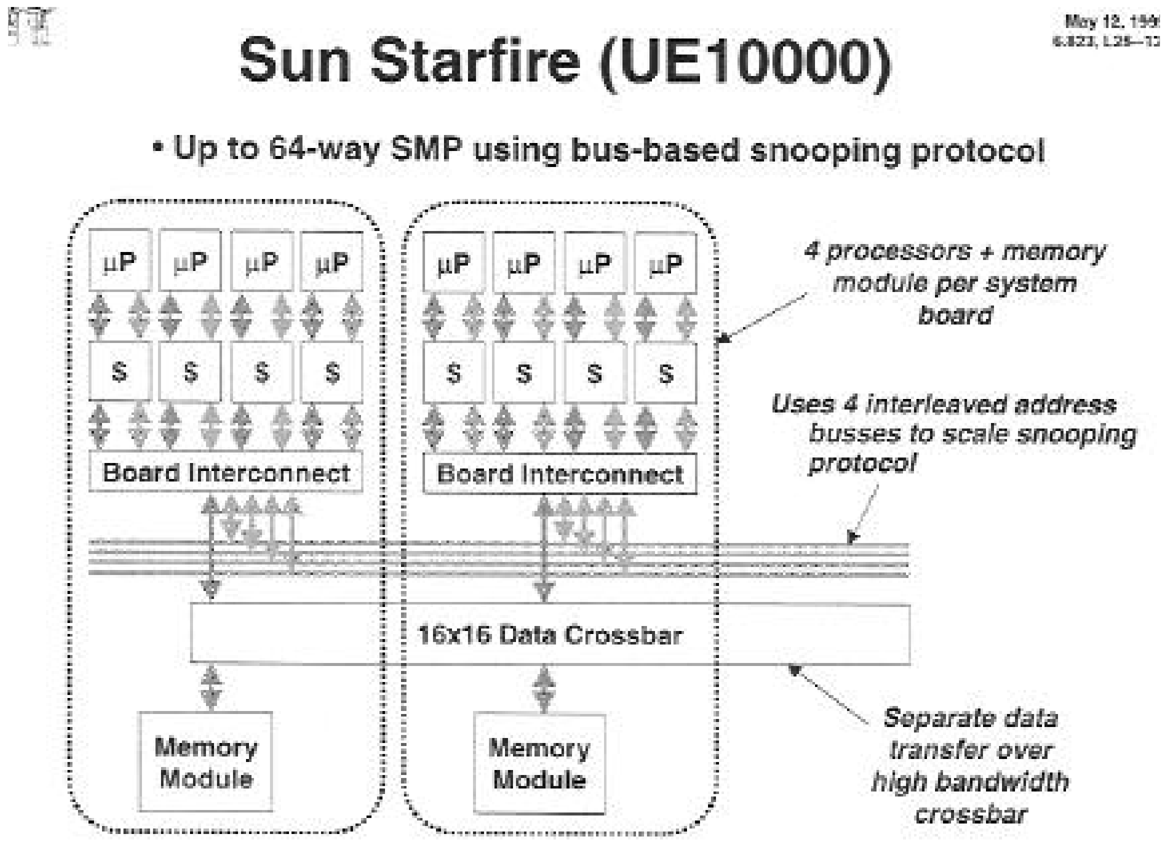
Rarely used - one example: 4 buses used in SUN Starfire UE10000 system for extending the cache snoop protocol.

Slide 525

Crossbar switch



Slide 526



May 12, 199
6.823, L25—13

Sun Starfire (UE10000)

- Up to 64-way SMP using bus-based snooping protocol

4 processors + memory module per system board

Uses 4 interleaved address busses to scale snooping protocol

Separate data transfer over high bandwidth crossbar

Slide 527

Programming Multiprocessors

In all cases, objective is to have each processor executing a program as much as possible simultaneously.

Need to decompose the problem into separate parts that can be done by different processors together.

Generally data may need to pass between the parts and the parts may need to be synchronized at times.

Slide 528

Several Alternatives for Programming Shared Memory Multiprocessors:

Using:

- Threads (Pthreads, Java, ..) in which the programmer decomposes the program into individual parallel sequences, each being thread, and each being able to access variables declared outside the threads.
- A sequential programming language with preprocessor compiler directives to declare shared variables and specify parallelism. Example OpenMP - industry standard
- A sequential programming language with user-level libraries to declare and access shared variables.
- A parallel programming language with syntax for parallelism, in which the compiler creates the appropriate executable code for each processor (not now common)
- A sequential programming language and ask a parallelizing compiler to convert it into parallel executable code. - also not now common

Slide 529

Sequential Consistency

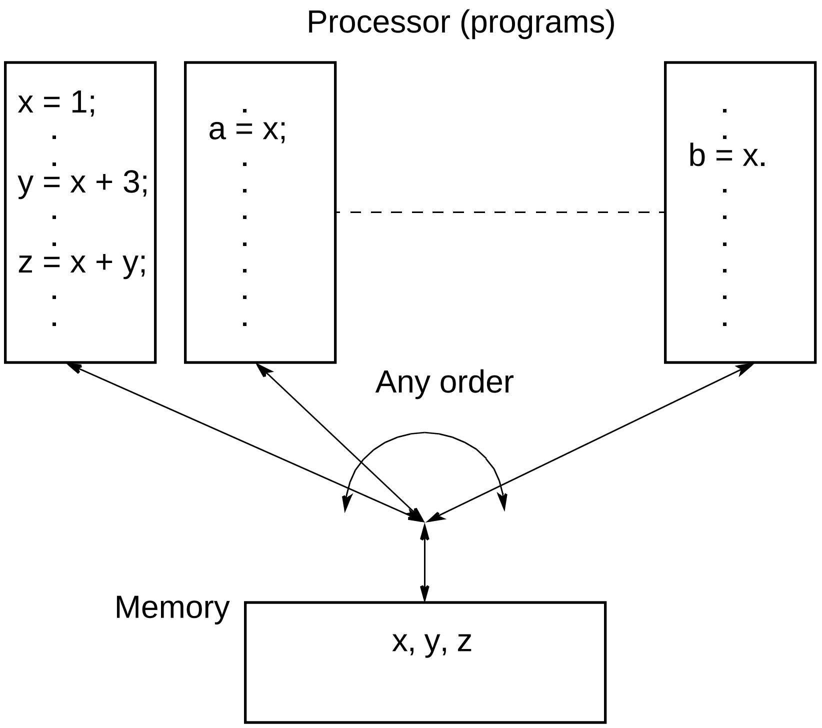
Mentioned before with regard to superscalar proessor design. Formally defined by Lamport (1979):

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 processors occur in this sequence in the order specified by its program.

i.e. the overall effect of a parallel program is not changed by any arbitrary interleaving of instruction execution in time.

Slide 530

Sequential Consistency



Processor (programs)

```
x = 1;
  .
  .
y = x + 3;
  .
  .
z = x + y;
  .
  .
```

```
  .
a = x;
  .
  .
  .
  .
  .
  .
  .
```

```
  .
  .
b = x.
  .
  .
  .
  .
  .
  .
```

Any order

Memory

x, y, z

Writing a parallel program for a system which is known to be sequentially consistent enables us to reason about the result of the program.

Example

```
Process P1                 Process 2
    .                          .
data = new;                    .
flag = TRUE;                   .
    .                          .
    .                      while (flag != TRUE) { };
    .                      data_copy = data;
    .                          .
```

Expect data_copy to be set to new because we expect the statement data = new to be executed before flag = TRUE and the statement while (flag != TRUE) { } to be executed before data_copy = data. Ensures that process 2 reads new data from another process 1. Process 2 will simple wait for the new data to be produced.

Program Order

Sequential consistency refers to “operations of each individual processor .. occur in the order specified in its program” or program order.

In previous figure, this order is that of the stored machine instructions to be executed.

Slide 533

Compiler optimizations

The order is not necessarily the same as the order of the corresponding high level statements in the source program as a compiler may reorder statements for improved performance. In this case, the term program order will depend upon context, either the order in the souce program or the order in the compiled machine instructions.

Slide 534

High performance processors

Modern processors usually reorder machine instructions internally during execution for increased performance.

This does not alter a multiprocessor being sequential consistency, if the processor only produces the final results in program order (that is, retires values to registers in program order which most processors do).

All multiprocessors will have the option of operating under the sequential consistency model. However, it can severely limit compiler optimizations and processor performance.

Slide 535

Example of processor re-ordering

```
Process P1              Process 2
    .                       .
new = a * b;                .
data = new;                 .
flag = TRUE;                .
    .                       .
    .                   while (flag != TRUE) { };
    .                   data_copy = data;
    .                       .
```

Multiply machine instruction corresponding to new = a * b is issued for execution. The next instruction corresponding to data = new cannot be issued until the multiply has produced its result. However the next statement, flag = TRUE, is completely independent and a clever processor could start this operation before the multiply has completed leading to the sequence:

Slide 536

```
Process P1              Process 2
    .                       .
new = a * b;                .
flag = TRUE;                .
data = new;                 .
    .                       .
    .                   while (flag != TRUE) { };
    .                   data_copy = data;
    .                       .
```

Now the while statement might occur before new is assigned to data, and the code would fail.

All multiprocessors will have the option of operating under the sequential consistency model, i.e. not reorder the instructions and forcing the multiply instruction above to complete before starting the subsequent instruction which depend upon its result.

Slide 537

Relaxing read/write orders

Processors may be provided with facilities to be able to relax the consistency in terms of the order of reads and writes of one processor with respect to those of another processor.

Slide 538

Processor consistency

Term is used to describe the particular situation in which individual processors write in program order but the interleaved writes of different processors can appear in different orders. This relaxation would allow some opportunities for improved performance (through buffering and reordering instructions).

To support general relaxed read and write orders, special machine instructions, variously called memory fences or memory barriers, are provided to synchronize the memory operations when necessary in the program.

Slide 539

Examples

Alpha processors

Memory barrier (MB) instruction - waits for all previously issued memory accesses instructions to complete before issuing any new memory operations.

Write memory barrier (WMB) instruction - as MB but only on memory write operations, i.e. waits for all previously issued memory write accesses instructions to complete before issuing any new memory write operations - which means memory reads could be issued after a memory write operation but overtake it and complete before the write operation. (check)

Slide 540

SUN Sparc V9 processors

memory barrier (MEMBAR) instruction with four bits for variations Write-to-read bit prevent any reads that follow it being issued before all writes that precede it have completed. Other: Write-to-write, read-to-read, read-to-write.

IBM PowerPC processor

SYNC instruction - similar to Alpha MB instruction (check differences)

Slide 541

Relaxed memory models

Weak Consistency

Synchronization operations are used by the programmer whenever it is necessary to enforce sequency consistency. The compiler/ processor is allowed in other places to reorder instructions without regard to sequential consistency.

This is quite reasonable model since any accesses to shared data should be provided with synchronization operations (locks etc.).

Slide 542

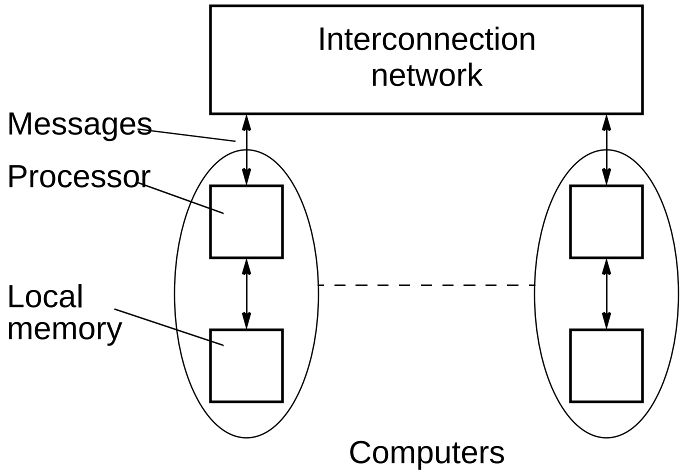
Basic Tenets of Shared Memory Programming

- Protect access to shared data through use of critical sections, locks, semaphores, etc.
- Synchronize processes using barriers

Slide 543

Distributed Memory Multicomputer

Complete computers connected through an interconnection network:



Slide 544

Cluster Computing

Using a group of interconnected computers to solve a problem - usually the motive is faster computation, but other motives include fault tolerance, larger amount of memory available, ...

Cluster computing became interesting when the cost of commody computers became low and their performance increased in the early 1990’s.

Originally terms such as networks of workstations (NOWs), cluster of workstations (COWs) were used.

Slide 545

Background

Using a groups of computers or processors collectively to solve a problem has been around for 40 years.

Parallel computers - computers designed with more than one processor- and parallel programming - programming such computers is a regular topic in computer science curriculum.

What is different now is that all institutions can establish their own parallel computing platform at little cost and the progarmming tools are readily available. Do not need access to supercomputers.

Slide 546

Programming

Involves dividing problem into parts that are intended to be executed simultaneously to solve the problem

Common approach is to use message-passing library routines that are linked to conventional sequential program(s) for message passing.

Problem divided into a number of concurrent processes.

Processes will communicate by sending messages; this will be the only way to distribute data and results between processes.

Slide 547

Message Passing Parallel Programming Software Tools for Clusters

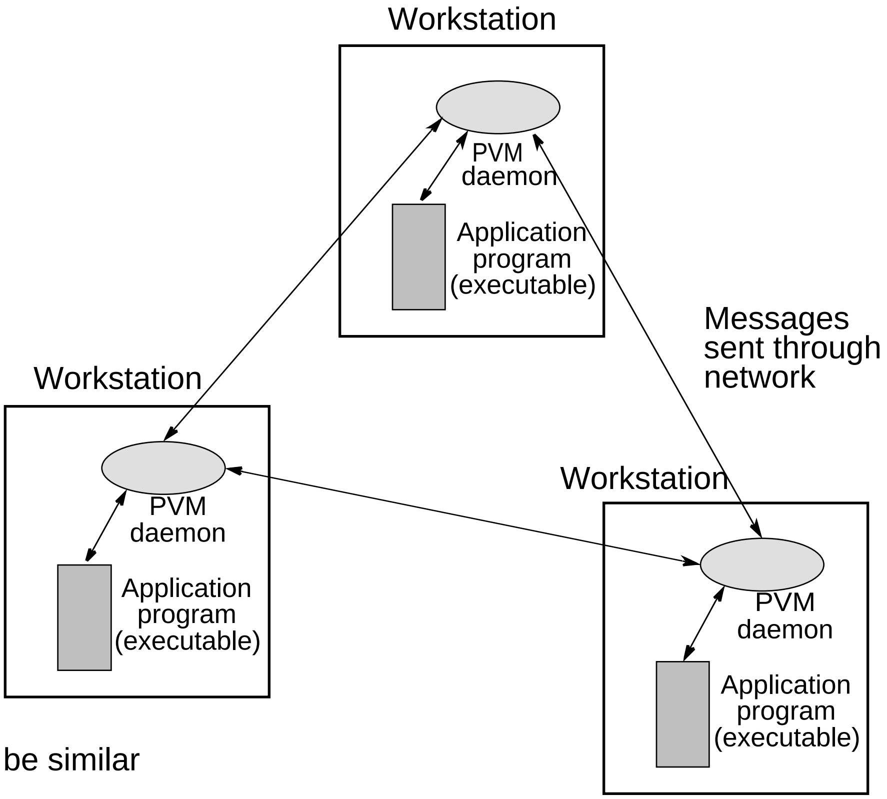
Parallel Virtual Machine (PVM) - developed in late 1980’s. Became very popular.

Message-Passing Interface (MPI) - standard defined in 1990s.

Both provide a set of user-level libraries for message passing. Use with regular programming languages (C, C++, ...).

Slide 548

Message passing between workstations using PVM.


Workstation
PVM daemon
Application program (executable)
Workstation
PVM daemon
Application program (executable)
Workstation
PVM daemon
Application program (executable)
Messages sent through network

MPI can be similar

Slide 549

Distributed Shared Memory

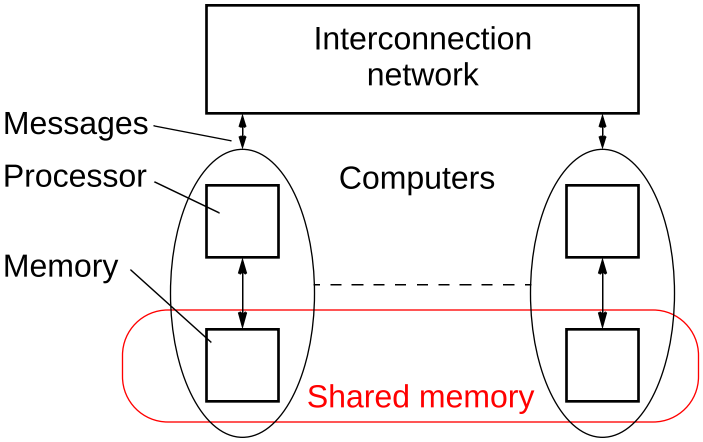
Making the main memory of a cluster of computers look as though it is a single memory with a single address space.

Then can use shared memory programming techniques.

Slide 550

DSM System

Still need messages or mechanisms to get data to processor, but these are hidden from the programmer:



Slide 551

Advantages of DSM

System scalable

Hides the message passing - do not explicitly specific sending messages between processes

Can us simple extensions to sequential programming

Can handle complex and large data bases without replication or sending the data to processes

Slide 552

Disadvantages of DSM

May incur a performance penalty

Must provide for protection against simultaneous access to shared data (locks, etc.)

Slide 553

Methods of Achieving DSM

Hardware - special network interfaces and cache coherence circuits

Software - adding a software layer between the operating system and the application

Slide 554

Software DSM Implementation

- Page based - Using the system’s virtual memory
- Shared variable approach
- Object based- Shared data within collection of objects.

  Access to shared data through object oriented discipline (ideally)

Slide 555

Performance Issues

Achieving Consistent Memory

- Data altered by one processor visible to other processors
- Similar problem to multiprocessor cache coherence
- Multiple writer protocols
- False Sharing - Different data within same page accessed by different processors (if page based)

Slide 556

Consistency Models

- Strict Consistenc - Processors sees most recent update, i.e. read returns the most recent wrote to location
- Sequential Consistency - Result of any execution same as an interleaving of individual programs
- Relaxed Consistency- Delay making write visible to reduce messages
- Release Consistency - programmer must use synchronization operators, acquire and release
- Lazy Release Consistency - update only done at time of acquire

Slide 557

Some Software DSM Systems

Treadmarks

JIAJIA

Adsmith object based (C++ library routines) - we have this installed on our cluster

Slide 558

Adsmith

- User-level libraries that create distributed shared memory system on a cluster.
- Object based DSM - memory seen as a collection of objects that can be shared among processes on different processors.
- Written in C++
- Built on top of pvm
- Freely available - installed on UNCC cluster

User writes application programs in C or C++ and calls Adsmith routines for creation of shared data and control of its access.
If required, can also call pvm routines in same program.

Slide 559

Another course deal with programming multiprocessor systems (parallel programming):

ITCS 5145 Parallel Programming

Slide 560

THE

END