## 1 Shared Memory Architecture

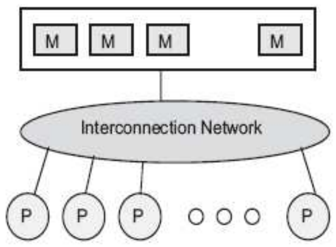


Figure 1: Shared memory systems.

- Shared memory systems form a major category of multiprocessors. In this category, all processors share a global memory (See Fig. 1).
- Communication between tasks running on different processors is performed through writing to and reading from the global memory.
- All interprocessor coordination and synchronization is also accomplished via the global memory.
- Two main problems need to be addressed when designing a shared memory system:
  1. *performance degradation due to contention.* Performance degradation might happen when multiple processors are trying to access the shared memory simultaneously. A typical design might use caches to solve the contention problem.
  2. *coherence problems.* Having multiple copies of data, spread throughout the caches, might lead to a coherence problem. The copies in the caches are coherent if they are all equal to the same value. However, if one of the processors writes over the value of one of the copies, then the copy becomes inconsistent because it no longer equals the value of the other copies.
- Scalability remains the main drawback of a shared memory system.

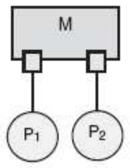


Figure 2: Shared memory via two ports.

## 1.1 Classification of Shared Memory Systems

- The simplest shared memory system consists of one memory module (M) that can be accessed from two processors P1 and P2 (see Fig. 2).
  - Requests arrive at the memory module through its two ports. An arbitration unit within the memory module passes requests through to a memory controller.
  - If the memory module is not busy and a single request arrives, then the arbitration unit passes that request to the memory controller and the request is satisfied.
  - The module is placed in the busy state while a request is being serviced. If a new request arrives while the memory is busy servicing a previous request, the memory module sends a wait signal, through the memory controller, to the processor making the new request.
  - In response, the requesting processor may hold its request on the line until the memory becomes free or it may repeat its request some time later.
  - If the arbitration unit receives two requests, it selects one of them and passes it to the memory controller. Again, the denied request can be either held to be served next or it may be repeated some time later.

### 1.1.1 Uniform Memory Access (UMA)

- In the UMA system a shared memory is accessible by all processors through an interconnection network in the same way a single processor accesses its memory.
- All processors have equal access time to any memory location. The interconnection network used in the UMA can be a single bus, multiple

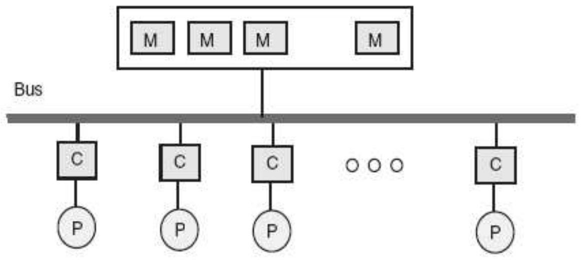


Figure 3: Bus-based UMA (SMP) shared memory system.

buses, or a crossbar switch.

- Because access to shared memory is balanced, these systems are also called SMP (symmetric multiprocessor) systems. Each processor has equal opportunity to read/write to memory, including equal access speed.
  - A typical bus-structured SMP computer, as shown in Fig. 3, attempts to reduce contention for the bus by fetching instructions and data directly from each individual cache, as much as possible.
  - In the extreme, the bus contention might be reduced to zero after the cache memories are loaded from the global memory, because it is possible for all instructions and data to be completely contained within the cache.
- This memory organization is the most popular among shared memory systems. Examples of this architecture are Sun Starfire servers, HP V series, and Compaq AlphaServer GS, Silicon Graphics Inc. multiprocessor servers.

### 1.1.2 Nonuniform Memory Access (NUMA)

- In the NUMA system, each processor has part of the shared memory attached (see Fig. 4).
- The memory has a single address space. Therefore, any processor could access any memory location directly using its real address. However, the access time to modules depends on the distance to the processor. This results in a nonuniform memory access time.

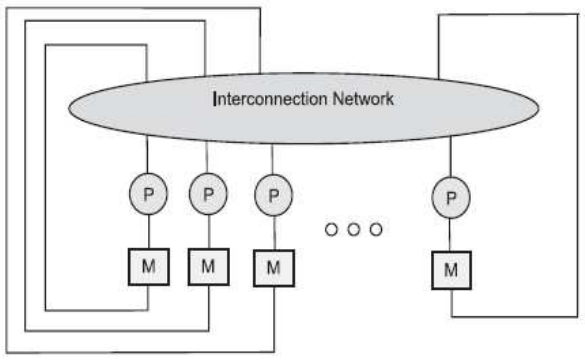


Figure 4: NUMA shared memory system.

- A number of architectures are used to interconnect processors to memory modules in a NUMA. Among these are the tree and the hierarchical bus networks.
- Examples of NUMA architecture are BBN TC-2000, SGI Origin 3000, and Cray T3E.

#### 1.1.3 Cache-Only Memory Architecture (COMA)

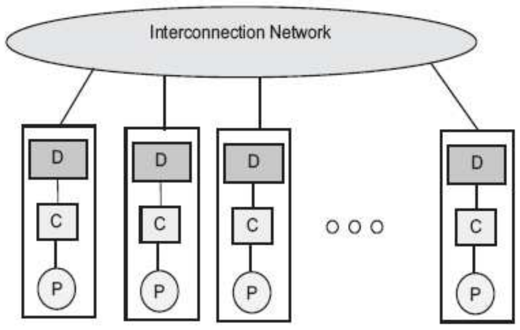


Figure 5: COMA shared memory system.

- Similar to the NUMA, each processor has part of the shared memory in the COMA (see Fig. 5).

- However, in this case the shared memory consists of cache memory. A COMA system requires that data be migrated to the processor requesting it.
- There is no memory hierarchy and the address space is made of all the caches.
- There is a cache directory (D) that helps in remote cache access.
- The Kendall Square Research's KSR-1 machine is an example of such architecture.

## 1.2 Bus-based Symmetric Multiprocessors

- Shared memory systems can be designed using bus-based or switch-based interconnection networks.
- The simplest network for shared memory systems is the bus.
- The bus/cache architecture facilitates the need for expensive multi-ported memories and interface circuitry as well as the need to adopt a message-passing paradigm when developing application software.
- However, the bus may get saturated if multiple processors are trying to access the shared memory (via the bus) simultaneously.
- High-speed caches connected to each processor on one side and the bus on the other side mean that local copies of instructions and data can be supplied at the highest possible rate.
- If the local processor finds all of its instructions and data in the local cache, we say the hit rate is 100%. The miss rate of a cache is the fraction of the references that cannot be satisfied by the cache, and so must be copied from the global memory, across the bus, into the cache, and then passed on to the local processor.
- Hit rates are determined by a number of factors, ranging from the application programs being run to the manner in which cache hardware is implemented.
  - A processor goes through a duty cycle, where it executes instructions a certain number of times per clock cycle.

– Typically, individual processors execute less than one instruction per cycle, thus reducing the number of times it needs to access memory.

– Subscalar processors execute less than one instruction per cycle, and superscalar processors execute more than one instruction per cycle.

– In any case, we want to minimize the number of times each local processor tries to use the central bus. Otherwise, processor speed will be limited by bus bandwidth.

– We define the variables for hit rate, number of processors, processor speed, bus speed, and processor duty cycle rates as follows:

  ∗ $N$ =number of processors;

  ∗ $h$ =hit rate of each cache, assumed to be the same for all caches;

  ∗ $(1 - h)$ =miss rate of all caches;

  ∗ $B$ = bandwidth of the bus, measured in cycles/second;

  ∗ $I$ = processor duty cycle, assumed to be identical for all processors, in fetches/cycle;

  ∗ $V$ = peak processor speed, in fetches/second.

– The effective bandwidth of the bus is $B * I$ fetches/second.

  ∗ If each processor is running at a speed of $V$, then misses are being generated at a rate of $V * (1 - h)$.

  ∗ For an $N$-processor system, misses are simultaneously being generated at a rate of $N * (1 - h) * V$.

  ∗ This leads to saturation of the bus when $N$ processors simultaneously try to access the bus. That is, $N*(1-h)*V \leq B*I$.

– The maximum number of processors with cache memories that the bus can support is given by the relation,

$$N \leq \frac{B * I}{(1 - h) * V} \tag{1}$$

– Example: Suppose a shared memory system is

  ∗ constructed from processors that can execute $V = 107$ instructions/s and the processor duty cycle $I = 1$.

  ∗ the caches are designed to support a hit rate of 97%,

  ∗ the bus supports a peak bandwidth of $B = 106$ cycles/s.

- $*$ Then, $(1-h) = 0.03$, and the maximum number of processors $N$ is
$$N \leq \frac{106}{(0.03 * 107)} = 3.33$$
- $*$ Thus, the system we have in mind can support only three processors!
- $*$ We might ask what hit rate is needed to support a 30-processor system. In this case,
$$h = 1 - \frac{B * I}{N * V} = 1 - \frac{106 * 1}{30 * 107} = 1 - \frac{1}{300}$$
so for the system we have in mind, $h = 0.9967$. Increasing $h$ by 2.8% results in supporting a factor of ten more processors.

## 1.3 Basic Cache Coherency Methods

- Multiple copies of data, spread throughout the caches, lead to a coherence problem among the caches. The copies in the caches are coherent if they all equal the same value.

- However, if one of the processors writes over the value of one of the copies, then the copy becomes inconsistent because it no longer equals the value of the other copies.

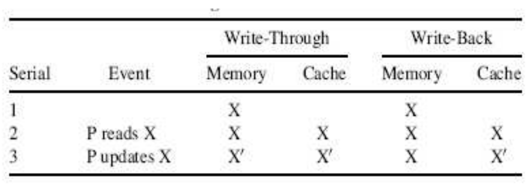
- If data are allowed to become inconsistent (incoherent), incorrect results will be propagated through the system, leading to incorrect final results.

- Cache coherence algorithms are needed to maintain a level of consistency throughout the parallel system.

- Cache coherence schemes can be categorized into two main categories: *snooping protocols* and *directory-based protocols.*

  - Snooping protocols are based on watching bus activities and carry out the appropriate coherency commands when necessary.
  - In cases when the broadcasting techniques used in snooping protocols are unpractical, coherence commands need to be sent to only those caches that might be affected by an update.
  - This is the idea behind directory-based protocols. Cache coherence protocols that somehow store information on where copies of blocks reside are called directory schemes.

### 1.3.1 Cache-Memory Coherence

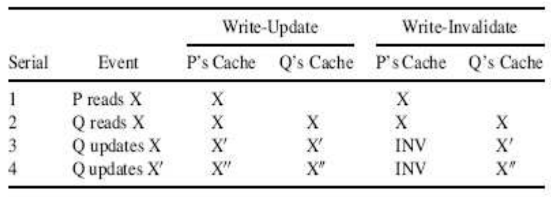


| | | Write-Through | | Write-Back | |
|---|---|---|---|---|---|
| Serial | Event | Memory | Cache | Memory | Cache |
| 1 | | X | | X | |
| 2 | P reads X | X | X | X | X |
| 3 | P updates X | X′ | X′ | X | X′ |

Figure 6: Write-Through vs. Write-Back.

- In a single cache system, coherence between memory and the cache is maintained using one of two policies:
  1. write-through,
  2. write-back (see Fig. 6).
- When a task running on a processor P requests the data in memory location X, for example, the contents of X are copied to the cache, where it is passed on to P.
- When P updates the value of X in the cache, the other copy in memory also needs to be updated in order to maintain consistency.
- In *write-through*, the memory is updated every time the cache is updated,
- while in *write-back*, the memory is updated only when the block in the cache is being replaced.

### 1.3.2 Cache-Cache Coherence



| | | Write-Update | | Write-Invalidate | |
|---|---|---|---|---|---|
| Serial | Event | P's Cache | Q's Cache | P's Cache | Q's Cache |
| 1 | P reads X | X | | X | |
| 2 | Q reads X | X | X | X | X |
| 3 | Q updates X | X′ | X′ | INV | X′ |
| 4 | Q updates X′ | X″ | X″ | INV | X″ |

Figure 7: Write-Update vs. Write-Invalidate.

- In multiprocessing system, when a task running on processor P requests the data in global memory location X, for example, the contents of X are copied to processor P's local cache, where it is passed on to P.
- Now, suppose processor Q also accesses X. What happens if Q wants to write a new value over the old value of X?
- There are two fundamental cache coherence policies:
  1. write-invalidate,
  2. write-update (see Fig. 7).
- *Write-invalidate* maintains consistency by reading from local caches until a write occurs. When any processor updates the value of X through a write, posting a dirty bit for X invalidates all other copies.
  - For example, processor Q invalidates all other copies of X when it writes a new value into its cache. This sets the dirty bit for X.
  - Q can continue to change X without further notifications to other caches because Q has the only valid copy of X.
  - However, when processor P wants to read X, it must wait until X is updated and the dirty bit is cleared.
- *Write-update* maintains consistency by immediately updating all copies in all caches.
  - All dirty bits are set during each write operation. After all copies have been updated, all dirty bits are cleared.

### 1.3.3 Shared Memory System Coherence

- The four combinations to maintain coherence among all caches and global memory are:
  1. Write-update and write-through;
  2. Write-update and write-back;
  3. Write-invalidate and write-through;
  4. Write-invalidate and write-back.
- If we permit a *write-update and write-through* directly on global memory location X, the bus would start to get busy and ultimately all processors would be idle while waiting for writes to complete.

- In *write-update and write-back*, only copies in all caches are updated. On the contrary, if the write is limited to the copy of X in cache Q, the caches become inconsistent on X. Setting the dirty bit prevents the spread of inconsistent values of X, but at some point, the inconsistent copies must be updated.

## 1.4 Shared Memory Programming

- Shared memory parallel programming is perhaps the easiest model to understand because of its similarity with operating systems programming and general multiprogramming.
- Shared memory programming is done through some extensions to existing programming languages, operating systems, and code libraries.
- Shared address space programming paradigms can vary on mechanisms for *data sharing*, *concurrency models*, and *support for synchronization.*
- Process based models assume that all data associated with a process is private, by default, unless otherwise specified (using UNIX system calls such as **shmget** and **shmat**).
- While this is important for ensuring protection in multiuser systems, it is not necessary when multiple concurrent aggregates are cooperating to solve the same problem.
- The overheads associated with enforcing protection domains make processes less suitable for parallel programming.
- In contrast, lightweight processes and threads assume that all memory is global. By relaxing the protection domain, lightweight processes and threads support much faster manipulation.
- Directive based programming models extend the threaded model by facilitating creation and synchronization of threads.
- In a shared memory parallel program, there must exist three main programming constructs:
  1. task creation,
  2. communication,
  3. synchronization.

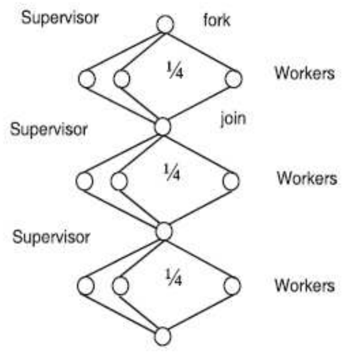


Figure 8: Supervisor-workers model used in most parallel applications on shared memory systems.

### 1.4.1 Task Creation

- At the *large-grained* level, a shared memory system can provide traditional timesharing.
  - Each time a new process is initiated, idle processors are supplied to run the new process.
  - If the system is loaded, the processor with least amount of work is assigned the new process.
  - These large-grained processes are often called heavy weight tasks because of their high overhead.
  - A heavy weight task in a multitasking system like UNIX consists of page tables, memory, and file description in addition to program code and data.
  - These tasks are created in UNIX by invocation of fork, exec, and other related UNIX commands. This level is best suited for heterogeneous tasks.
- At the *fine-grained level*, lightweight processes makes parallelism within a single application practical, where it is best suited for homogeneous tasks.
  - At this level, an application is a series of fork-join constructs. This pattern of task creation is called the supervisor-workers model, as shown in Fig. 8.

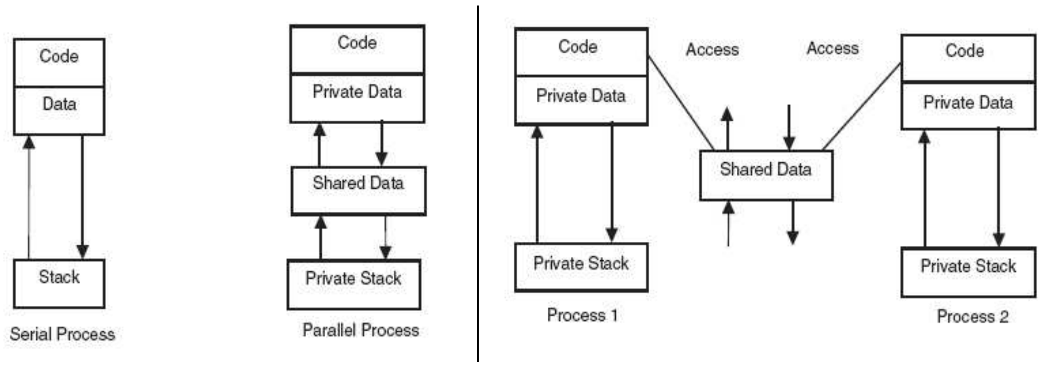


Figure 9: Serial process vs. parallel process and two parallel processes communicate using the shared data segment.

### 1.4.2 Communication

- In general, the address space on an executing process has three segments called the text, data, and stack (see Fig. 9a).
    - The text is where the binary code to be executed is stored;
    - the data segment is where the program's data are stored;
    - the stack is where activation records and dynamic data are stored.
- The data and stack segments expand and contract as the program executes. Therefore, a gap is purposely left in between the data and stack segments.
- Serial processes are assumed to be mutually independent and do not share addresses. The code of each serial process is allowed to access data in its own data and stack segments only.
- A parallel process is similar to the serial process plus an additional shared data segment. This shared area is allowed to grow and is placed in the gap between private data and stack segments.
- Communication among parallel processes can be performed by writing to and reading from shared variables in the shared data segments as shown in Fig. 9b.

### 1.4.3 Synchronization

- Synchronization is needed to protect shared variables by ensuring that they are accessed by only one process at a given time (mutual exclu-

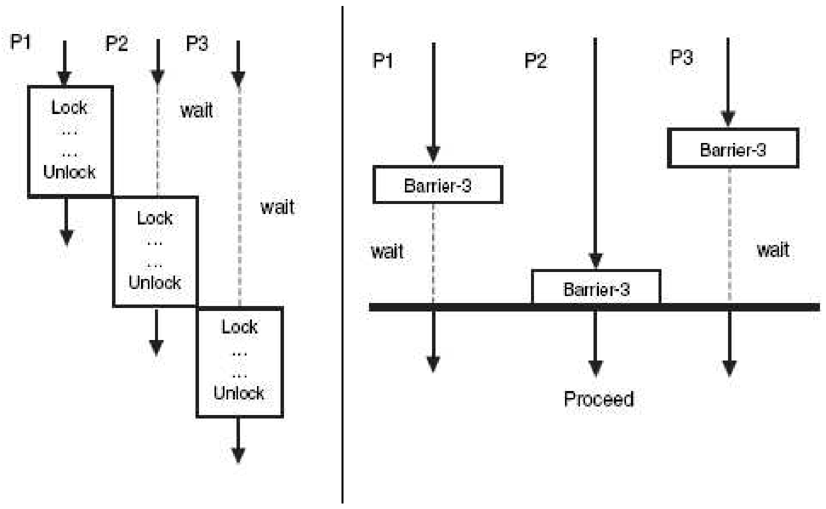


Figure 10: Locks and barriers.

sion).

- They can also be used to coordinate the execution of parallel processes and synchronize at certain points in execution.
- There are two main synchronization constructs in shared memory systems:
  1. locks
  2. barriers.
- Figure 10a shows three parallel processes using locks to ensure mutual exclusion. Process P2 has to wait until P1 unlocks the critical section; similarly P3 has to wait until P2 issues the unlock statement.
- In Figure 10b, P3 and P1 reach their barrier statement before P2, and they have to wait until P2 reaches its barrier. When all three reach the barrier statement, they all can proceed.

### 1.4.4 Thread Basics

- A thread is a single stream of control in the flow of a program. We initiate threads with a simple example:
- What are threads? Consider the following code segment that computes the product of two dense matrices of size $n \times n$.

```c
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            dot_product(get_row(a, row),
                        get_col(b, col));
```

  - The for loop in this code fragment has $n^2$ iterations, each of which can be executed independently.
  - Such an independent sequence of instructions is referred to as a thread.
  - In the example presented above, there are $n^2$ threads, one for each iteration of the for-loop.
  - Since each of these threads can be executed independently of the others, they can be scheduled concurrently on multiple processors.

We can transform the above code segment as follows:

```c
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            create_thread(dot_product(get_row(a, row),
                                      get_col(b, col)));
```

  - Here, we use a function, *create_thread*, to provide a mechanism for specifying a C function as a thread.
  - The underlying system can then schedule these threads on multiple processors.

- **Logical Memory Model of a Thread**; To execute the code fragment in the previous example on multiple processors,
  - each processor must have access to matrices a, b, and c. This is accomplished via a shared address space.
  - All memory in the logical machine model of a thread is globally accessible to every thread. However, since threads are invoked as function calls, the stack corresponding to the function call is generally treated as being local to the thread.
  - This is due to the liveness considerations of the stack. Since threads are scheduled at runtime (and no a priori schedule of their execution can be safely assumed), it is not possible to determine which stacks are live.

  - Therefore, it is considered poor programming practice to treat stacks (thread-local variables) as global data. This implies a logical machine model, where memory modules M hold thread-local (stack allocated) data.

- While this logical machine model gives the view of an equally accessible address space, physical realizations of this model deviate from this assumption.

- In distributed shared address space machines such as the Origin 2000, the cost to access a physically local memory may be an order of magnitude less than that of accessing remote memory.

- Even in architectures where the memory is truly equally accessible to all processors, the presence of caches with processors skews memory access time.

- Issues of locality of memory reference become important for extracting performance from such architectures.