High Performance Platforms

Salvatore Orlando
Scope of Parallelism: the roadmap towards HPC

- Conventional architectures are coarsely comprised of
  - processor, memory system, I/O
- Each of these components presents significant performance bottlenecks.
- It is important to understand each of these performance bottlenecks to reduce their effects
- **Parallelism** addresses each of these components in significant ways to improve performance and reduce the bottlenecks

- Applications utilize different aspects of parallelism, e.g.,
  - data intensive applications utilize high aggregate memory bandwidth
  - server applications utilize high aggregate network bandwidth
  - scientific applications typically utilize high processing and memory system performance
Implicit Parallelism: Trends in Microprocessor Architectures

• Microprocessor clock speeds have posted impressive gains over the past two decades (two to three orders of magnitude)
  – there are limits to this increase, also due to power consumption

• Higher levels of device integration have made available a large number of transistors
  – the issue is how best to transform these large amount of transistors into computational power

• Single-core processors use these resources in multiple functional units and execute multiple instructions in the same cycle.

• The precise manner in which these instructions are selected and executed provides impressive diversity in architectures.
Pipelining and Superscalar Execution

- Pipelining overlaps various stages of instruction execution to achieve performance
- At a high level of abstraction, an instruction can be executed while the next one is being decoded and the next one is being fetched.
- This is akin to an assembly line for manufacture of cars.

A simple pipeline
**IF:** Instruction fetch
**ID:** Instruction decode
**EX:** Execution and address calculation
**MEM:** Memory access
**WB:** Write back of registers

![Diagram of pipelining stages](image-url)
Pipelining and Superscalar Execution

• Pipelining, however, has several limitations.

• The speed of a pipeline is eventually limited by the slowest stage.
• For this reason, conventional processors rely on very deep pipelines (20 stage pipelines in some Pentium processors).

• However, in typical program traces, every 5-6 instructions, there is a conditional jump! This requires very accurate branch prediction.
  – The penalty of a misprediction grows with the depth of the pipeline, since a larger number of instructions will have to be flushed.

• Data and resource dependency may stall the pipeline
Pipelining and Superscalar Execution

- We can use multiple pipelines
- Superscalar processors simultaneously dispatch multiple instructions to redundant functional units on the processor
  - Instructions are still issued from a sequential instruction stream
  - CPU hardware dynamically checks for data dependencies between instructions at run time (versus software checking at compile time)
  - The CPU issues multiple instructions per clock cycle
- The question then becomes how to select from the stream the instructions to be issued on these pipelines
- Compilers can help in generation code that can be easily parallelized
Superscalar Execution: An Example of two-way execution

- A pipeline of 4 stages
- Instructions with different addressing modes: register and memory

1. load R1, @1000
2. load R2, @1008
3. add R1, @1004
4. add R2, @100C
5. add R1, R2
6. store R1, @2000

(i)

(a) Three different code fragments for adding a list of four numbers.

Instruction cycles

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>ID</td>
<td>OF</td>
<td>load R1, @1000</td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>OF</td>
<td>load R2, @1008</td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>OF</td>
<td>add R1, @1004</td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>OF</td>
<td>add R2, @100C</td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>NA</td>
<td>add R1, R2</td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>NA</td>
<td>store R1, @2000</td>
<td></td>
</tr>
</tbody>
</table>

(b) Execution schedule for code fragment (i) above.

IF: Instruction Fetch
ID: Instruction Decode
OF: Operand Fetch
E: Instruction Execute
WB: Write-back
NA: No Action

(b) Execution schedule for code fragment (i) above.

(c) Hardware utilization trace for schedule in (b).

NA means that the pipeline stalls, waiting for the availability of data (data dependency due to registers R1 and R2)
Superscalar Execution: An Example

- In the above example, there is some wastage of resources due to data dependencies.
- The example also illustrates that different instruction mixes with identical semantics can take significantly different execution time.
Superscalar Execution

- Scheduling of instructions is determined by a number of factors:
  - **True Data Dependency**: The result of one operation is an input to the next.
  - **Resource Dependency**: Two operations require the same resource.
  - **Branch Dependency**: Scheduling instructions across conditional branch statements cannot be done deterministically a-priori.

- The scheduler, a piece of hardware, looks at a large number of instructions in an instruction queue and selects appropriate number of instructions to execute concurrently based on these factors.

- The complexity of this hardware is an important constraint on superscalar processors.
Superscalar Execution: Issue Mechanisms

- In the simpler model, instructions can be issued only in the order in which they are encountered.
  - That is, if the second instruction cannot be issued because it has a data dependency with the first, only one instruction is issued in the cycle.
  - This is called **in-order** issue.
  - The compiler may statically reschedule instructions to increase the chance of parallel execution.
- In a more aggressive model, instructions can be issued **out-of-order**.
  - In this case, if the second instruction has data dependencies with the first, but the third instruction does not, the first and third instructions can be co-scheduled. This is also called **dynamic issue**.

- Performance of in-order issue is generally limited
Superscalar Execution: Efficiency Considerations

• Not all functional units can be kept busy at all times.
  – If during a cycle, no functional units are utilized, this is referred to as *vertical waste*.
  – If during a cycle, only some of the functional units are utilized, this is referred to as *horizontal waste*.

• Due to **limited parallelism** in typical instruction traces, **dependencies**, or the inability of the scheduler to extract parallelism, the performance of superscalar processors is eventually limited.

• Conventional microprocessors typically support **four-way** superscalar execution.
Very Long Instruction Word (VLIW) Processors

- The hardware cost and complexity of the superscalar scheduler is a major consideration in processor design.
- To address this issue, VLIW processors rely on compile time analysis to identify and bundle together instructions that can be executed concurrently.
- These instructions are packed and dispatched together, and thus the name very long instruction word.

- This concept was used with some commercial success in the Multiflow Trace machine (1984).
- Variants of this concept are employed in the Intel Itanium IA64 processors.
VLIW Processors: Considerations

• Issue hardware is simpler.
• Compiler has a bigger context (than Superscalar CPUs) from which to select instructions to co-schedule
  – to pack in a VLIW instruction
• Compilers, however, do not have runtime information such as cache misses or branch taken.
• **Scheduling** decision must be based on sophisticated static predictions to pack independent instructions in each VLIW
  – may result **inherently conservative**.
• VLIW performance is highly dependent on the compiler. A number of techniques such as loop unrolling, speculative execution, branch prediction are critical.
• Typical VLIW processors are limited to 4-way to 8-way parallelism.
Limitations of Memory System Performance

• Memory system, and not processor speed, is often the bottleneck for many applications.
• Memory system performance is largely captured by two parameters, latency and bandwidth.

• Latency is the time from the issue of a memory request to the time the data is available at the processor.
• Bandwidth is the rate at which data can be pumped to the processor by the memory system.
Memory System Performance: Bandwidth / Latency

• It is very important to understand the difference between latency and bandwidth.

• Consider the example of a fire-hose.
  – If the water comes out of the hose two seconds after the hydrant is turned on, the latency of the system is two seconds.
  – Once the water starts flowing, if the hydrant delivers water at the rate of 20 liters/second, the bandwidth of the system is 20 liters/second.

• If you want immediate response from the hydrant, it is important to reduce latency.
• If you want to fight big fires, you want high bandwidth.
Memory Latency: An Example

• Consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns for each request of 1 word (no caches).

• Assume that the processor has two multiply-add units and is capable of executing four instructions (2 mult and 2 add) in each cycle of 1 ns.

• The following observations follow:
  – The peak processor rating is 4 GFLOPS.
  – Since the memory latency is equal to 100 cycles and block size is one word, every time a memory request is made, the processor must wait 100 cycles before it can process the data.
Memory Latency: An Example

• On the above architecture, consider the problem of computing a dot-product of two vectors.
  – A dot-product computation performs one multiply-add (2 FP operations, storing the add result in an accumulator) on each single pair (2 operands) of vector elements
    • On average, each floating point operation requires one data fetch.
  – It follows that the peak speed of this computation is limited to two floating point ops (one multiply-add) every fetch of a pair (200 ns) ..... or
    • one floating point operation every 100 ns, or
    • an actual rate of \( \frac{1}{(100 \times 10^{-9})} = 10^7 = 10 \text{ MFLOPS}, \) a very small fraction of the peak processor rate (4 GFLOPS)!!
Improving Effective Memory Latency Using Caches

• **Caches** are small and fast memory elements between the processor and DRAM.
• This memory acts as a low-latency high-bandwidth storage.
• If a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache.

• The fraction of data references satisfied by the cache is called the cache **hit ratio** of the computation on the system.
• Cache hit ratio achieved by a code on a memory system often determines its performance.
Impact of Caches: Example

• Consider the architecture from the previous example. In this case, we introduce a cache of size 32 KB with a latency of 1 ns or one cycle.

• We use this setup to multiply two matrices A and B of dimensions $32 \times 32 = 2^5 \times 2^5$.
  – If a word (4 Bytes) is needed to store each element of a matrix, 4 KB are needed to store the whole matrix.

• With these numbers we can think that the cache is large enough to store matrices A and B, as well as the result matrix C.
Impact of Caches: Example (continued)

• The following observations can be made about the problem:
  – (Pre)-Fetching the two matrices A and B into the cache corresponds to fetching **2K words**, which takes approximately **200 µs** (miss penalty = 100 ns)
  – Multiplying two \( n \times n \) matrices takes \( 2n^3 \) operations. For our problem, this corresponds to \( 2 \times 2^5 \times 2^5 \times 2^5 = 64K \) operations, which can be performed in **16K cycles** (or **16 µs**) at **4 operation per cycle**.
  – The total time for the computation is therefore approximately the sum of time for load/store operations and the time for the computation itself, i.e., \( (200 + 16) \) µs.
  – This corresponds to a peak computation rate of \( 64K \text{ ops} / 216 \text{ µs} = (64 \times 2^{10}) / (216 \times 10^{-6}) \approx 303 \text{ MFLOPS} \).
Impact of Caches

- Repeated references to the same data item correspond to temporal locality
- In our example, we had $O(n^2)$ data to access and $O(n^3)$ computation. This asymptotic difference makes the above example particularly desirable for caches
- Data reuse is critical for cache performance

- Anyway, we saw that the compulsory cache misses to load the cache is enough to reduces greatly the computation rate
  - 303 MFLOPS vs. 4 GFLOPS
Impact of Memory Bandwidth

- Memory bandwidth is determined by the bandwidth of the memory bus as well as the memory units.
- Memory bandwidth can be improved by increasing the size of memory blocks.
- The underlying system takes $l$ time units (where $l$ is the latency of the system) to deliver $b$ units of data (where $b$ is the block size).
Impact of Memory Bandwidth: Example

• Consider the same setup as before (memory penalty = 100 ns), except, in this case, the block size is 4 words instead of 1 word.
  – 40 MB/s with blocks of 1 word (4 B)
  – 160 MB/s with blocks of 4 words (16 B)

• We repeat the dot-product computation in this scenario:
  – Assuming that the vectors are laid out linearly in memory, eight FLOPs (four multiply-adds) can be performed in 200 cycles.
  – This is because a single memory access fetches four consecutive words in the vector.
  – Therefore, two accesses can fetch four elements of each of the vectors ⇒ 200 cycles = 200 ns
    • 8 ops (performed in 2 ns) overlapped with the load of the next block
  – This corresponds to a FLOP every 25 ns, for a peak speed of 40 MFLOPS.
    • spatial locality
Impact of Memory Bandwidth

• It is important to note that increasing block size does not change latency of the system.
  – Physically, the scenario illustrated here can be viewed as a wide data bus (4 words or 128 bits) connected to multiple memory banks.
  – In practice, such wide buses are expensive to construct.
  – In a more practical system, consecutive words are sent on the memory bus on subsequent bus cycles after the first word is retrieved.
Impact of Memory Bandwidth

- The above examples clearly illustrate how increased bandwidth results in higher peak computation rates, but …

- The data layouts were assumed to be such that consecutive data words in memory were used by successive instructions (spatial locality of reference).

- If we take a data-layout centric view, computations must be reordered to enhance spatial locality of reference.
Consider the following code fragment:

```c
for (i = 0; i < 1000; i++)
    column_sum[i] = 0.0;
for (j = 0; j < 1000; j++)
    column_sum[i] += b[j][i];
```

The code fragment sums columns of the matrix b into a vector column_sum.
Impact of Memory Bandwidth: Example

- Vector `column_sum` is small and easily fits into the cache.
- Matrix `b` is accessed in a column order, but is stored in row-order.
- The strided access results in very poor performance.

We can fix the above code as follows:

```c
for (i = 0; i < 1000; i++)
    column_sum[i] = 0.0;
for (j = 0; j < 1000; j++)
    for (i = 0; i < 1000; i++)
        column_sum[i] += b[j][i];
```

- In this case, the matrix is traversed in a row-order and performance can be expected to be significantly better.
Memory System Performance: Summary

• The series of examples presented in this section illustrate the following concepts:
  – Exploiting **spatial and temporal locality** in applications is critical for amortizing memory latency and increasing effective memory bandwidth.
  – **Memory layouts** and organizing computation appropriately can make a significant impact on the spatial and temporal locality.
  – The **ratio of the number of operations to number of memory accesses** is a good anticipated indicator of tolerance to memory bandwidth.
Consider the problem of browsing the web on a very slow network connection. We deal with the problem in one of three possible ways:

1. we anticipate which pages we are going to browse ahead of time and issue requests for them in advance;
2. we open multiple browsers and access different pages in each browser, thus while we are waiting for one page to load, we could be reading others; or
3. we access a whole bunch of pages in one go - amortizing the latency across various accesses.

The first approach is called *prefetching*, the second *multithreading*, and the third one corresponds to a technique to exploit *spatial locality* (increasing the memory block) in accessing memory words.

– We already saw the third approach to reduce the effect of memory latency …
Multithreading for Latency Hiding

- A thread is a single stream of control in the flow of a program.
- We illustrate threads with a simple example (matrix by vector):

```c
for (i = 0; i < n; i++)
    c[i] = dot_product(get_row(a, i), b);
```

- Each dot-product is independent of the other, and therefore represents a concurrent unit of execution.
- We can safely rewrite the above code segment as:

```c
for (i = 0; i < n; i++)
    c[i] = create_thread(dot_product, get_row(a, i), b);
```
Multithreading for Latency Hiding: Example

• In the last code, the first instance of the function accesses a pair of vector elements and waits for them.
• In the meantime, the second instance of this function can be scheduled, to access two other vector elements in the next cycle, and so on.

• After \( l \) units of time, where \( l \) is the latency of the memory system, the first function instance gets the requested data from memory and can perform the required computation.
• In the next cycle, the data items for the next function instance arrive, and so on.
• In this way, in every clock cycle, we can perform a computation.
Multithreading for Latency Hiding

• The execution schedule in the previous example is predicated upon two assumptions:
  – the memory system is capable of servicing multiple outstanding requests, and
  – the processor is capable of switching threads at every cycle.
• It also requires the program to have an explicit specification of concurrency in the form of threads.
• Machines such as the HEP and Tera rely on multithreaded processors that can switch the context of execution in every cycle. Consequently, they are able to hide latency effectively.
Prefetching for Latency Hiding

- **Misses** on loads cause programs to **stall**.
- Why not **advance the loads** so that by the time the data is actually needed, it is already there!
- The only drawback is that you might need more space to store advanced loads.
- However, if the advanced loads are overwritten, we are no worse than before!

- Unfortunately, if you prefetch in the cache, you can introduce pollution in the cache, by removing blocks to be used in the next future
Tradeoffs of Multithreading and Prefetching

- Multithreading and (single-thread) prefetching are critically impacted by the memory bandwidth when combined with cache. Consider the following example:
  - Consider a computation running on a machine with a 1 GHz clock, 1-word cache line, single cycle access to the cache, and 100 ns latency to DRAM.
  - The computation has a cache hit ratio at 1 KB of 25% (miss 75%) and at 32 KB of 90% (miss 10%).
  - Consider two cases: first, a single threaded execution in which the entire cache is available to the serial context, and second, a multithreaded execution with 32 threads where each thread has a cache residency of 1 KB.
  - If the computation makes one data request in every cycle of 1 ns (4 GB/s), you may notice that the first scenario requires 400MB/s (10% 4 GB/s) of memory bandwidth and the second 3 GB/s (75% 4 GB/s)
Tradeoffs of Multithreading and Prefetching

- Bandwidth requirements of a multithreaded system may increase very significantly because of the smaller cache residency of each thread.
- Multithreaded systems become bandwidth bound instead of latency bound.
- Multithreading and prefetching only address the latency problem and may often exacerbate the bandwidth problem.
- Multithreading and prefetching also require significantly more hardware resources in the form of storage.
Parallel Architecture and Parallel Programming Models
Parallel platforms

• We now introduce, at a high level, the elements of parallel computing platforms that are critical for performance oriented and portable parallel programming
  – We first explore the logical architectural organization
• The critical aspects of the logical organization are how to support the execution of concurrent tasks and the interaction among them
• The former is related to the control structure model and the latter to the cooperation/communication model of the architecture
  – Control structure model
    • SIMD vs MIMD
  – Cooperation/communication model
    • shared memory (SM) vs. messages passing/distributed memory (DM).
• About concurrent tasks, parallelism can be expressed at various levels of granularity
  – from instruction level to processes.
  – between these extremes there exists a range of models, along with corresponding architectural support.
Parallel platforms

- **Flynn** [1972] introduced the following taxonomy of computers, where the parallel computers are **MIMD** (Multiple Instruction stream - Multiple Data stream) and **SIMD** (Single Instruction stream - Multiple Data stream).
- **Control structure model**
  - **Processing units** in parallel computers either operate under a centralized control or work independently under a private control.
  - **SIMD**: single control unit that dispatches the same instruction to various processors (that work on different data).
  - **MIMD**: each processor has its own control control unit, each processor can execute different instructions on different data items.
Capabilities of the four computer models

- **SISD**
  - +
  - A B
  - A + B

- **SIMD**
  - +
  - A B
  - C D
  - A + B
  - C + D

- **MISD**
  - +
  - A B
  - A * B

- **MIMD**
  - +
  - A B
  - C D
  - A + B
  - C * D
SIMD

- The array of processing elements (PEs) work *synchronously*
- At each step (global clock tick), PEs execute the same instruction on different data elements
  - **Array processors**: Connection Machine, ICL DAP (Distributed Array Processor), MasPar MP
  - **Pipelined vector computer**: CRAY 1 & 2 e CYBER 205
  - **co-processors SIMD** (MMX, SSE)
  - **GPUs (Graphics Proc. Units)**
- **SIMD architectures** suitable for solving data parallel programs, i.e. the case of the same function applied to distinct elements of a given data structure
  - Sum of two 2x2 matrixes: \( A + B = C \).
  - If A and B are distributed and stored in the memories of 4 PEs, so in parallel:
    \[
    \begin{align*}
    A_{11} + B_{11} &= C_{11} \\
    A_{12} + B_{12} &= C_{12} \\
    A_{21} + B_{21} &= C_{21} \\
    A_{22} + B_{22} &= C_{22}
    \end{align*}
    \]
MIMD (Multiprocessor / Multicomputer)

- Each processor works *asynchronously* under the control of a distinct stream of instructions (*programs*) working on different data.
- As for SIMD, cooperation through shared memory or interconnection network.
- MIMD computer with SM are known as *multiprocessors* or *tightly coupled machines*. The current multi-cores fall within this class.
- MIMD computer where the processes interact by *message passing* are known as *multicomputers* or *loosely coupled machines*.
- Some examples: IBM SP, Intel Paragon, TM CM5, Cray T3E, Meiko CS-2, all modern *clusters*.

**Diagram:**

- **Shared Memory OR Interconnection Network**
- **Data Stream (DS):** DS1, DS2, ..., DS N
- **Instruction Stream (IS):** IS1, IS2, ..., IS N
- **Processors:** Processor One, Processor Two, ..., Processor N
- **Controls:** Control One, Control Two, Control N

DS = DATA STREAM  IS = INSTRUCTION STREAM
Shared memory vs. message passing

- In both SIMD and MIMD the cooperation model can be based either on
  - SHARED MEMORY (SM) and SHARED VARIABLES
  - DISTRIBUTED/PRIVATE MEMORY (DM) and MESSAGE PASSING

- Examples of MIMD architectures:

```
P: processor     M: memory bank     C: cache
```

```
Multiprocessor (SM)  Multicomputer (DM)
```
SM multiprocessors

- Load/Store instructions can directly access all (part of) the shared physical address space
  - the same instruction supported by uniprocessors
  - simple and fast mechanism to communicate and share information
- Specific instructions to build mechanisms of synchronization (test&set, exch, load-locked, ecc.)
UMA vs. NUMA

- Different Shared Memory organization
  - Cost of **Memory Access**: *Uniform* (UMA) or *Non-Uniform*: (NUMA)

- The distinction between NUMA and UMA platforms is important from the point of view of algorithm design
  - NUMA machines performance also depends on memory allocation
  - Both UMA and NUMA machines need to exploit locality thus improving cache usage
Shared memory vs. message passing

• In SM cooperation model we have the same physical address space that can be accessed by all processors
  – cooperation by reading/writing the same shared variable mapped in SM
  – however, read-write data to shared data must be coordinated/synchronized (this will be discussed in greater detail when we will talk about threads programming)
  – cooperation method suitable for small numbers of processors (2-32 ways MIMD multiprocessors), at least in UMA organizations

• In DM cooperation model, we have to cope with a private physical address space for each processor
  – cooperation/synchronization via message passing over the network
  – common cooperation model adopted by clusters and high-end massively parallel computers over fast interconnection networks
  – data must be explicitly assigned to the private address spaces the various computational node
The programming model refers to how the parallel platforms virtually appears to programmers:

- due to the presence of software layers, it is nowadays possible to have programming models that diverge from the underlying physical architecture model
- to exploit some specific features of the architecture, it is often needed to use programming languages and environments that are close to the architecture

Control
- How parallel tasks can be created, terminated, and how they can synchronize

Naming
- Shared or private data
- Naming of data/processes/network abstraction

Operations
- Which are the base operations
- Which are the associated costs
Comparing programming models

- We consider a simple example

- A sum of \( n \) values, in turn obtained by applying a function \( f() \) to each element of a set of \( n \) elements

\[
\sum_{i=0}^{n-1} f(A[i])
\]
Comparing programming models

- All the $p$ tasks compute a partial sum of distinct $n/p$ values, each obtained by the application of function $f()$ over distinct elements
  - The tasks compute independently the partial sums by using “private” variables
- At the end, one (or all) task takes the partial sums and produces the global sum
  \[
  \sum_{i=0}^{n-1} f(A[i]) \rightarrow \sum_{j=0}^{p-1} \sum_{i=0}^{n/p-1} f(A[i + j \times n/p])
  \]

- Data that are logically shared
  - Variable storing the $n$ original numbers and the final global sum
- Data that are logically private
  - Temporary variables and the variables used for partial sums

In DM architectures such data must be allocated and copied in the private memory space of each processor.
“Shared address” programming model

- **Shared Address Space**
  - A program consists of a collection of threads of control
  - Each with a set of private variables
    - E.g., local variables on the private stack frame of the various function instances executed by the threads
  - Collectively and asynchronously access a set of shared variables
    - E.g., static variables, shared common blocks, global heap
  - Threads communicate implicitly by writing and reading shared variables
  - Threads coordinate explicitly by synchronization operations on shared variables
    - Writing and reading flags
    - Locks, semaphores

- **Like concurrent programming on uniprocessor**
The associated architectural model

- An SM – MIMD architecture
  - Processors all connected to a large shared memory
  - Caches are a sort of low-latency local memories which maintain copies of shared data

- Shared address programming models are, to some extent, the native programming models for MIMD SM architectures

- From the programming viewpoint, even if shared data can be logically partitioned among threads, they are not owned
  - it can become complex to guarantee locality in memory accesses
Shared address code for computing a sum

- **Shared variables**
  - Array A[N], read-only
  - Variable s, which eventually contains the final sum
- **Private variables**
  - Index i
  - Variables local_s1 and locals_s2 are physically shared, but are virtually private

**Thread 1**

- 
  - \([s = 0 \text{ initially}]\)
  - local_s1 = 0
  - for \(i = 0, n/2-1\)
    - local_s1 = local_s1 + f(A[i])
  - s = s + local_s1

**Thread 2**

- 
  - \([s = 0 \text{ initially}]\)
  - local_s2 = 0
  - for \(i = n/2, n-1\)
    - local_s2 = local_s2 + f(A[i])
  - s = s + local_s2

What could go wrong?
Pitfall and solution via synchronization

- Pitfall in computing a global sum: $s = local_s1 + local_s2$

![Diagram showing thread operations]

- Instructions from different threads can be interleaved arbitrarily
- What can final results stored in memory be?
- Race Condition
- Possible solution: Mutual Exclusion with Locks

- Lock operations must be atomic (execute completely without interruption)
“Message passing” programming model

- **Message Passing**
  - a program consists of a collection of “named” processes/task
    - threads of control plus local address spaces
    - local variables, static variables, common blocks, heap
  - processes communicate by **explicit data transfers**
    - matching pair of send & receive by source and destination processes
  - coordination/synchronization is implicit in every communication event
  - **How can we manage logically shared data?**
    - read-only data are easy to manage
      - they can be partitioned / distributed or replicated in the local address space, according to the access patterns
    - read-write data are more complex
      - they can be either centralized (easy management, but bottleneck) or partitioned/distributed

- **Like distributed programming**
The associated architectural model

- A DM – MIMD architecture
  - Processors can access in an exclusive way to private memory & local cache
    - Processors cannot directly access another processor’s memory
  - Each “node” has a network interface (NI)
    - All communication and synchronization done through the message passing mechanisms

- Message passing programming models are, to some extent, the native programming models for DM - MIMD architectures
Compute $S = X(1) + X(2)$ on each processor of a pair

- Sub-vectors $A[0 : n/2-1]$ and $A[n/2 : n]$ allocated in the private memory of two distinct processors
- Possible solution to realize $S = X(1) + X(2)$, where $X(1)$ and $X(2)$ are the private variables storing the two partial sums (this solution can entail deadlocks)

### Processor 1

- send $x_{local}$, proc2
- $[x_{local} = x(1)]$
- receive $x_{remote}$, proc2
- $s = x_{local} + x_{remote}$

### Processor 2

- send $x_{local}$, proc1
- $[x_{local} = x(2)]$
- receive $x_{remote}$, proc1
- $s = x_{local} + x_{remote}$

- Second solution

### Processor 1

- send $x_{local}$, proc2
- $[x_{local} = x(1)]$
- receive $x_{remote}$, proc2
- $s = x_{local} + x_{remote}$

### Processor 2

- receive $x_{remote}$, proc1
- send $x_{local}$, proc1
- $[x_{local} = x(2)]$
- $s = x_{local} + x_{remote}$
“Data parallel” programming model

- **Data Parallel**
  - Single sequential thread of control consisting of parallel operations
  - Parallel operations applied to all (or defined subset) of a data structure ➔ DATA PARALLELISM
  - Communication is implicit in parallel operators and “shifted” data structures
  - Elegant and easy to understand and reason about
  - Not all problems fit this paradigm

\[
\begin{align*}
A &= \text{array of all data} \\
fA &= f(A) \\
s &= \text{sum}(fA)
\end{align*}
\]
The associated architectural model

- An SIMD (Single Instruction Multiple Data) machine
- A large number of small functional units (low parallelism granularity)
- A single “control processor” issues each instruction
  - each processor executes the same instruction
  - some processors may be turned off on any instruction

- Data parallel programming model are, to some extent, the native programming models for SIMD architectures
Programming Abstractions vs. Architectural Model

- It is important to note the difference between
  - the terms used to refer to the *programming model*, and thus the programming abstractions
  - the terms used to refer to *architectural model* behind the physical realization of a given machine

- **Shared-Address-Space** vs. **Shared Memory**
  - We refer to the former as a programming abstraction and to the latter as a physical machine attribute
  - It is possible to provide a shared address space using a physically distributed memory, even though there are problems of efficiency

- **Message Passing** vs. **Distributed Memory**
  - We refer to the former as a programming abstraction and to the latter as a physical machine attribute.
  - It is easy to provide the run-time support of a message passing programming language/library over a physically shared memory, where communication channels become shared memory queues/mailboxes
Physical Organization of Parallel Architectures
SM UMA Multiprocessors

- **Bus**: inexpensive network adopted by many symmetric multiprocessors (SMP) with limited parallelism degree
  - inexpensive and expandable
  - not too much scalable due to the **limited bandwidth** of the bus
  - **coherent caches** relieve the problem of the shared bus

- **For SMPs with higher degree of parallelism**
  - **Crossbar**: Expensive but not too expandable
  - **Multistage**: w.r.t. crossbar, less expensive, increased latency, reduced aggregate bandwidth
SM NUMA Multiprocessors

- The architecture appears to be similar to a DM architecture (with Private/Distributed Memory)
  - convergence among the scalable architectures
  - the scalability of SM Numa Multiprocessors is paid in term of programming complexity
- Access to the global shared memory transformed into either
  - accesses to the “local” memory, or
  - accesses to the “remote” memory, through low level messages sent to / received from the controllers of the remote memories
  - read-request  read-response
SM NUMA Multiprocessors

• Alternative 1 (SGI / Cray T3E)
  – The model of memory (local/remote) is exposed to the programmer, whose task is to reduce as much as possible the need of remote accesses
  – The memory controller generates messages for remote load / store
  – We do not have hardware mechanisms for consistency
    • If data are replicated on different memory modules, the consistency between copies is under the programmer's responsibility
  – The local cache is present, but maintains only copies of data from the local memory

• Alternative 2 (modern one)
  – Local caches can also store copies of remote data blocks
    • The miss penalty is not uniform !!!
    • Still programmer’s intervention to exploit locality at the level of local memories close to processors
  – Scalable Cache Coherence Mechanism for the various caches
SM MIMD: Multicore Organization (UMA and NUMA)

• Main design variables:
  – Number of core processors on chip (dual, quad ... )
  – Number of levels of cache on chip (L1, L2, L3, ...)
  – Amount of shared cache vs. not shared (1MB, 4MB, ...)

• The following slide has examples of each organization:
  a) ARM11 MPCore
  b) AMD Opteron
  c) Intel Core Duo
  d) Intel Core i7

Multicore UMA Organization Alternatives

(a) Dedicated L1 cache

(b) Dedicated L2 cache

(c) Shared L2 cache

(d) Shared L3 cache

On-chip Dedicated caches

On-chip Shared Caches
Pros and Cons of shared L2/L3 Cache

• Pros
  – Constructive interference reduces overall miss rate (A wants X then B wants X → good!)
    • Data shared by multiple cores not replicated at cache level (one copy of X for both A and B)
  – The amount of shared cache dedicated to each core is dynamic
    • Threads with less locality can obtain more cache blocks
  – Easy and fast inter-process communication through shared cache
  – Cache coherency confined to small L1

• Cons
  – Dedicated L2 cache gives each core more rapid access
    • Good for threads with strong locality
• Shared L3 cache may also improve performance
Core i7 (Xeon) vs. Duo

- Intel Core Duo uses superscalar cores
- Intel Core i7 uses cores supporting simultaneous multi-threading (SMT)
  - Scales up number of threads supported
  - 4 SMT cores, each supporting 4 threads appears as a 16 core (the illustrated corei7 has 2 threads per CPU: virtually 8 cores)
Intel Core Duo Block Diagram

- 32 KB L1 Caches
- Execution Resources
- Arch. state
- Thermal Control
- APIC
- Power Management Logic
- 2 MB L2 Shared Cache
- Bus Interface
- Front-Side Bus
Intel x86 Multicore Organization - Core Duo (1)

- 2006
- Two x86 superscalar, shared L2 cache
- Dedicated L1 cache per core
  - 32KB instruction and 32KB data
- Advanced Programmable Interrupt Controlled (APIC)
  - Inter-process interrupts between cores
  - Routes interrupts to appropriate core
  - Includes timer so OS can self-interrupt a core
• Power Management Logic
  – Monitors thermal conditions and CPU activity
  – Adjusts voltage (and thus power consumption)
  – Can switch on/off individual logic subsystems to save power

• 2MB shared L2 cache, and dedicated L1
  – Dynamic allocation
  – MESI cache coherence support for L1 caches
  – Extended to support multiple Core Duo in SMP
Intel Core i7 Block Diagram

Need to maintain cache coherence

Need to maintain cache coherence
Intel x86 Multicore Organization - Core i7

- November 2008
- Four x86 SMT processors
- Dedicated L2, shared L3 cache
- Speculative pre-fetch for caches
- On chip DDR3 memory controller
  - Three 8 byte channels (192 bits) giving 32GB/s

- **QuickPath Interconnect**
  - Cache coherent point-to-point link
  - High speed communications between processor chips
  - Dedicated bi-directional pairs
  - Total bandwidth 25.6GB/s
UMA/NUMA Intel Core i7 (Xeon)

- We can mix several UMAs to create a mixed UMA/NUMA architecture
- Processors within a UMA node
  - shared access to memory modules
  - may access memory in a remote node by using a shared interconnect, but with slower performance
- Intel Core i7 processors employ the fast interconnect technology known as Intel QuickPath Interconnect (QPI) to mitigate (but not eliminate) the problem of slower remote memory performance.
NUMA performance issues

- **Data placement**
  - The more data can effectively be placed in memory local to the core that needs it (due to the thread/processes running on the core), the more overall access time will benefit from the architecture.
  - To realize this potential, strategies are needed to ensure smart data placement, and its effective exploitation.

- **Data Placement Using Implicit Memory Allocation Policies**
  - Many OSs transparently provide support for NUMA-friendly data placement.
  - Physical memory pages are assigned to the memory banks associated with the thread's node that requests memory allocation (or accesses the first time the allocated memory).
  - This ensures that memory is local to the thread and access performance is optimal.

- **Data Placement Using Explicit Memory Allocation Directives**
  - Another approach to data placement in NUMA-based systems is to make use of system APIs that explicitly configure the memory page allocations.
NUMA performance issues

- **Processor Affinity**
  - Data placement is not enough, because threads can be scheduled dynamically
  - **Thread migration** from one core to another poses a problem for the NUMA shared memory architecture, because of the way it disassociates a thread from its local memory allocations.
  - Using a system API, or by modifying an OS data structure (e.g., affinity mask), a specific core or set of affine cores can be associated with an application thread
    - The scheduler will choose among these alternatives without migrating the thread to other cores
  - *In general, forcing processor affinity may significantly harm system performance, by restricting scheduler options and creating resource contention when better resources management could have otherwise been used*
DM MIMD: Multicomputers

- **Private memory** and **private cache** for each node
- Node interconnected by a **network**
  - *such networks may share characteristics with the ones adopted in SM multiprocessors*
- Processes exchange private data by **message passing** (MP)
  - **Send + MatchingRecv** are a mechanism for **copying** between private memories, and for **synchronizing process activities**
Myrinet: an example of fast cluster network

- A very fast network per cluster
  - physically consists of two fiber optic cables, upstream and downstream, connected to the host computers with a single connector
  - switch boxes that can be connected to other boxes or single ports of each host (NIC)
  - bi-directional links support up to 2+2 Gb/s
  - the latency of each switch box is about 550 ns.
Myrinet

- The packets can have whichever length, and encapsulate other packets, e.g. IP
- Using the APIs to access the Myricom's GM message-passing system the network can sustain one-way bandwidth of
  - ~1.96 Gbits/s between Unix processes mapped on different machines
  - Latencies for short messages of ~7µs
- Message passing libraries as MPI, VIA, and TCP/IP have been implemented efficiently on the GM level
A mesh-based Myrinet

- More complex networks by interconnecting many Myrinet switches
- Dimensional routing to avoid deadlock
  - Packets traverse the mesh first on dimension $x$, then on dimension $y$
- The head of the packet contains routing info
SIMD architectures

- Processor array (Connection Machine CM-1 and 2, MasPar MP-1 and 2): in the early 1980s
- Specialized for data parallelism, with simple PEs (processing elements) processing one bit (or a few bits) at a time, and direct interconnection networks
- Synchronous Operations (global clock): arithmetic, global data transfer
- Large number of PE: e.g., CM-1 has max 64K PEs
Pipelined-SIMD Architectures (Vector Machines)

- Vector machines = *Supercomputers of ‘70-’80 (Cray)*
- Able to execute “in pipeline” data parallel instructions affecting vectors/arrays: (+, -, *, /)
- Transformation of data parallelism into pipeline parallelism
  - Vector Instruction = Data-parallel collective operation
  - Single scalar operation = decomposed into a pipeline of n stages
  - Identical and independent scalar operations on distinct vector elements can be executed in pipeline

E.g. `mult C, A, B` with C, A and B vector registers of n elements

\[
\begin{align*}
A[n] & \quad B[n] \\
\end{align*}
\]

\[
\begin{align*}
C[1] & \\
C[2] & \\
C[n] & \\
\end{align*}
\]

A very long data stream is needed to keep units busy
An example of supercomputer (vector machine)

- Instead of leaving the data in memory, we have "vector registers"
  - in Cray-1 the vector registers held sixty-four 64-bit words each
- The vector instructions are applied between registers, which is much faster than working with main memory data
- Pipeline parallelism to implement vector instructions rather than multiple ALUs.
Streaming SIMD Extensions

- Streaming SIMD Extensions (SSE) is a SIMD instruction set extension to the Intel x86 architecture (SSE, SSE2, …, SSE5)
  - introduced in 1999 in Pentium III series processors as
  - SSE contains 70 new SIMD instructions
  - 128-bit (16B) registers: XMM0, XMM1, …, XMM7
  - each register packs together:
    - sixteen 8-bit bytes or characters, or
    - …
    - four 32-bit single-precision floating point numbers, or
    - two 64-bit double-precision floating point numbers

- Example of SSE SIMD code
  - v1 and v2 are the memory addresses of two arrays of four words

\[
\begin{align*}
\text{movaps} & \quad \text{xmm0, address-of-v1} & \# \text{xmm0} = v1.w \mid v1.z \mid v1.y \mid v1.x \\
\text{addps} & \quad \text{xmm0, address-of-v2} & \# \text{xmm0} = xxm0.w + v2.w \mid xxm0.z + v2.z \mid xxm0.y + v2.y \mid xxm0.x + v2.x \\
\text{movaps} & \quad \text{address-of-res, xmm0} & \# \text{res.w} \mid \text{res.z} \mid \text{res.y} \mid \text{res.x} = \text{xmm0}
\end{align*}
\]
The NVIDIA GeForce 8800 GTX features 16 streaming multiprocessors of 8 (stream) processors each.

Each streaming multiprocessor (8 SP each) is SIMD. Each SP runs a copy of the same thread.
• A pair of *streaming multiprocessors*, each composed of 8 *stream processors* (mds), with shared instruction/data caches, control logic, a 16 kB shared memory, and two special function units.

**GPU SIMD processors**
GPU SIMD processors

- At the heart of the system there is a parallel processing unit (streaming multiprocessor - SM) that operates on \( n \) entities at a time
  - The SM is further replicated \( m \) times to allow \( m \) sets of entities to be processed in parallel

- An SM operating on \( n \) entities at a time typically uses a SIMD design
  - a single stream of instructions is used to control the \( n \) Scalar Processors (SPs)
  - More specifically, Thread Processor is the unit responsible for coordinate the execution within the of SM, leading to an execution model called SIMT (Single Instruction Multiple Threads)
  - SIMT is an extension to the SIMD model (Single Instruction Multiple Data) to be the thread responsible for accessing to the different data.
GPU SIMD processors

- Achieving maximum efficiency requires processing $n$ entities executing an identical instruction stream (computational coherence) within the same thread
  - challenging when the streaming programs (threads/kernels) executed by each multiprocessors are allowed to make use of conditional flow control (branching) constructs.

- Other issues are the ways how data are accessed, which can hurt performance
  - Device memory (slower)
  - Per SM - Shared memory (faster)
  - Coalesced access to memory
Dataflow Machines: historical perspective

- Computational model that is very different from the classical von Neumann one

\[ a = (b+1) \times (b-c) \]
\[ d = c \times e \]
\[ f = a \times d \]

Program represented as a data flow graph

Instructions = Graph Nodes
Data Storage = Instruction themselves

- In the **dataflow model**, the instructions are fired by the operand availability (data flow) ⇒ data-driven
- In the **von Neumann model** (control flow), there is and explicit execution order between the instructions ⇒ control-driven
- The main advantage of the data flow model is that only the important/true dependencies (due to data) are made explicit in the graph!
  - all the parallelism is explicitly exposed to the HW
Dataflow Machines

• **Pros**
  – Explicit parallelism, dynamic scheduling of the all fire-able instructions

• **Cons**
  – Copying of data (from the producer instruction to the consumer ones). Problems with data structures
  – No locality exploitation
    • we can exploit locality by grouping instruction connected in the graph to reduce communication/copies

• **Influence of the data flow model:**
  – Data flow analyses used by compilers to extract ILP, and by the HW of microprocessors to dynamically schedule ready/fire-able instructions (out-of-order execution)
  – The *macro data flow SW paradigm*
    • can be realized on conventional MIMD machines
    • the *data flow operations* have a *greater granularity* (tasks composed of many instructions)
    • the run-time support must be able to dynamically schedule multiple tasks on worker threads
    • exploitation of a shared address space to avoid copying of data
Convergence of HW/SW architecture

• Programming language and supporting middle-ware make the SW portable between diverse architectures
  – We can exploit programming models that are not native for that specific architecture
  – Send/Recv (message passing) on SM architectures
  – Global shared address space (Virtual SM) on a DM multicompiler
    (Global Addr -> Proc Id | Local Addr)
    • Page-based (o finer-grained) shared virtual memory

• HW is converging
  – Also the HW of SM and DM machines look like similar
    • NUMA multiprocessors vs. multicompurers
  – Network Interface close to the processor-memory interconnection to guarantee low-latency and high-bandwidth
Parallel Scalable Architecture

- Node: processor(s), memory system, *communication assist (CA)*
  - CA very close to processor-memory
- Scalable network
- Both multicomputer (DM-MIMD) machines and NUMA multiprocessors (SM-MIMD) share this scheme
Convergence of HW/SW architecture

Even for distinct *architectural models*, the physical organization converges

- Systolic Arrays
- Dataflow
- Generic Scalable Architecture
- SIMD
- Message Passing
- Shared Memory
Message passing and interconnection networks
MPP networks vs. LAN networks

- **High data rates**
  - Channel data rates today are in the range from hundreds to thousands of Megabits/s (Mb/s)
  - Channels organized in full-duplex pairs called links
  - Compare with 100Mb/s "100-Base" Ethernet, only one direction at once

- **Regular topologies and scalability**
  - Network constructed by interconnecting elementary routing circuits in a mathematically regular topology (mesh, hypercubes, etc.)
  - The regularity of the network allows simple, algorithmic routing that avoids deadlocks that might otherwise occur due to cyclic dependencies in routes
  - Aggregate capacity grows with the number of nodes because many packets may be in transit concurrently along different paths

- **Very low error rate**
  - Networks operates in an intra-computer environment, and thus are reliable
  - This avoids error-tolerant protocols that require additional storage and significant overhead for each packet
Performance of message passing

• **Latency of startup**
  – Time taken by a generic task to start working
  – **Startup latency** of sending a message
    • time spent between the instant when the delivery of the first bit of
      the message starts on the sender computer, and the instant when
      the same bit arrives at the receiver computer

• **Bandwidth**
  – Bits/s that can be transmitted over a given network connection

• **Simple performance model for message passing**
  – time to deliver a message of $N$ Bytes
    \[
    \text{Time} = \text{latency} + \frac{N}{\text{bandwidth}}
    \]
Performance of message passing

- Overhead HW/SW on sender/receiver
- The dispatch/reception of a message is an expensive I/O operation:
  - packing/unpacking (protocol)
  - dispatch of data on the network
  - Deliver/reception of data from the network
    - copying data from/to the user space (kernel space) to/from the network buffer (and vice versa)
Some concepts of networking

- **Topology**
  - The network graph topology determines the possible routes between sender and receiver

- **Routing**
  - Algorithm to compute a path from source to destination
  - It can be decided on the source (whole path), or on the fly over the intermediate devices

- **Flow control**
  - It controls when a link will be available for transmission, otherwise the packet must wait

- **Switching**
  - Refers to single packet
  - Once the communication links determined by the routing algorithm are determined and granted, they must be switched to build the path
  - “How and when are the link paths allocated for the packets?”
    - Before the arrival of the packet: *circuit switching*
    - Once the whole packet is received: *store-and-forward packet switching*
    - Once a small portion of the packet is received: *cut-through (wormhole) packet switching*
Performance of message passing

So, a more accurate performance model has to take into account:
- the overheads on sender/receiver sides
- network topology
- routing strategy
  - to avoid and reduce possible conflicts on the traversed links
- switching strategy
  - Circuit/Packet/CutThrough switching
- flow control
  - A mechanism for the receiver to control the transmission speed. When can a message/packet traverse a portion of a given path?
Evolution of MPP networks

- **Switches and specialized networks**
  - In the first generation networks, it was very important to reduce the number of hops to deliver a packet, and this depends on the topology
    - store&forward *switching*
    - each switching node in the routing path has to receive and buffer the whole packet before forwarding it to the next node
    - communication cost *linearly* depends on: hops number \( \times \) packet size
  - The modern *switch* devices transmit in *pipeline* the packet (*wormhole*)
    - more links work in parallel to deliver *packet portions* (*flit*=flow control digits), which follows the same path fixed by the head of the packet
    - this method reduces the importance of the distance between sender/receiver (number of hops) as a measure of cost
    - the cost depends on: *number of hops + packet size*
Performance of Wormhole (Virtual cut-through)

- **Switching strategies**
  - Store&Forward vs. Wormhole (Virtual cut-through)
- Cut-through strategy (Wormhole)
  - Packets are divided in small flits, where the first contains the address of the destination, opens the circuit, and all the other flits follow the same path (a sort of “worm”)
  - The performance is similar to the circuit-switching for large packet sizes

\[ n_{\text{hops}} \left( \frac{\text{dim}}{\text{bw}} + \Delta \right) \]

\[ \frac{\text{dim}}{\text{bw}} + n_{\text{hops}} \Delta \]
Performance of Wormhole (Virtual cut-through)

- **Wormhole**
  - We do not have memory buffers on the intermediate switches
    - Simpler technology and thus largely adopted
  - When a conflict occurs (output link busy), all the “worms” is blocked “in-place”

- **Virtual cut-through**
  - When a conflict occurs (output link busy), the following flits of the packet are extracted from the network and stored on the switch
  - In presence of many conflicts (and enough memory), the performance can become similar to packet switching (store&forward)

- **Blocked worm (wormhole)**
- **Worm rolled up and buffered (cut-through)**
Evolution of MPP networks

• Several techniques to “attach” the network to a computing node
• The goal of the evolution is to reduce the transmission latency
  – overheads to copy data, build packets, switch between user/kernel mode
• HW/SW techniques
  – DMA + specialized co-processors
  – user-level primitives with less copies
  – move the communication subsystem on the fast processor-memory interconnection/bus
Evolution of MPP networks

- **First generation:**
  - **Static Direct Networks**
  - The hardware only guarantees communication between nodes directly connected with direct links
  - SW to route messages
  - Network topologies exposed to programmer
  - Topology-aware algorithms

- **Evolution**
  - Nodes attached to a switch, which also **dynamically route** message
  - The HW guarantees a **virtual complete interconnection** to the SW levels of communication
Interconnection networks

• In the past years, message passing programming had to be topology-aware
  – e.g., different algorithms for meshes or hypercubes
• Nowadays
  – we have switched networks, with hw/fw routing/switching
  – number of hops are less important, thanks to wormhole

• On modern hw it is better to reduce the amount of message exchanged
  – the gap between the computing bandwidth of microprocessors and the communication cost has been increasing
  – less communications also mean to have less probability of link conflicts, and this way wormhole works better

• The programmers can write algorithms by assuming a virtually completed connected topology
  – S/he can however refer to general parameters about the network topology, able to summarize the main features of the network, and thus the impact on the communication cost and the performance of parallel programming
Interconnection networks: communication cost

- The cost of sending $m$ data units can be modeled, in absence of conflicts, as $t_{\text{comm}}(m) = t_{\text{startup}} + m \ t_{\text{data}} = t_{\text{startup}} + \frac{m}{\text{bandwidth}}$

- Some typical values for $t_{\text{startup}}$ and $\text{BW}$
  - we can deduce that the startup cost (protocol overhead + startup hw) is decreased in the year, but not enough due to the performance increase in the modern microprocessors:

<table>
<thead>
<tr>
<th>Machines</th>
<th>Year</th>
<th>Startup (µs)</th>
<th>BW (MB/s)</th>
<th>Startup (cicles)</th>
<th>MFLOPS (per proc)</th>
<th>Startup (FLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nCUBE/2</td>
<td>1990</td>
<td>160</td>
<td>2</td>
<td>3000</td>
<td>2</td>
<td>300</td>
</tr>
<tr>
<td>CM-5</td>
<td>1992</td>
<td>95</td>
<td>8</td>
<td>3135</td>
<td>20</td>
<td>1900</td>
</tr>
<tr>
<td>Meikp CS-2</td>
<td>1993</td>
<td>83</td>
<td>43</td>
<td>7470</td>
<td>24</td>
<td>1992</td>
</tr>
<tr>
<td>SP-2</td>
<td>1994</td>
<td>35</td>
<td>40</td>
<td>3960</td>
<td>200</td>
<td>7000</td>
</tr>
<tr>
<td>Cray T3D</td>
<td>1994</td>
<td>21</td>
<td>27</td>
<td>3150</td>
<td>94</td>
<td>1974</td>
</tr>
<tr>
<td>NOW</td>
<td>1996</td>
<td>16</td>
<td>38</td>
<td>2672</td>
<td>180</td>
<td>2880</td>
</tr>
<tr>
<td>SUN E6000</td>
<td>1996</td>
<td>11</td>
<td>160</td>
<td>1760</td>
<td>180</td>
<td>1980</td>
</tr>
</tbody>
</table>
Interconnection networks (Crossbar and bus)

- **Crossbar.** We have a number of switching elements equal to $p^2$
  - High performance (direct connections)
  - Supporting parallel permutations
  - Not expandable

- **Bus.** We can easily expand the number of communicating nodes attached to the bus

- Scalability is however not ensured, since the bus becomes a bottleneck when we increase the number of concurrent communicating nodes
Interconnection networks

- Design of network topologies that are a tradeoff between crossbar and bus

- **Direct/Static Networks**
  - each node (processor, memory modules, computer) is directly and statically connected by *links* to a finite number of other nodes

- **Indirect/Dynamic Networks**
  - the connections among nodes traverse a set of *switches* that dynamically route messages
Interconnection networks

• The previous definition is now imprecise
  – all the direct topologies are now dynamic, with switches that deliver messages to destination

• “Dynamic” Direct Network
  – There is a switch for each interconnected element
  – The direct links between the nodes become direct connection between the “correspondent” switches
  – Ratio of switches to processing elements is 1:1
    • Every switch is connected to 1 processor element, and at least 1 other switch node
  – Node = Element + Switch

• “Dynamic” Indirect Network
  – the number of switches is different from the number of interconnected elements
  – Ratio of switches to processor elements is different from 1:1
  – Some switches are simply connected to other switches
Direct vs. Indirect

Classification of interconnection networks: (a) a static network; and (b) a dynamic network.
Switch Boxes

• A switch box is a switching element that
  – maps a fixed number of input ports to a fixed number of output ports

• The number of input/output switch ports is called switch degree: \( d \)
  – The cost of a switch grows as \( d^2 \) (number of switching elements)
    if the switch box is internally realized as a crossbar

• The switch boxes can support routing, multicast, buffering
Indirect networks

- An example is a *multistage network*, where each *stage* is implemented with a *set of switch boxes*
- In this example, the connected nodes can be processors & memories (SM-MIMD UMA), but the same network can be used for a multicomputer

![Diagram of a multistage interconnection network](image-url)
Indirect networks: Omega network

- \( \log p \) stages, where each stage is made up of \( p/2 \) small switch boxes
- Each switch box has degree 2: it is a small 2x2 crossbar (4 switching elements), able to perform a pair of operations: pass-through and cross-over
- A total number of \( 2 \log p \) switching elements rather than \( p^2 \) (crossbar)
- Simple deterministic routing, only comparing at the \( i\text{-th} \) stage the \( i\text{-th} \) bit of the binary representation of the IDs of the sender/receiver:
  - if different: cross-over
  - if equal: pass-through
Indirect networks: Omega Stage Interconnections

- Perfect shuffle: is a specific permutation
  - See the example from 8 inputs to 8 outputs

- General formulation: input $i$ is connected to output $j$ if:

  $$ j = \begin{cases} 
  2i, & 0 \leq i \leq p/2 - 1 \\
  2i + 1 - p, & p/2 \leq i \leq p - 1 
  \end{cases} $$
Indirect networks: conflict blocking in Omega networks

- An example of blocking due to a conflict
  - Deterministic routing
  - One of the two packets, either $010 \rightarrow 111$ or $110 \rightarrow 100$, is blocked on link AB
Indirect networks: Butterfly networks \((k\text{-ary } n\text{-fly})\)

- **Butterflies of** dimension \(n\) and variety \(k\)
- **Example:** 2-ary 3-fly for 8 + 8 nodes:

\[
\begin{align*}
C_0 & \rightarrow S_0 \\
C_1 & \rightarrow S_1 \\
C_2 & \rightarrow S_2 \\
C_3 & \rightarrow S_3 \\
C_4 & \rightarrow S_4 \\
C_5 & \rightarrow S_5 \\
C_6 & \rightarrow S_6 \\
C_7 & \rightarrow S_7
\end{align*}
\]

Number of processing nodes:
\(N = 2 \, k^n\)
e.g., in an SMP, \(k^n\) processors \((C_i)\) interconnected to \(k^n\) memory modules \((S_j)\)

**Network degree** = \(2k\).

**Network distance** (constant):
\(d = n\)
proportional to the base latency.
Thus, base latency = \(O(\log N)\).

As in Omega, there is **one and only one unique path** between any \(C_i\) and any \(S_j\). It is exploited in **deterministic routing** as in the Omega netwok
Direct networks

- 2-D mesh, 2D-torus, 3D-mesh
  - Both static and dynamic
Measures to evaluate topologies

- **Degree, d**
  - Number of links incident to a node
- **Diameter, D**
  - Longest (average) minimum path between pairs of nodes
- **Bisection Width, BW**
  - Minimum number of links that connect two disjoint partitions of the network
    - Each partition contains about half of the nodes
  - \( BW = \) number of links that cross the cut that *bi-sect* the topology
  - Bisection Bandwidth, \( BB \), is equal to: \( BW \times \text{link bandwidth} \)
Measures to evaluate topologies

- Networks where the Degree $d$ does not increase along with the numbers of nodes are expandable (and also scalable?)
  - it is easy to add further nodes, even if the concurrent communications supported per node do not increase
  - e.g., a $k$-ary $n$-cube where we fix $n$ and change $k$ to expand

- The Bisection Width (BW) takes into account the amount of maximum communication traffic that can traverse the cut of the two halves of the network
  - it is pessimistic
  - all the traffic between nodes of distinct partitions
  - the bisection has to cut the minimum number of link

- The Diameter $D$ is important for store\&forward
  - its is a pessimistic measure too
Family of direct networks

- **k-ary n-cube**
  - *n*: number of dimensions
  - *k*: radix (or base) used for numbering the nodes
- Both *static* and *dynamic* (with a switch per each node)
- Number of nodes $N = k^n$
- In detail;
  - *k* nodes in each dimension
  - each node labeled with a digit of *n* symbols expressed in radix *k*
  - each node is connected to nodes such that
    - the two associated labels differ for *only a digit* and for an *a single unit* $(mod \ k)$
- A *k*-ary *n*-cube can be built *recursively*
  - connecting *k* copies of a *k*-ary (*n-1*)-cube
  - the connections refer to nodes with the same label in the various original *k*-ary (*n-1*)-cubes
- There exists an analogous model, the *k*-ary *n*-fly one, for the *multistage butterfly networks*
Some examples of \( k \)-ary \( n \)-cubes

- **\( k \)-ary 1-cube**
  - 1D mesh
  - in the figure: \( k=4 \)
  - *limited degree*: \( d=2 \)
  - \( D=k-1 \)
  - \( N=k^n=k \)

- **\( k \)-ary 2-cube**
  - 2D-mesh
  - in the figure, \( k=4 \)
  - *limited degree*: \( d=4 \)
  - \( D=2(k-1) \)
  - \( N=k^n=k^2 \)

- There exist the corresponding *tori*
  - *wrap-around links*
2-ary $n$-cube (hypercube)

- 0-D hypercube
  - 2-ary 0-cube
- 1-D hypercube
  - 2-ary 1-cube
- 2-D hypercube
  - 2-ary 2-cube
- 3-D hypercube
  - 2-ary 3-cube
- 4-D hypercube
  - 2-ary 4-cube

- Note base-2 number system used to label the nodes
- *The degree is not limited to:*
  \[ d = \log N = \log k^n = n \]
  - $D = n$
  - $N = k^n = 2^n$
BW vs. communication patterns

- In principle, the interconnection networks with *richer topologies* (*more links and ports per node*) as hypercubes allow
  - more packets to traverse simultaneously more *links*

- *Bisection bandwidth* is higher for richer topologies
  - *Worst case measure* that models the potential of the network to transmit multiple packets
BW vs. communication patterns

- In wormhole switching, message passing can be modeled as:
  - \( t_s + \frac{\text{dim}}{\text{bw}} \), where \( t_s \) is the initial transmission cost to open the circuit
- Mesh of \( p \times p \) nodes, with the relative BW
- If every node communicates with one of its neighbors, all the parallel communications completed in a time: \( t_s + \frac{\text{dim}}{\text{bw}} \)
- If every node communicates with a random partner
  - \( \frac{(p \times p)}{4} \) messages traverse on average the bisection in one direction, and \( \frac{(p \times p)}{4} \) messages in the opposite direction
  - The \( p \) links on the cut are full-duplex
    - if the messages are optimally distributed over the \( p \) lines of the cut, on the average every link is traversed by \( \frac{(p \times p)}{4} / p = \frac{p}{4} \) messages in each direction, for a total size of \( \frac{p}{4} \times \frac{\text{dim}}{\text{bw}} \)
    - The \( \frac{p}{4} \) messages must be serialized (conflict + flow control)
- Total time \( t_s + \frac{p}{4} \times \frac{\text{dim}}{\text{bw}} \)
## Parameters and evaluation measures

<table>
<thead>
<tr>
<th></th>
<th>( N = k^n )</th>
<th>( BW = k^{n-1} )</th>
<th>( D = n (k-1) )</th>
<th>( d = 2n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>k-ary 1-cube</td>
<td>( N = k )</td>
<td>( BW = 1 )</td>
<td>( D = k-1 )</td>
<td>( d = 2n = 2 )</td>
</tr>
<tr>
<td>k-ary 2-cube</td>
<td>( N = k^2 )</td>
<td>( BW = k )</td>
<td>( D = 2(k-1) )</td>
<td>( d = 2n = 4 )</td>
</tr>
<tr>
<td>k-ary 3-cube</td>
<td>( N = k^3 )</td>
<td>( BW = k^2 )</td>
<td>( D = 3(k-1) )</td>
<td>( d = 2n = 6 )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2-ary ( n )-cube</td>
<td>( N = k^n = 2^n )</td>
<td>( BW = 2^{n-1} )</td>
<td>( D = n )</td>
<td>( d = n = \log N )</td>
</tr>
</tbody>
</table>

- If degree \( d \) of nodes is a factor that limits expandability, fix (and reduce) dimension \( n \) to reduce \( d \) \( (d = 2n) \)
  - fix \( n \) and thus \( d \)
  - expand by increasing \( k \)

- If the number of hops is important for routing, we need a small \( D = n (k-1) = n (\sqrt{n}N-1) \) and a large value of \( BW = k^{n-1} = (\sqrt{n}N)^{n-1} \) it is enough to increase dimension \( n \) for a given number \( N \) of nodes, by also increasing \( d \)
  - The 2-ary \( n \)-cube is an optimal choice in this case
  - \( D = \)
Parameters and evaluation measures

- In practice, which is the best topology?

- Topologies with small and fixed $n$ and $d$ are less expensive, allow for expandability, are easier to build, and perform very well
  - issues in packaging lines for $n$ greater than 3
  - high-dimension cubes are critical from the pin count and link cost viewpoint: in practice, they are forced to use a few-bit parallel lines for each link
  - low dimension cubes ($n = 2, 3$) are more feasible structures, and in practice they work well if parallel programs are written to exploit topology
Tree Networks

- In the tree network we have only a single path for every pair of communicating nodes
- Linear or star topologies can be considered as limit cases of a tree one
- We can have
  - direct networks, where all the nodes are communicating elements, such as processors/memory banks
  - indirect networks, where communicating elements are leaf nodes, whereas the internal nodes are switch boxes
- To deliver a message:
  - the message has to go up, till reaching an internal node that is the root of a sub-tree that include both the source and the destination of the message
- BW=1
Tree Networks

• In this kind of networks, the bottlenecks are the levels that are close to the tree root
  – if many leaf nodes on the left have to communicate with nodes on the right of the tree, all the message have to cross the switch associated with the root of the tree itself
• A solution requires increasing the number of links between nodes close to the root: fat tree
Fat tree

Realized with 2X2 switch boxes
Measures for other direct topologies

<table>
<thead>
<tr>
<th>Network</th>
<th>Diameter</th>
<th>Bisection Width</th>
<th>Arc Connectivity</th>
<th>Cost (No. of links)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completely-connected</td>
<td>1</td>
<td>(p^2/4)</td>
<td>(p - 1)</td>
<td>(p(p - 1)/2)</td>
</tr>
<tr>
<td>Star</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>(p - 1)</td>
</tr>
<tr>
<td>Complete binary tree</td>
<td>(2 \log((p + 1)/2))</td>
<td>1</td>
<td>1</td>
<td>(p - 1)</td>
</tr>
<tr>
<td>Linear array</td>
<td>(p - 1)</td>
<td>1</td>
<td>1</td>
<td>(p - 1)</td>
</tr>
<tr>
<td>2-D mesh, no wraparound</td>
<td>(2(\sqrt{p} - 1))</td>
<td>(\sqrt{p})</td>
<td>2</td>
<td>(2(p - \sqrt{p}))</td>
</tr>
<tr>
<td>2-D wraparound mesh</td>
<td>(2\lceil\sqrt{p}/2\rceil)</td>
<td>(2\sqrt{p})</td>
<td>4</td>
<td>(2p)</td>
</tr>
<tr>
<td>Hypercube</td>
<td>(\log p)</td>
<td>(p/2)</td>
<td>(\log p)</td>
<td>((p \log p)/2)</td>
</tr>
<tr>
<td>Wraparound k-ary d-cube</td>
<td>(d\lceil k/2 \rceil)</td>
<td>(2k^{d-1})</td>
<td>(2d)</td>
<td>(dp)</td>
</tr>
</tbody>
</table>

Degree \(d\)
Measures for other indirect topologies

<table>
<thead>
<tr>
<th>Network</th>
<th>Diameter</th>
<th>Bisection Width</th>
<th>Arc Connectivity</th>
<th>Cost (No. of links)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crossbar</td>
<td>1</td>
<td>$p$</td>
<td>1</td>
<td>$p^2$</td>
</tr>
<tr>
<td>Omega Network</td>
<td>$\log p$</td>
<td>$p/2$</td>
<td>2</td>
<td>$p/2$</td>
</tr>
<tr>
<td>Dynamic Tree</td>
<td>$2 \log p$</td>
<td>1</td>
<td>2</td>
<td>$p - 1$</td>
</tr>
</tbody>
</table>

Degree $d$
Impact of routing

• How does one compute the route that a message takes from source to destination?
  – Deterministic vs. adaptive
  – Routing must prevent deadlocks
    • for this reason, we use dimension-ordered or e-cube routing in k-ary n-cubes (deterministic, also simple to realize)
  – Routing must avoid hot-spots
    • adaptive
    • two-step routing has been theorized: a message from source $s$ to destination $d$ is first sent to a randomly chosen intermediate processor $i$ and then forwarded to destination $d$. 
E-cube deterministic routing

Routing a message from node $P_s$ (010) to node $P_d$ (111) in a three-dimensional hypercube using E-cube routing.

Deterministic order in routing over the 3 dimensions
Example of deadlock (w/o E-cube routing)

Message 0, 1, 2, and 3 need to go to nodes A, B, C, and D, respectively

0 and 2 go first horizontally and then vertically

1 and 3 go first vertically and then horizontally

There is a deadlock because all the buffers are occupied

With E-cube routing the cycles cannot occur
Process-to-Processor Mapping vs. Routing

- Often, we need to embed a known communication pattern into a given interconnection topology.
- We may have an algorithm designed for a given network, which we are porting to another topology.
- For these reasons, it is useful to understand mapping between graphs
  - embedding of a logical communication graph into to a physical interconnection graph
Process-to-Processor Mapping vs. Routing

- **Metrics of the mapping** \( G(V,E) \) in \( G'(V',E') \)

- **Congestion** of the mapping
  - The maximum number of edges mapped onto any edge in \( E' \)

- **Dilatation** of the mapping
  - The maximum number of links in \( E' \) on which any edge in \( E \) is mapped

- **Expansion** of the mapping
  - The ratio of the number of nodes in set \( V' \) to that in set \( V \)
Embedding of topologies

- Mapping of processes to processors
  - Suppose to have an algorithm designed for a given logic topology, e.g. a 3-D mesh, where all the communications are between neighbors, to be mapped onto a hypercube physical topology
  - The mapping is easier if the physical topology is richer than the logical one (in both nodes and edges)
    - Goal: a mapping where all the connections between processes are mapped on direct links (congestion and dilatation = 1)

- General remark:
  - In the current parallel machines is however difficult to control the mapping, since the physical topology is not visible by the programmer
  - Many programming environments only allow the parallelism degree to be specified (but programming environment for MPP, like the BluGene supercomputer)
  - Network+Routing+Switching : logically complete interconnection, where the cost of transmission depends less on the number of hops
  - Unfortunately the message conflicts (Congestion) must still be avoided to guarantee good performance
Figure 2.29  Impact of process mapping on performance: (a) underlying architecture; (b) processes and their interactions; (c) an intuitive mapping of processes to nodes; and (d) a random mapping of processes to nodes.
Example of mapping: embedding of a linear array over an hypercube

- A linear array composed of $2^d$ nodes (labels from 0 to $2^{d-1}$)

- The embedding is simple for a $d$-dimensional hypercube
  - mapping of node $i$ on node $G(i, d)$ of the hypercube

- Function $G(i, d)$ defined as follows:

\[
G(0, 1) = 0 \\
G(1, 1) = 1 \\
G(i, x + 1) = \begin{cases} 
G(i, x), & i < 2^x \\
2^x + G(2^x+1 - 1 - i, x), & i \geq 2^x 
\end{cases}
\]
Example of mapping: embedding of a linear array over an hypercube

- $G(0, 3) = G(0, 2) = G(0, 1) = 000_2$
- $G(1, 3) = G(1, 2) = G(1, 1) = 001_2$
- $G(2, 3) = G(2, 2) = 2^1 + G(2^2 -1-2, 1) = 2^1 + G(1, 1) = 2^1 + 1 = 011_2$
- $G(3, 3) = G(3, 2) = 2^1 + G(2^1 -1-3, 1) = 2^1 + G(0, 1) = 2^1 + 0 = 010_2$
- $G(4, 3) = 2^2 + G(2^3 -1-4, 3) = 2^2 + G(3, 3) = 2^2 + 2 = 110_2$
- .....
Example of mapping: embedding of a hypercube over a 2-D mesh

(a) $P = 16$

(b) $P = 32$
SM multiprocessor: cache coherence, synchronization issues
SM multiprocessors with cache coherence

- SMP (Symmetric MP) multiprocessors *bus-based*
  - UMA, tightly coupled (fine grain *sharing*)
  - accessed data are automatically copied and replicated in the various caches
  - data sharing and synchronizations via load/store on shared variables

- Programmer tasks
  - reduce synchronizations to avoid *software lockout*
  - limit the portion of memory actually shared by threads (increase the *logically private memory* per thread)
  - allocate data and orchestrate the data accesses in order to exploit locality, and thus use caches effectively

![Diagram of memory and I/O devices connected by a bus]

```
P^n

P

$ $

Mem

Bus

I/O devices
```
Cache advantages

- The caches reduce the bus bottleneck
  - many load/store are not transformed in bus transaction, since they can simply completed by accessing cached data

- Overall, the caches reduce the average latency for data access

- Work very well for read-only data

- What does it happen if replicated/cached data are modified by some processors?
Example Cache Coherence Problem

The processors see different values of $u$ after event 3

- cache write-back
  - the value written to memory depends on when the cache flushes
  - the processors that access the memory can find versions of data that are not updated w.r.t. the corresponding replicas in cache (events 4 and 5)

- cache write-through
  - processors that access the memory always find coherent copies of data (event 5)
  - processors can still access cached copies that are non coherent (event 4)
Snoopy Cache coherence protocol

- A bus natively permits broadcast communications
- Caches know which data lines they stores in cache blocks
- The Cache Controller “snoops” all the bus transactions
  - a transaction is relevant for a cache if the referred data line (univocally identified by the (block) address) is present in cache
  - the possible actions to guarantee the cache coherence are:
    - invalidate, update
  - the right action to carry out depends on both the state of cache block and the protocol type
    - write-update or write-invalidate (coherence among caches)
    - write-through or write-back (coherence between cache and memory)
Invalidation vs. Update

- **Update**
  - send a transaction on the bus to *update* all the copies of the data line that are possibly replicated in the various caches
  - A bus transaction corresponds to transmitting:
    - The data line address
    - The modified cache block

- **Invalidation**
  - send a transaction on the bus to *invalidate* all the copies of the data line that are possibly replicated in the various caches
  - A bus transaction corresponds to transmitting:
    - The data line address
Invalidate vs. Update

• Update
  – cons: can waste unnecessarily the bandwidth of the bus
    • when a cache block is updated remotely, but is no longer read by the associated processor
    • subsequent writes by the same processor cause multiple updates
  – pros: multiple R/W copies are kept coherent after each write.
    • This avoids misses for each subsequent read access (thus saving bus bandwidth)

• Invalidate
  – pros: multiple writes by the same processors do not cause any additional bus traffic
  – cons: an access that follows an invalidation causes a miss (which in turn causes a bus transaction)

⇒ Pros and Cons of the two approaches depend on the programs and their patterns of reads/writes

• Modern systems use
  – write-back and write-invalidate, since it is the best compromise to reduce bus traffic
Block size and False sharing

- **Coherence protocols** work in terms of **cache blocks**, rather than single words/bytes

- The block size plays an important role in the coherence protocol
  - with small blocks, the protocols are more efficient (less data to transmit when update or flush)
  - large blocks are better for spatial locality

- **False sharing**
  - consider two **unrelated variables** (e.g., variables that are logically private to distinct threads), which are allocated in the same block
  - write accesses to a blocks by different processors running the threads are considered in **conflicts** by the **coherence protocol**
    - even if the two processors access disjoint words of the same block
  - thus, it is needed to put on the same block **related variables**
    - e.g., all of which are logically private for a given thread
    - compiling techniques, programmer …
Write-through Write-Invalidate Protocol

• Simple protocol, with only 2 states for each block
  – similar to uniprocessor cache (a single Valid bit)

• The FSM (Finite State Machine) models the state transition for each cache block, triggered by
  – a CPU request (PrRd e PrWr)
  – a bus transaction (BusRd e BusWr)

• The Writes (BusWr) invalidate all the blocks that are replicated in the caches
  – we can have multiple readers, but only a single writer (which invalidates all the replicated blocks)

• Note that, due to the write-through, the writes do not lead to block read from memory (the block continues to be invalid)

• The bus transactions are atomic, since they are arbitrated
Example of Write-back Write-Invalidate
Protocol W-back (W-invalidate)

- Extend the controller of a Write-back cache of a uniprocessors
  - 2 request types from the CPU (PrRd, PrWr), with a Replace in case of miss/conflict
  - 3 states
    - Invalid, Valid (clean), Modified (dirty)
  - In a uniprocessors, 2 transactions suffice:
    - read (BusRd), write-back (BusWB)
    - entire cache-blocks are transferred

Extension for SMPs (W-invalidate):

=> Valid considered as “Shared”
Modified becomes a synonym for “Exclusive”
a block in state Modified must be W-back even if
external requests arrive for the block

=> it is needed to introduce a new type of bus transaction for invalidating

- Read-exclusive: read for modifying
  (I -> M e V->M): in this way all processors know that a block is acquired for writing and invalidate their block copies
MSI Invalidate Protocol

- Reads (PrRd) cause the blocks to be assigned a “shared” state S (that replaces V)
  - even if a single copy of the cache line exists in the various caches
- The exclusive ownership must be acquired before a Write
  - BusRdx invalidate all other blocks
  - If the block is Modified (M) in another cache, then flush
  - BusRdx even when we have a hit and the state is S
    - promotion to M
- The requests from the CPU for replacing a block (Replace), as a consequence of a miss and conflict (block to reuse), are not shown
  - S->I, M->I as in the previous FSM
Example of Write-Back Write-invalidate Protocol

PrRd U

P₀

S 5

PrRd U

P₁

S 7

PrRd U

P₂

S 7

PrWr U 7

U: 7

Memory

BusRd U

BusRd U

BusRdx U

BusRD U

Flush

I/O devices
Example of execution (3 state coherence protocol)

<table>
<thead>
<tr>
<th>Time</th>
<th>Instruction at Processor 0</th>
<th>Instruction at Processor 1</th>
<th>Variables and their states at Processor 0</th>
<th>Variables and their states at Processor 1</th>
<th>Variables and their states in Global mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>reading x</td>
<td>reading y</td>
<td>x = 5, S</td>
<td>y = 12, S</td>
<td>x = 5, S</td>
</tr>
<tr>
<td></td>
<td>x = x + 1</td>
<td>y = y + 1</td>
<td>x = 6, D</td>
<td>y = 13, D</td>
<td>x = 5, I</td>
</tr>
<tr>
<td></td>
<td>reading y</td>
<td>reading x</td>
<td>y = 13, S</td>
<td>y = 13, S</td>
<td>y = 12, I</td>
</tr>
<tr>
<td></td>
<td>x = x + y</td>
<td>y = x + y</td>
<td>x = 6, S</td>
<td>x = 6, S</td>
<td>x = 6, S</td>
</tr>
<tr>
<td></td>
<td>x = x + 1</td>
<td>y = y + 1</td>
<td>x = 19, D</td>
<td>x = 6, I</td>
<td>x = 6, I</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>y = 13, I</td>
<td>y = 19, D</td>
<td>y = 13, I</td>
</tr>
</tbody>
</table>

- **M = D = dirty/modified**
- **I = invalid**
- **S = Shared**
MESI Protocol

- Both the Pentium and the PowerPC adopt a protocol called MESI
  - Write-invalidate Write-back
- Compared to MSI, state S (read-only) is further split
  - E (Exclusive): the block is read-only by a single CPU
  - S (Shared): the block is read-only by many CPUs

- Pros:
  - the promotion E->M can be performed without any bus transaction (BusRdx) for invalidating the possible replicated cache blocks
  - the E state guarantees that a single read-only copy exists in all CPU caches
Synchronization Mechanisms

- The synchronization to access a resource takes place reading/writing a memory location (lock, semaphore)
- Consider a variable for a lock (binary semaphore) with 2 states
  - 0: go-ahead
  - 1: stop
- We know the problem ..... 
  - read the variable, if it is equal to 0, decide to access, and write 1 to this variable to signal the blocked status of the associated resource
  - between a read and a write another write could occur (race condition)
=> the read and the subsequent write must occur in an atomic manner

- we cannot guarantee the atomicity by only disabling the interrupt, as in uniprocessors
  - multiple threads can contemporarily perform the same access by running on different SMP processors
Synchronization Mechanisms

- Different architectures have special instructions
  - atomic exchange
    - exchange the value of a register and the one of a memory location
  - test-and-set (old method)
    - test the value of a memory variables. Set the variable only if the test is ok
  - load locked & store conditional
    - recent mechanism present in some processors
    - first perform a load locked
    - the next store conditional fails if the value stored in the memory location has been modified in the meanwhile
Synchronization Mechanisms

- Simple *spin-lock* with an *atomic exchange*

  ```
  li R3, #1 ; store constant 1 to reg. R3
  lockit: exch R3, 0(R1) ; exchange R3 with the memory
                   ; line addressed by R1
  bnez R3, lockit ; if R3!=0, it’s already locked
  ```

- The above solution is not the best one ....
  - *the continuous writes* entailed by *exch*, along with the *coherence mechanism*, can generate heavy traffic on the bus
  - a more economic method is the following: we continue looping with simple reads, which normally access local cached copies, without generating bus traffic:

  ```
  lockit: lw R3, 0(R1) ; read in R3 the memory line
                       ; addressed by R1
  bnez R3, lockit ; if R3!=0, it’s already locked
  li R3, #1 ; store constant 1 to reg. R3
  exch R3, 0(R1) ; exchange R3 with the memory
                  ; line addressed by R1
  bnez R3, lockit ; if R3!=0, it’s already locked
  ```
Spin-lock and busy-waiting

- Busy-waiting
  - threads that compete to access a critical region by *spin-locking* are unfortunately engaged in *busy waiting*
  - we have to avoid exploiting spin locks to access large critical regions
  - e.g.: use the spin locks to make the code implementing the P or V semaphore mechanisms *atomic* (this is a small critical section)

Semaphore = Synchronization variable.

P(s): [ while (s == 0)
  wait();
  s--; ]

V(s): [ s++; ]

Programmer use P and V to make larger code portions (*critical sections*) atomic:

[ atomic code ] = P(s); ...code...; V(s)
Sequential consistency

- The cache coherence mechanism guarantees that multiple processors have a consistent view of the shared memory.
- The one we have assumed is a type of consistency said sequential consistency.
  - “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]
- It is the same as the following:

- The memory operations arriving from the various processors must be executed atomically [issue, execute, complete]
Sequential consistency

• Example
  – Suppose processors 1 and 2 send pairs of requests to the memory according to the following order:

    (1a) A=1 (W)           (2a) print B (R)
    (1b) B=2 (W)           (2b) print A (R)

    the possible sequential consistent (SC) orders are

    1^: 1a-> 1b-> 2a-> 2b
    2^: 1a-> 2a-> 1b-> 2b
    3^: 1a-> 2a-> 2b-> 1b
    4^: 2a-> 2b-> 1a-> 1b
    5^: 2a-> 1a-> 2b-> 1b
    6^: 2a-> 1a-> 1b-> 2b

• If at the beginning A=0 and B=0
  – Processor 2 cannot read both B=2 and A=0
  – To read A=0 it is needed that  2b->1a
    • 4^ order
  – To read B=2, the orders should 1b->2a
    • 1^ order
Sequential and Relaxed Consistency

• Maintaining the sequential consistency can lead to performance issues
  – We have to pay the latency for update / invalidate/ flush without overlapping useful computations
  – We cannot put a store request in a write buffer for delaying the completion, and meanwhile continuing with the other computations
  – If concurrent writes modify distinct data lines, it is not mandatory maintain a sequential consistency among them, for achieving a correct semantics. The same for the reads ...

• Possible solutions
  – Implementations that guarantee SC and, at the same time, hide the high latency for completing memory operations
    • very complex .... especially in large scale multiprocessors
  – Implementations that realize a consistency model less restrictive (relaxed), thus allowing for faster implementations
Relaxed consistency

• SC simplifies the programming model for programmers

• Relaxed models
  – remove some constraints on the overlapping and the reordering of the memory operations
  – thus improving the efficiency of the implementation

• Usually the shared memory programs are synchronized
  – the programmer assumes that a concurrent program is correct if the shared data are accessed after suitable calls to synchronization primitives (BTW: the shared data are exactly the ones that lead to consistency issues ....)

• The idea of the relaxed model of consistency is
  – to force a strict sequential order only between operations that are explicitly ordered by explicit synchronizations
    • e.g., a lock before a READ has to force the completion of a previous WRITE (which is usually followed by an unlock)
  – the synchronized program, executed according to a relaxed model, has to be same semantics of the program executed on a system based on a SC model
Cache coherence in large scale multiprocessors

- **Snoopy protocols** are intimately tied to multiprocessors based on broadcast shared networks such as buses.
  - This is because all cache controllers must “snoop” all the bus transactions.
  - Clearly, broadcasting all the important processors’ memory operations to all the other processors is not a scalable solution.

- **Memory Coherence on large scale multiprocessors**
  - Replace the bus by a **scalable network**
  - The snooping protocol does not work for these scalable networks
    - since the transmission media is not a shared network like a bus, communications are point-to-point
Directory-based Cache Coherence

- The *snooping protocol* is completely *decentralized*  
  - There is no *single place* where we can check whether the data has been replicated in some caches, or whether these replicas have been modified  
  - In the snooping protocol we exploit the *broadcast* communications allowed by busses, and the ability of the cache controller to *snoop* such communications

- **Directory-based** cache-coherence protocol  
  - The information about the sharing/replications of memory blocks are instead centralized in the *directory*

- Scalable networks of multiprocessors permit point-to-point communications  
  - It is possible to selectively send “coherence messages” to only those processors/caches that must be notified  
  - The directory contains the information to select the processors/caches to notify
Directory-based Cache Coherence

- Simple schema
  - the directory has an entry for each memory block
  - for each block, it stores information about which are the processors/caches (along with the associated state) that store a copy of this block
  - with a centralized directory, the information stored in the directory is proportional to
    - \( \text{no. Procs/Caches} \times \text{no. Blocks} \)

- example of states associated with each memory block / processors (write invalidate):
  - Shared (R), Uncached, Exclusive/Modified (W)
  - The Processor whose cache stores a block with a state of Exclusive/Modified (W), becomes the owner of that memory block

- Example of invalidation: a message must be sent to the directory, which selectively sends an invalidation message to all the processor/caches that hold the line in \textit{shared state}
Directory-based Cache Coherence

• The centralized directory is a bottleneck of the system
  – all the messages to access a new block must be queues towards the same centralization point
  – note that while the snooping protocol is completely decentralized, this directory-based one is instead completely centralized

⇒ Solution: decentralize the directory, a sub-directory for each node, managing a group of memory blocks

It works for NUMA shared memory architectures, where a node is a single processors with local memory (or an SMP UMA multiprocessor)

introduce the concept of home node, that is a node that stores both a primary copy a memory block and the corresponding entry of the directory
A completely **centralized directory**. Number of bits equal to the numbers of processors/caches. UMA organization.
Directory-based organizations

- With a pure NUMA organization, still bits are proportional to procs
- Hierarchical (NUMA of UMA SMPs), the number of bits is reduced: bits per each UMA SMP (cluster) rather than per proc
Directory-based Protocol - Write-invalidate

- To **invalidate**, as a consequence of a write, **two messages are exchanged**
  1. node that generated the write $\rightarrow$ directory
  2. directory $\rightarrow$ one or more nodes to invalidate (multicast)

where a node is a single processor/cache or a UMA multiprocessor

- Note that in the **snooping** protocol, the **two step above are combined**: a single broadcast to all nodes

- Types of messages between directory and nodes
  - **local node**: where the request has been originated
  - **home node**: node where the memory block and the associated directory entry are allocated

- Physical addresses **distributed statically**
  - given a physical address, the **node number** where the memory block and the associated directory entry are allocated is known
  - e.g.: the **high-order bits** of the address used as the node number
## Directory-based Protocol - Write-invalidate

<table>
<thead>
<tr>
<th>Message type</th>
<th>Source</th>
<th>Destination</th>
<th>Message contents</th>
<th>Function of this message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read miss</td>
<td>local cache</td>
<td>home directory</td>
<td>P, A</td>
<td>Processor P has a read miss at address A; request data and make P a read sharer.</td>
</tr>
<tr>
<td>Write miss</td>
<td>local cache</td>
<td>home directory</td>
<td>P, A</td>
<td>Processor P has a write miss at address A; request data and make P the exclusive owner.</td>
</tr>
<tr>
<td>Invalidate</td>
<td>local cache</td>
<td>home directory</td>
<td>A</td>
<td>Request to send invalidates to all remote caches that are caching the block at address A.</td>
</tr>
<tr>
<td>Invalidate</td>
<td>home directory</td>
<td>remote cache</td>
<td>A</td>
<td>Invalidate a shared copy of data at address A.</td>
</tr>
<tr>
<td>Fetch</td>
<td>home directory</td>
<td>remote cache</td>
<td>A</td>
<td>Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.</td>
</tr>
<tr>
<td>Fetch/invalidate</td>
<td>home directory</td>
<td>remote cache</td>
<td>A</td>
<td>Fetch the block at address A and send it to its home directory; invalidate the block in the cache.</td>
</tr>
<tr>
<td>Data value reply</td>
<td>home directory</td>
<td>local cache</td>
<td>D</td>
<td>Return a data value from the home memory.</td>
</tr>
<tr>
<td>Data write back</td>
<td>remote cache</td>
<td>home directory</td>
<td>A, D</td>
<td>Write back a data value for address A.</td>
</tr>
</tbody>
</table>

- **On a Write miss**
  - broadcast on the bus in the snooping schema
  - messages of Fetch/Invalidate, sent selectively by the directory controller
Directory-based Protocol - Write-invalidate

- Like the snooping protocol
  - a modified/written cache block must be assigned a state of modified/exclusive, and there must exist a single copy in the various cache
  - a cached block with a shared state must be the exact copy of the primary copy kept in memory, and many copies can co-exist in the various caches

- In the directory-based protocol, the directory is a further actor that participates in the protocol along with the various cache controllers
  - a message sent to the directory causes two actions
    - update the state of the directory
    - the delivery of additional messages to satisfy the request
The directory-based protocol discussed here is the Write-invalidate protocol. This protocol is used in cache-coherent distributed memory systems where multiple processors share a common memory. The protocol is designed to ensure consistency and avoid conflicts when multiple processors access the same memory locations.

### Cache-controller side

- **The gray labels and edges represent external events (from directory):**
  - The protocol involves external events from the directory, such as invalidation requests, which affect the state of the cache.

- **Both the messages:**
  - **Read miss:** When a processor requests a data value that is not in its local cache but is in another processor's cache, a read miss occurs, and a request is sent to the directory to retrieve the data.
  - **Write miss:** When a processor attempts to write data to a memory location, and the location is invalidated by another processor, a write miss occurs, and the data is retrieved from the directory.

- **Data value reply messages:**
  - These messages are sent from the directory to the processor to acknowledge the receipt of data, ensuring that the processor's cache is updated with the correct data.

The diagram illustrates the state transitions between different cache states (Invalid, Shared, Modified) and the messages that are exchanged between the processor and the directory to maintain consistency.
Directory-based Protocol - Write-invalidate

Directory-controller side

Uncached

Data value reply; Sharers = \{P\}

Read miss

Exclusive (read/write)

Data value reply; Sharers = \{P\}

Write miss

Data write back

Read miss

Fetch; data value reply; Sharers = Sharers + \{P\}

Write miss

Invalidate; Sharers = \{P\}; data value reply

Data value reply
Sharers = Sharers + \{P\}

Write miss

Fetch/invalidate
Sharers = \{P\}

Shared (read only)
Parallel architectures: wrap-up

Interconnection networks and communication mechanisms are similar

**CC**: Cache Coherent
**NC**: No Cache
**COMA**: Cache Only Memory Access