Parallel Computer Architecture

• A parallel computer is a collection of processing elements that cooperate to solve large problems.

• Broad issues involved:
  – Resource Allocation:
    • Number of processing elements (PEs).
    • Computing power of each element.
    • Amount of physical memory used.
  – Data access, Communication and Synchronization
    • How the elements cooperate and communicate.
    • How data is transmitted between processors.
    • Abstractions and primitives for cooperation.
  – Performance and Scalability:
    • Performance enhancement of parallelism: Speedup.
    • Scalability of performance to larger systems/problems.
The Goal of Parallel Computing

• Goal of applications in using parallel machines: Speedup

\[
\text{Speedup (} p \text{ processors)} = \frac{\text{Performance (} p \text{ processors)}}{\text{Performance (1 processor)}}
\]

• For a fixed problem size (input data set), performance = \(1/\text{time}\)

\[
\text{Speedup}_{\text{fixed problem}} (p \text{ processors)} = \frac{\text{Time (1 processor)}}{\text{Time (} p \text{ processors)}}
\]
Elements of Modern Computers

- Computing Problems
- Algorithms and Data Structures
- Mapping
- High-level Languages
- Binding (Compile, Load)
- Performance Evaluation
- Mapping
- Hardware Architecture
- Operating System
- Applications Software
- Compile, Load

EECC756 - Shaaban
Approaches to Parallel Programming

**Programmer**

Source code written in sequential languages C, C++, FORTRAN, LISP ..

Parallelizing compiler

Parallel object code

Execution by runtime system

(a) Implicit Parallelism

**Programmer**

Source code written in concurrent dialects of C, C++, FORTRAN, LISP ..

Concurrency preserving compiler

Concurrent object code

Execution by runtime system

(b) Explicit Parallelism
Evolution of Computer Architecture

Scalar
  - Sequential
  - Lookahead
    - I/E Overlap
    - Functional Parallelism
      - Multiple Func. Units
      - Pipeline
        - Implicit Vector
        - Explicit Vector
          - SIMD
            - Memory-to-Memory
          - Register-to-Register
            - MIMD
              - SIMD
              - MIMD
                - Associative Processor
                - Processor Array
                - Multicomputer
                - Multiprocessor

Massively Parallel Processors (MPPs)

I/E: Instruction Fetch and Execute
SIMD: Single Instruction stream over Multiple Data streams
MIMD: Multiple Instruction streams over Multiple Data streams
Programming Models

- Programming methodology used in coding applications.
- Specifies communication and synchronization.

- Examples:
  - Multiprogramming:
    No communication or synchronization at program level
  - Shared memory address space:
  - Message passing:
    Explicit point to point communication.
  - Data parallel:
    More regimented, global actions on data.
    - Implemented with shared address space or message passing.
Flynn’s 1972 Classification of Computer Architecture

• Single Instruction stream over a Single Data stream (SISD): Conventional sequential machines.

• Single Instruction stream over Multiple Data streams (SIMD): Vector computers, array of synchronized processing elements.

• Multiple Instruction streams and a Single Data stream (MISD): Systolic arrays for pipelined execution.

• Multiple Instruction streams over Multiple Data streams (MIMD): Parallel computers:
  • Shared memory multiprocessors.
  • Multicomputers: Unshared distributed memory, message-passing used instead.
Current Trends In Parallel Architectures

• The extension of “computer architecture” to support communication and cooperation:
  
  − OLD: Instruction Set Architecture
  − NEW: Communication Architecture

• Defines:
  
  − Critical abstractions, boundaries, and primitives (interfaces).
  − Organizational structures that implement interfaces (hardware or software).

• Compilers, libraries and OS are important bridges today.
Models of Shared-Memory Multiprocessors

• The Uniform Memory Access (UMA) Model:
  – The physical memory is shared by all processors.
  – All processors have equal access to all memory addresses.

• Distributed memory or Nonuniform Memory Access (NUMA) Model:
  – Shared memory is physically distributed locally among processors.

• The Cache-Only Memory Architecture (COMA) Model:
  – A special case of a NUMA machine where all distributed main memory is converted to caches.
  – No memory hierarchy at each processor.
Models of Shared-Memory Multiprocessors

**Uniform Memory Access (UMA) Model**
- Interconnect: Bus, Crossbar, Multistage network
- P: Processor
- M: Memory
- C: Cache
- D: Cache directory

**Distributed memory or Nonuniform Memory Access (NUMA) Model**

**Cache-Only Memory Architecture (COMA)**
Message-Passing Multicomputers

- Comprised of multiple autonomous computers (nodes).
- Each node consists of a processor, local memory, attached storage and I/O peripherals.
- Programming model is more removed from basic hardware operations.
- Local memory is only accessible by local processors.
- A message-passing network provides point-to-point static connections among the nodes.
- Inter-node communication is carried out by message passing through the static connection network.
- Process communication achieved using a message-passing programming environment.
Convergence: Generic Parallel Architecture

- A generic modern multiprocessor

Node: processor(s), memory system, plus communication assist

- Network interface and communication controller
- Scalable network
- Convergence allows lots of innovation, now within framework
  - Integration of assist with node, what operations, how efficiently...
Fundamental Design Issues

• At any layer, interface (contract) aspect and performance aspects
  – **Naming**: How are logically shared data and/or processes referenced?
  – **Operations**: What operations are provided on these data
  – **Ordering**: How are accesses to data ordered and coordinated?
  – **Replication**: How are data replicated to reduce communication?
  – **Communication Cost**: Latency, bandwidth, overhead, occupancy

• Understand at programming model first, since that sets requirements

• Other issues
  – **Node Granularity**: How to split between processors and memory?
Synchronization

Mutual exclusion (locks):
- Ensure certain operations on certain data can be performed by only one process at a time.
- Room that only one person can enter at a time.
- No ordering guarantees.

Event synchronization:
- Ordering of events to preserve dependencies.
  - e.g. Producer → Consumer of data
- Three main types:
  - Point-to-point
  - Global
  - Group
Communication Cost Model

Comm Time per message = Overhead + Assist Occupancy
+ Network Delay + Size/Bandwidth + Contention

= \( o_v + o_c + l + n/B + T_c \)

Overhead = Time to initiate the transfer
Occupancy = The time it takes data to pass through the slowest component on
the communication path. Limits frequency of communication operations.

\( l + n/B + T_c \) = Network Latency, can be hidden by overlapping with other processor
operations

• Overhead and assist occupancy may be \( f(n) \) or not.
• Each component along the way has occupancy and delay.
  – Overall delay is sum of delays.
  – Overall occupancy (1/bandwidth) is biggest of occupancies.

Comm Cost = frequency * (Comm time - overlap)
Conditions of Parallelism: Data Dependence

1 True Data or Flow Dependence: A statement $S_2$ is data dependent on statement $S_1$ if an execution path exists from $S_1$ to $S_2$ and if at least one output variable of $S_1$ feeds in as an input operand used by $S_2$ denoted by $S_1 \rightarrow S_2$

2 Antidependence: Statement $S_2$ is antidependent on $S_1$ if $S_2$ follows $S_1$ in program order and if the output of $S_2$ overlaps the input of $S_1$ denoted by $S_1 +\rightarrow S_2$

3 Output dependence: Two statements are output dependent if they produce the same output variable denoted by $S_1 \circ\rightarrow S_2$
Conditions of Parallelism: Data Dependence

4 I/O dependence: Read and write are I/O statements. I/O dependence occurs not because the same variable is involved but because the same file is referenced by both I/O statements.

5 Unknown dependence:
   - Subscript of a variable is subscribed (indirect addressing).
   - The subscript does not contain the loop index.
   - A variable appears more than once with subscripts having different coefficients of the loop variable.
   - The subscript is nonlinear in the loop index variable.
Data and I/O Dependence: Examples

A -
S1: Load R1,A
S2: Add R2, R1
S3: Move R1, R3
S4: Store B, R1

B -
S1: Read (4), A(I) /Read array A from tape unit 4/
S2: Rewind (4) /Rewind tape unit 4/
S3: Write (4), B(I) /Write array B into tape unit 4/
S4: Rewind (4) /Rewind tape unit 4/

I/O dependence caused by accessing the same file by the read and write statements
Conditions of Parallelism

- **Control Dependence:**
  - Order of execution cannot be determined before runtime due to conditional statements.

- **Resource Dependence:**
  - Concerned with conflicts in using shared resources including functional units (integer, floating point), memory areas, among parallel tasks.

- **Bernstein’s Conditions:**
  Two processes $P_1$, $P_2$ with input sets $I_1$, $I_2$ and output sets $O_1$, $O_2$ can execute in parallel (denoted by $P_1 || P_2$) if:

  $I_1 \cap O_2 = \emptyset$

  $I_2 \cap O_1 = \emptyset$

  $O_1 \cap O_2 = \emptyset$
Bernstein’s Conditions: An Example

- For the following instructions $P_1, P_2, P_3, P_4, P_5$ in program order and
  - Instructions are in program order
  - Each instruction requires one step to execute
  - Two adders are available

$P_1: C = D \times E$
$P_2: M = G + C$
$P_3: A = B + C$
$P_4: C = L + M$
$P_5: F = G \div E$

Using Bernstein’s Conditions after checking statement pairs:

$P_1 \parallel P_5, \quad P_2 \parallel P_3, \quad P_2 \parallel P_5, \quad P_5 \parallel P_3, \quad P_4 \parallel P_5$

Parallel execution in three steps assuming two adders are available per step

Dependence graph:
Data dependence (solid lines)
Resource dependence (dashed lines)
Theoretical Models of Parallel Computers

• Parallel Random-Access Machine (PRAM):
  – $n$ processor, global shared memory model.
  – Models idealized parallel computers with zero synchronization or memory access overhead.
  – Utilized parallel algorithm development and scalability and complexity analysis.

• PRAM variants: More realistic models than pure PRAM
  – **EREW-PRAM**: Simultaneous memory reads or writes to/from the same memory location are not allowed.
  – **CREW-PRAM**: Simultaneous memory writes to the same location is not allowed.
  – **ERCW-PRAM**: Simultaneous reads from the same memory location are not allowed.
  – **CRCW-PRAM**: Concurrent reads or writes to/from the same memory location are allowed.
Example: sum algorithm on P processor PRAM

**Input:** Array A of size \( n = 2^k \) in shared memory

**Initialized local variables:**
- the order \( n \),
- number of processors \( p = 2^q \leq n \),
- the processor number \( s \)

**Output:** The sum of the elements of A stored in shared memory

begin
1. for \( j = 1 \) to \( l = n/p \) do
   
   Set \( B(l(s - 1) + j) = A(l(s-1) + j) \)

2. for \( h = 1 \) to \( \log n \) do
   
   2.1 if \( (k - h - q \geq 0) \) then
       
       for \( j = 2^{k-h-q}(s-1) + 1 \) to \( 2^{k-h-q}s \) do
       
       Set \( B(j) = B(2j - 1) + B(2s) \)

   2.2 else if \( (s \leq 2^{k-h}) \) then
       
       Set \( B(s) = B(2s - 1) + B(2s) \)

3. if \( (s = 1) \) then set \( S = B(1) \)
end

Running time analysis:
- Step 1: takes \( O(n/p) \) each processor executes \( n/p \) operations
- The \( h \)th of step 2 takes \( O(n / (2^h p)) \) since each processor has to perform \( n / (2^h p) \) operations
- Step three takes \( O(1) \)
- Total Running time:

\[
T_p(n) = O\left( \frac{n}{p} + \sum_{h=1}^{\log n} \left[ \frac{n}{2^h p} \right] \right) = O\left( \frac{n}{p} + \log n \right)
\]
Example: Sum Algorithm on P Processor PRAM

For \( n = 8 \) \( p = 4 \)

Processor allocation for computing the sum of 8 elements on 4 processor PRAM

Operation represented by a node is executed by the processor indicated below the node.
Example: Asynchronous Matrix Vector Product on a Ring

- **Input:**
  - \( n \times n \) matrix \( A \); vector \( x \) of order \( n \)
  - The processor number \( i \). The number of processors
  - The \( i \)th submatrix \( B = A(1:n, (i-1)r +1 ; ir) \) of size \( n \times r \) where \( r = n/p \)
  - The \( i \)th subvector \( w = x(i - 1)r + 1 : ir) \) of size \( r \)

- **Output:**
  - Processor \( P_i \) computes the vector \( y = A_1x_1 + \ldots A_ix_i \) and passes the result to the right
  - Upon completion \( P_1 \) will hold the product \( Ax \)

Begin

1. Compute the matrix vector product \( z = Bw \)
2. If \( i = 1 \) then set \( y_1 = 0 \)
   
   else receive \( (y, \text{left}) \)
3. Set \( y_i = y + z \)
4. send \( (y, \text{right}) \)
5. If \( i = 1 \) then receive \( (y, \text{left}) \)

End

\[
T_{\text{comp}} = \frac{k(n^2/p)}{}
\]
\[
T_{\text{comm}} = p(l + mn)
\]
\[
T = T_{\text{comp}} + T_{\text{comm}} = \frac{k(n^2/p)}{} + p(l + mn)
\]
Levels of Parallelism in Program Execution

- **Level 5**: Jobs or programs (Multiprogramming)
- **Level 4**: Subprograms, job steps or related parts of a program
- **Level 3**: Procedures, subroutines, or co-routines
- **Level 2**: Non-recursive loops or unfolded iterations
- **Level 1**: Instructions or statements

Increasing communications demand and mapping/scheduling overhead:

- Coarse Grain
- Medium Grain
- Fine Grain

Higher degree of Parallelism
Limited Concurrency: Amdahl’s Law

– Most fundamental limitation on parallel speedup.
– If fraction $s$ of sequential execution is inherently serial,
  
speedup $\leq \frac{1}{s}$

– Example: 2-phase calculation,
  • sweep over $n$-by-$n$ grid and do some independent computation.
  • sweep again and add each value to global sum.

– Time for first phase $= \frac{n^2}{p}$
– Second phase serialized at global variable, so time $= n^2$

– Speedup $\leq \frac{2n^2}{\frac{n^2}{p} + n^2}$ or at most 2

– Possible Trick: divide second phase into two:
  • Accumulate into private sum during sweep.
  • Add per-process private sum into global sum.

– Parallel time is $\frac{n^2}{p} + \frac{n^2}{p} + p$, and speedup at best $\frac{2n^2}{2n^2 + p^2}$
Parallel Performance Metrics

Degree of Parallelism (DOP)

• For a given time period, DOP reflects the number of processors in a specific parallel computer actually executing a particular parallel program.

• Average Parallelism:
  - Given maximum parallelism = \( m \)
  - \( n \) homogeneous processors
  - Computing capacity of a single processor \( \Delta \)
  - Total amount of work \( W \) (instructions, computations):

\[
W = \Delta \int_{t_1}^{t_2} DOP(t) \, dt \quad \text{or as a discrete summation} \quad W = \Delta \sum_{i=1}^{m} i \cdot t_i
\]

Where \( t_i \) is the total time that \( DOP = i \) and \( \sum_{i=1}^{m} t_i = t_2 - t_1 \)

The average parallelism \( A \):

\[
A = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} DOP(t) \, dt \quad \text{In discrete form} \quad A = \left( \sum_{i=1}^{m} i \cdot t_i \right) / \left( \sum_{i=1}^{m} t_i \right)
\]
Example: Concurrency Profile of A Divide-and-Conquer Algorithm

- Execution observed from $t_1 = 2$ to $t_2 = 27$
- Peak parallelism $m = 8$
- $A = \frac{\left( \sum_{i=1}^{m} i \cdot t_i \right)}{\sum_{i=1}^{m} t_i}$

$$A = \frac{(1 \cdot 5 + 2 \cdot 3 + 3 \cdot 4 + 4 \cdot 6 + 5 \cdot 2 + 6 \cdot 2 + 8 \cdot 3)}{5 + 3 + 4 + 6 + 2 + 2 + 3} = \frac{93}{25} = 3.72$$
Steps in Creating a Parallel Program

4 steps:

Decomposition, Assignment, Orchestration, Mapping.

- Done by programmer or system software (compiler, runtime, ...).
- Issues are the same, so assume programmer does it all explicitly.
Summary of Parallel Algorithms Analysis

- Requires characterization of multiprocessor system and algorithm.
- Historical focus on algorithmic aspects: partitioning, mapping.
- PRAM model: data access and communication are free:
  - Only load balance (including serialization) and extra work matter:
    \[ \text{Speedup} \leq \frac{\text{Sequential Instructions}}{\text{Max} (\text{Instructions} + \text{Synch Wait Time} + \text{Extra Instructions})} \]
  - Useful for early development, but unrealistic for real performance.
  - Ignores communication and also the imbalances it causes.
  - Can lead to poor choice of partitions as well as orchestration.
  - More recent models incorporate communication costs; BSP, LogP, ...
Summary of Tradeoffs

• Different goals often have conflicting demands
  – Load Balance:
    • Fine-grain tasks.
    • Random or dynamic assignment.
  – Communication:
    • Usually coarse grain tasks.
    • Decompose to obtain locality: not random/dynamic.
  – Extra Work:
    • Coarse grain tasks.
    • Simple assignment.
  – Communication Cost:
    • Big transfers: amortize overhead and latency.
    • Small transfers: reduce contention.
Generic Message-Passing Routines

• Send and receive message-passing procedure/system calls often have the form:

\[
\begin{align*}
\text{send}(\text{parameters}) \\
\text{recv}(\text{parameters})
\end{align*}
\]

– where the parameters identify the source and destination processes, and the data.

\[\text{send()} \quad \text{recv()}\]

Figure 2.2 Passing a message between processes using send() and recv() system calls
Blocking `send()` and `recv()` System Calls

(a) When `send()` occurs before `recv()`

(b) When `recv()` occurs before `send()`

Figure 2.3 Blocking `send()` and `recv()` system calls
Non-blocking send( ) and recv( ) System Calls

Figure 2.4 Non-blocking send() system call
Message-Passing Computing Examples

• Problems with a very large degree of parallelism:
  – Image Transformations:
    • Shifting, Rotation, Clipping etc.
  – Mandelbrot Set:
    • Sequential, static assignment, dynamic work pool assignment.

• Divide-and-conquer Problem Partitioning:
  – Parallel Bucket Sort.
  – Numerical Integration:
    • Trapezoidal method using static assignment.
    • Adaptive Quadrature using dynamic assignment.

• Pipelined Computation.
Synchronous Iteration

• Iteration-based computation is a powerful method for solving numerical (and some non-numerical) problems.

• For numerical problems, a calculation is repeated and each time, a result is obtained which is used on the next execution. The process is repeated until the desired results are obtained.

• Though iterative methods are sequential in nature, parallel implementation can be successfully employed when there are multiple independent instances of the iteration. In some cases this is part of the problem specification and sometimes one must rearrange the problem to obtain multiple independent instances.

• The term "synchronous iteration" is used to describe solving a problem by iteration where different tasks may be performing separate iterations but the iterations must be synchronized using point-to-point synchronization, barriers, or other synchronization mechanisms.
Barriers

A synchronization mechanism applicable to shared-memory as well as message-passing, where each process must wait until all members of a specific process group reach a specific reference point in their computation.

- Possible Implementations:
  - A library call possibly.
    implemented using a counter
  - Using individual point-to-point synchronization forming:
    - A tree.
    - Butterfly connection pattern.
Message-Passing Local Synchronization

Process $P_{i-1}$
recv($P_i$);
send($P_i$);

Process $P_i$
send($P_{i-1}$);
send($P_{i+1}$);
recv($P_{i+1}$);
recv($P_{i-1}$);
recv($P_i$);

Process $P_{i+1}$
recv($P_i$);
send($P_i$);
Network Characteristics

• Topology:
  – Physical interconnection structure of the network graph:
    • Node Degree.
    • Network diameter: Longest minimum routing distance between any two nodes in hops.
    • Average Distance between nodes.
    • Bisection width: Number of links whose removal disconnects the graph and cuts it in half.
    • Symmetry: The property that the network looks the same from every node.
    • Homogeneity: Whether all the nodes and links are identical or not.

  – Type of interconnection:
    • Static or Direct Interconnects: Nodes connected directly using static links point-to-point links.
    • Dynamic or Indirect Interconnects: Switches are usually used to realize dynamic links between nodes:
      – Each node is connected to specific subset of switches. (e.g. multistage interconnection networks MINs).
      – Blocking or non-blocking, permutations realized.
    • Shared-, broadcast-, or bus-based connections. (e.g. Ethernet-based).
Sample Static Network Topologies

- Linear
- Ring
- 2D Mesh
- Hypercube
- Binary Tree
- Fat Binary Tree
- Fully Connected
Static Connection
Networks Examples:
2D Mesh

For an $r \times r$ 2D Mesh:

- Node Degree: 4
- Network diameter: $2(r-1)$
- No of links: $2N - 2r$
- Bisection Width: $r$
- Where $r = \sqrt{N}$
Static Connection Networks Examples:

**Hypercubes**

- Also called binary $n$-cubes.
- Dimension $= n = \log_2 N$
- Number of nodes $= N = 2^n$
- Diameter: $O(\log_2 N)$ hops
- Good bisection BW: $N/2$
- Complexity:
  - Number of links: $N(\log_2 N)/2$
  - Node degree is $n = \log_2 N$
**Message Routing Functions Example**

**Network Topology:**
3-dimensional static-link hypercube
Nodes denoted by $C_2C_1C_0$

Routing by least significant bit $C_0$

Routing by middle bit $C_1$

Routing by most significant bit $C_2$
Embeddings In Two Dimensions

- Embed multiple logical dimension in one physical dimension using long interconnections.
Dynamic Connection Networks

• Switches are usually used to implement connection paths or virtual circuits between nodes instead of fixed point-to-point connections.

• Dynamic connections are established based on program demands.

• Such networks include:
  – Bus systems.
  – Multi-stage Networks (MINs):
    • Omega Network.
    • Baseline Network etc.
  – Crossbar switch networks.
Dynamic Networks Definitions

- **Permutation networks:** Can provide any one-to-one mapping between sources and destinations.
- **Strictly non-blocking:** Any attempt to create a valid connection succeeds. These include Clos networks and the crossbar.
- **Wide Sense non-blocking:** In these networks any connection succeeds if a careful routing algorithm is followed. The Benes network is the prime example of this class.
- **Rearrangeably non-blocking:** Any attempt to create a valid connection eventually succeeds, but some existing links may need to be rerouted to accommodate the new connection. Batcher's bitonic sorting network is one example.
- **Blocking:** Once certain connections are established it may be impossible to create other specific connections. The Banyan and Omega networks are examples of this class.
- **Single-Stage networks:** Crossbar switches are single-stage, strictly non-blocking, and can implement not only the N! permutations, but also the $N^N$ combinations of non-overlapping broadcast.
Permutations

• For $n$ objects there are $n!$ permutations by which the $n$ objects can be reordered.
• The set of all permutations form a permutation group with respect to a composition operation.
• One can use cycle notation to specify a permutation function.
  For Example:
  The permutation $\pi = (a, b, c)(d, e)$
  stands for the bijection mapping:
  $a \rightarrow b, \ b \rightarrow c, \ c \rightarrow a, \ d \rightarrow e, \ e \rightarrow d$
  in a circular fashion.
  The cycle $(a, b, c)$ has a period of 3 and the cycle $(d, e)$ has a period of 2. Combining the two cycles, the permutation $\pi$ has a cycle period of $2 \times 3 = 6$. If one applies the permutation $\pi$ six times, the identity mapping $I = (a) (b) (c) (d) (e)$ is obtained.
Perfect Shuffle

- Perfect shuffle is a special permutation function suggested by Harold Stone (1971) for parallel processing applications.
- Obtained by rotating the binary address of an one position left.
- The perfect shuffle and its inverse for 8 objects are shown here:
Multi-Stage Networks:
The Omega Network

- In the Omega network, perfect shuffle is used as an inter-stage connection pattern for all $\log_2 N$ stages.
- Routing is simply a matter of using the destination's address bits to set switches at each stage.
- The Omega network is a single-path network: There is just one path between an input and an output.
- It is equivalent to the Banyan, Staran Flip Network, Shuffle Exchange Network, and many others that have been proposed.
- The Omega can only implement $\frac{N^{N/2}}{N!}$ of the $N!$ permutations between inputs and outputs, so it is possible to have permutations that cannot be provided (i.e. paths that can be blocked).
  - For $N = 8$, there are $\frac{8^4}{8!} = \frac{4096}{40320} = 0.1016 = 10.16\%$ of the permutations that can be implemented.
- It can take $\log_2 N$ passes of reconfiguration to provide all links. Because there are $\log_2 N$ stages, the worst case time to provide all desired connections can be $(\log_2 N)^2$. 
Shared Memory Multiprocessors

• Symmetric Multiprocessors (SMPs):
  – Symmetric access to all of main memory from any processor.

• Currently Dominate the high-end server market:
  – Building blocks for larger systems; arriving to desktop.

• Attractive as high throughput servers and for parallel programs:
  – Fine-grain resource sharing.
  – Uniform access via loads/stores.
  – Automatic data movement and coherent replication in caches.

• Normal uniprocessor mechanisms used to access data (reads and writes).
  – Key is extension of memory hierarchy to support multiple processors.
Shared Memory Multiprocessors Variations

(a) Shared cache

(b) Bus-based shared memory

(c) Dancehall

(d) Distributed-memory
Caches And Cache Coherence In Shared Memory Multiprocessors

• Caches play a key role in all shared memory multiprocessor system variations:
  – Reduce average data access time.
  – Reduce bandwidth demands placed on shared interconnect.

• Private processor caches create a problem:
  – Copies of a variable can be present in multiple caches.
  – A write by one processor may not become visible to others:
    • Processors accessing stale value in their private caches.
  – Process migration.
  – I/O activity.
  – Cache coherence problem.
  – Software and/or software actions needed to ensure write visibility to all processors thus maintaining cache coherence.
Shared Memory Access Consistency Models

• Shared Memory Access Specification Issues:
  – Program/compiler expected shared memory behavior.
  – Specification coverage of all contingencies.
  – Adherence of processors and memory system to the expected behavior.

• **Consistency Models:** Specify the order by which shared memory access events of one process should be observed by other processes in the system.
  – Sequential Consistency Model.
  – Weak Consistency Models.

• **Program Order:** The order in which memory accesses appear in the execution of a single process without program reordering.

• **Event Ordering:** Used to declare whether a memory event is legal when several processes access a common set of memory locations.
Sequential Consistency (SC) Model

• Lamport’s Definition of SC:

[Hardware 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.

• Sufficient conditions to achieve SC in shared-memory access:
  – Every process issues memory operations in program order
  – After a write operation is issued, the issuing process waits for the write to complete before issuing its next operation.
  – After a read operation is issued, the issuing process waits for the read to complete, and for the write whose value is being returned by the read to complete, before issuing its next operation (provides write atomicity).

• According to these Sufficient, but not necessary, conditions:
  – Clearly, compilers should not reorder for SC, but they do!
    • Loop transformations, register allocation (eliminates!).
  – Even if issued in order, hardware may violate for better performance
    • Write buffers, out of order execution.
  – Reason: uniprocessors care only about dependences to same location
    • Makes the sufficient conditions very restrictive for performance.
Sequential Consistency (SC) Model

- As if there were no caches, and a only single memory exists.
- Total order achieved by *interleaving* accesses from different processes.
- Maintains *program order*, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others.
- Programmer’s intuition is maintained.
Further Interpretation of SC

- Each process’s program order imposes partial order on set of all operations.

- Interleaving of these partial orders defines a total order on all operations.

- Many total orders may be SC (SC does not define particular interleaving).

- *SC Execution*: An execution of a program is SC if the results it produces are the same as those produced by some possible total order (interleaving).

- *SC System*: A system is SC if any possible execution on that system is an SC execution.
Weak (Release) Consistency (WC)

- The DBS Model of WC: In a multiprocessor shared-memory system:
  - Accesses to global synchronizing variables are strongly ordered.
  - No access to a synchronizing variable is issues by a processor before all previous global data accesses have been globally performed.
  - No access to global data is issued by a processor before a previous access to a synchronizing variable has been globally performed.

⇒ Dependence conditions weaker than in SC because they are limited to synchronization variables.

⇒ Buffering is allowed in write buffers except for hardware-recognized synchronization variables.
**TSO Weak Consistency Model**

- Sun’s SPARK architecture WC model.

- Memory access order between processors determined by a hardware memory access “switch”.

- Stores and swaps issued by a processor are placed in a dedicated store FIFO buffer for the processor.
  
  ⇒ Order of memory operations is the same as processor issue order.

- A load by a processor first checks its store buffer if it contains a store to the same location.
  
  – If it does then the load returns the value of the most recent such store.
  
  – Otherwise the load goes directly to memory.
  
  – A processor is logically blocked from issuing further operations until the load returns a value.
Cache Coherence Using A Bus

- Built on top of two fundamentals of uniprocessor systems:
  - Bus transactions.
  - State transition diagram in cache.

- Uniprocessor bus transaction:
  - Three phases: arbitration, command/address, data transfer.
  - All devices observe addresses, one is responsible

- Uniprocessor cache states:
  - Effectively, every block is a finite state machine.
  - Write-through, write no-allocate has two states: valid, invalid.
  - Write-back caches have one more state: Modified (“dirty”).

- Multiprocessors extend both these two fundamentals somewhat to implement coherence.
Write-invalidate Snoopy Bus Protocol
For Write-Through Caches
State Transition Diagram

W(i) = Write to block by processor i
W(j) = Write to block copy in cache j by processor j ≠ i
R(i) = Read block by processor i.
R(j) = Read block copy in cache j by processor j ≠ i
Z(i) = Replace block in cache.
Z(j) = Replace block copy in cache j ≠ i
Write-invalidate Snoopy Bus Protocol
For Write-Back Caches
State Transition Diagram

RW: Read-Write
RO: Read Only
INV: Invalidated or not in cache

\[
\begin{align*}
R(i) & \quad W(i) \\
W(i) & \quad Z(j) \\
R(j) & \quad W(i) \\
W(j) & \quad Z(i) \\
Z(i) & \quad \text{R(i)} \\
R(j) & \quad Z(j) \\
W(j), Z(i) &
\end{align*}
\]

W(i) = Write to block by processor i
W(j) = Write to block copy in cache j by processor j ≠ i
R(i) = Read block by processor i.
R(j) = Read block copy in cache j by processor j ≠ i
Z(i) = Replace block in cache.
Z(j) = Replace block copy in cache j ≠ i
- BusRd(S) Means shared line asserted on BusRd transaction.
- Flush: If cache-to-cache sharing, only one cache flushes data.
Parallel System Performance: Evaluation & Scalability

- Factors affecting parallel system performance:
  - Algorithm-related, parallel program related, architecture/hardware-related.

- Workload-Driven Quantitative Architectural Evaluation:
  - Select applications or suite of benchmarks to evaluate architecture either on real or simulated machine.
  - From measured performance results compute performance metrics:
    - Speedup, System Efficiency, Redundancy, Utilization, Quality of Parallelism.
  - Application Models of Parallel Computer Models: How the speedup of an application is affected subject to specific constraints:
    - Fixed-load Model.
    - Fixed-time Model.
    - Fixed-Memory Model.

- Performance Scalability:
  - Definition.
  - Conditions of scalability.
  - Factors affecting scalability.
Parallel Performance Metrics Revisited

• **Degree of Parallelism (DOP):** For a given time period, reflects the number of processors in a specific parallel computer actually executing a particular parallel program.

• **Average Parallelism:**
  - Given maximum parallelism = \( m \)
  - \( n \) homogeneous processors.
  - Computing capacity of a single processor \( \Delta \)
  - Total amount of work (instructions, computations):

\[
W = \Delta \int_{t_1}^{t_2} DOP(t) \, dt \quad \text{or as a discrete summation} \quad W = \Delta \sum_{i=1}^{m} i \cdot t_i
\]

Where \( t_i \) is the total time that DOP = \( i \) and \( \sum_{i=1}^{m} t_i = t_2 - t_1 \)

The average parallelism \( A \):

\[
A = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} DOP(t) \, dt \quad \text{In discrete form} \quad A = \frac{\left( \sum_{i=1}^{m} i \cdot t_i \right)}{\left( \sum_{i=1}^{m} t_i \right)}
\]
Parallel Performance Metrics Revisited

Asymptotic Speedup:

Execution time with one processor:

\[ T(1) = \sum_{i=1}^{m} t_i(1) = \sum_{i=1}^{m} \frac{W_i}{\Delta} \]

Execution time with an infinite number of available processors:

\[ T(\infty) = \sum_{i=1}^{m} t_i(\infty) = \sum_{i=1}^{m} \frac{W_i}{i\Delta} \]

Asymptotic speedup \( S_\infty \)

\[ S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i}} \]
Harmonic Mean Performance

• Arithmetic mean execution time per instruction:

\[ T_a = \frac{1}{m} \sum_{i=1}^{m} T_i = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{R_i} \]

• The harmonic mean execution rate across \( m \) benchmark programs:

\[ R_h = \frac{1}{T_a} = \frac{m}{\sum_{i=1}^{m} \left(1/R_i \right)} \]

• Weighted harmonic mean execution rate with weight distribution \( \pi = \{f_i|i = 1, 2, \ldots, m\} \)

\[ R^*_h = \frac{1}{\sum_{i=1}^{m} (f_i/R_i)} \]

• Harmonic Mean Speedup for a program with \( n \) parallel execution modes:

\[ S = T_1/T^*_s = \frac{1}{(\sum_{i=1}^{n} f_i/R_i)} \]
Parallel Performance Metrics Revisited

Efficiency, Utilization, Redundancy, Quality of Parallelism

• System Efficiency: Let $O(n)$ be the total number of unit operations performed by an $n$-processor system and $T(n)$ be the execution time in unit time steps:
  
  – Speedup factor:
    
    $$ S(n) = \frac{T(1)}{T(n)} $$

  – System efficiency for an $n$-processor system:
    
    $$ E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)} $$

• Redundancy:

  $$ R(n) = \frac{O(n)}{O(1)} $$

• Utilization:

  $$ U(n) = R(n)E(n) = \frac{O(n)}{nT(n)} $$

• Quality of Parallelism:

  $$ Q(n) = \frac{S(n)E(n)}{R(n)} = \frac{T^3(1)}{nT^2(n)O(n)} $$
Parallel Performance Metrics Revisited: Amdahl’s Law

- **Harmonic Mean Speedup** (i number of processors used):

  \[ S = \frac{T_1}{T^*} = \frac{1}{\left(\sum_{i=1}^{n} f_i/R_i\right)} \]

- In the case \( w = \{f_i \text{ for } i = 1, 2, \ldots, n\} = (\alpha, 0, 0, \ldots, 1-\alpha) \), the system is running sequential code with probability \( \alpha \) and utilizing \( n \) processors with probability \( 1-\alpha \) with other processor modes not utilized.

  **Amdahl’s Law:**

  \[ S_n = \frac{n}{1 + (n - 1)\alpha} \]

  \( S \rightarrow 1/\alpha \) as \( n \rightarrow \infty \)

  \( \Rightarrow \) Under these conditions the best speedup is upper-bounded by \( 1/\alpha \)
The Isoefficiency Concept

- Workload \( w \) as a function of problem size \( s \) : \( w = w(s) \)
- \( h \) total communication/other overhead, as a function of problem size \( s \) and machine size \( n \), \( h = h(s,n) \)
- Efficiency of a parallel algorithm implemented on a given parallel computer can be defined as:
  \[
  E = \frac{W(s)}{W(s) + h(s,n)}
  \]
- Isoefficiency Function: \( E \) can be rewritten as:
  \[E = \frac{1}{1 + h(s, n)/w(s)}\]
  To maintain a constant \( E \), \( W(s) \) should grow in proportion to \( h(s,n) \) or,
  \[w(s) = \frac{E}{1 - E} \times h(s, n)\]

\( C = E/(1-E) \) is a constant for a fixed efficiency \( E \).

The isoefficiency function is defined as follows:
\[\int_E f_E(n) = C \times h(s, n)\]
If the workload \( w(s) \) grows as fast as \( f_E(n) \) then a constant efficiency can be maintained for the algorithm-architecture combination.
Speedup Performance Laws: Fixed-Workload Speedup

When  \( DOP = i > n \)  \((n = number \ of \ processors)\)

Execution time of \( W_i \)

\[
t_i(n) = \frac{W_i}{i\Delta} \left\lceil \frac{i}{n} \right\rceil
\]

Total execution time

\[
T(n) = \sum_{i=1}^{m} \frac{W_i}{i\Delta} \left\lceil \frac{i}{n} \right\rceil
\]

If  \( DOP = i < n \), then  \( t_i(n) = t_i(\infty) = \frac{W_i}{i\Delta} \)

Fixed-load speedup factor is defined as the ratio of  \( T(1) \) to  \( T(n) \):

\[
S_n = \frac{T(1)}{T(n)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i} \left\lceil \frac{i}{n} \right\rceil}
\]

Let  \( Q(n) \) be the total system overheads on an  \( n \)-processor system:

\[
S_n = \frac{T(1)}{T(n) + Q(n)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n)}
\]

The overhead delay  \( Q(n) \) is both application- and machine-dependent and difficult to obtain in closed form.
Amdahl’s Law for Fixed-Load Speedup

• For the special case where the system either operates in sequential mode (DOP = 1) or a perfect parallel mode (DOP = n), the Fixed-load speedup is simplified to:

\[ S_n = \frac{W_1 + W_n}{W_1 + W_n / n} \]

We assume here that the overhead factor \( Q(n) = 0 \)

For the normalized case where:

\[ W_1 + W_n = \alpha + (1 - \alpha) = 1 \] with \( \alpha = W_1 \) and \( 1 - \alpha = W_n \)

The equation is reduced to the previously seen form of Amdahl’s Law:

\[ S_n = \frac{n}{1 + (n - 1)\alpha} \]
Fixed-Time Speedup

- To run the largest problem size possible on a larger machine with about the same execution time.

Let $m'$ be the maximum DOP for the scaled up problem, $W'_{i}$ be the scaled workload with $DOP = i$

In general, $W'_{i} > W_{i}$ for $2 \leq i \leq m'$ and $W'_{1} = W_{1}$

Assuming that $T(1) = T'(n)$ we obtain:

$$\sum_{i=1}^{m} W_{i} = \sum_{i=1}^{m'} \frac{W'_{i}}{i} \left[ \frac{i}{n} \right] + Q(n)$$

Speedup $S'_{n} = T(1) / T'(n)$ is given by:

$$S'_{n} = \frac{T(1)}{T'(n)} = \frac{\sum_{i=1}^{m'} W'_{i}}{\sum_{i=1}^{m} \frac{W'_{i}}{i} \left[ \frac{i}{n} \right] + Q(n)} = \frac{\sum_{i=1}^{m'} W'_{i}}{\sum_{i=1}^{m} W_{i}}$$
Gustafson’s Fixed-Time Speedup

- For the special fixed-time speedup case where DOP can either be 1 or $n$ and assuming $Q(n) = 0$

\[
S'_n = \frac{T(1)}{T'(n)} = \frac{\sum_{i=1}^{m'} W'_i}{m} = \frac{W'_1 + W'_n}{W_1 + W_n} = \frac{W_1 + nW_n}{W_1 + W_n}
\]

Where $W'_n = nW_n$ and $W_1 + W_n = W'_1 + W'_n/n$

Assuming $a=W_1$ and $1-\alpha = W_n$ and $W_1 + W_n = 1$

\[
S'_n = \frac{T(1)}{T'(n)} = \frac{\alpha + n(1-\alpha)}{\alpha + (1-\alpha)} = n - \alpha(n-1)
\]
Fixed-Memory Speedup

- Let $M$ be the memory requirement of a given problem
- Let $W = g(M)$ or $M = g^{-1}(W)$ where:

$$W = \sum_{i=1}^{m} W_i \quad \text{workload for sequential execution} \quad W^* = \sum_{i=1}^{m} W^*_i \quad \text{scaled workload on } n \text{ nodes}$$

The memory bound for an active node is $g^{-1}\left(\sum_{i=1}^{m} W_i\right)$

The fixed-memory speedup is defined by:

Assuming $g^*(nM) = G(n)g(M) = G(n)W_n$ and either sequential or perfect parallelism and $Q(n) = 0$

$$S^*_n = \frac{T(1)}{T'(n)} = \frac{\sum_{i=1}^{m} W^*_i}{\sum_{i=1}^{m} \left[ W^*_i \left\lfloor \frac{i}{n} \right\rfloor \right] + Q(n)}$$

$$S^*_n = \frac{W^*_1 + W^*_n}{W^*_1 + W^*_n / n} = \frac{W^*_1 + G(n)W_n}{W^*_1 + G(n)W_n / n}$$
Scalability Metrics

• The study of scalability is concerned with determining the degree of matching between a computer architecture and an application algorithm and whether this degree of matching continues to hold as problem and machine sizes are scaled up.

• Basic scalability metrics affecting the scalability of the system for a given problem:

  Machine Size $n$  
  Problem Size $s$  
  I/O Demand $d$  
  Communication overhead $h(s, n)$, where $h(s, 1) = 0$  
  Computer Cost $c$  
  Programming Overhead $p$  
  Clock rate $f$  
  CPU time $T$  
  Memory Capacity $m$
Parallel Scalability Metrics

Scalability of An architecture/algorithm Combination

- CPU Time
- Machine Size
- Hardware Cost
- I/O Demand
- Memory Demand
- Programming Cost
- Problem Size
- Communication Overhead
Parallel System Scalability

- **Scalability** (informal restrictive definition):
  A system architecture is scalable if the system efficiency $E(s, n) = 1$ for all algorithms with any number of processors and any size problem $s$.

- **Scalability Definition** (more formal):
  The scalability $\Phi(s, n)$ of a machine for a given algorithm is defined as the ratio of the asymptotic speedup $S(s, n)$ on the real machine to the asymptotic speedup $S_I(s, n)$ on the ideal realization of an EREW PRAM

$$S_I(s, n) = \frac{T(s, 1)}{T_I(s, n)}$$

On the ideal realization of an EREW PRAM

$$\Phi(s, n) = \frac{S(s, n)}{S_I(s, n)} = \frac{T_I(s, n)}{T(s, n)}$$
MPPs Scalability Issues

• Problems:
  – Memory-access latency.
  – Interprocess communication complexity or synchronization overhead.
  – Multi-cache inconsistency.
  – Message-passing overheads.
  – Low processor utilization and poor system performance for very large system sizes.

• Possible Solutions:
  – Low-latency fast synchronization techniques.
  – Weaker memory consistency models.
  – Scalable cache coherence protocols.
  – To realize shared virtual memory.
  – Improved software portability; standard parallel and distributed operating system support.
Cost Scaling

- cost(p,m) = fixed cost + incremental cost (p,m)
- Bus Based SMP?
- Ratio of processors : memory : network : I/O ?
- Parallel efficiency(p) = Speedup(P) / P
- Costup(p) = Cost(p) / Cost(1)
- Cost-effective: speedup(p) > costup(p)
- Is super-linear speedup
Scalable Distributed Memory Machines

Goal: Parallel machines that can be scaled to hundreds or thousands of processors.

- Design Choices:
  - Custom-designed or commodity nodes?
  - Network scalability.
  - Capability of node-to-network interface (critical).
  - Supporting programming models?

- What does hardware scalability mean?
  - Avoids inherent design limits on resources.
  - Bandwidth increases with machine size $P$.
  - Latency should not increase with machine size $P$.
  - Cost should increase slowly with $P$. 
Generic Distributed Memory Organization

- Network bandwidth?
- Bandwidth demand?
  - Independent processes?
  - Communicating processes?
- Latency? $O(\log_2 P)$ increase?
- Cost scalability of system?

OS Supported?
Network protocols?

Communication Assist
Extend of functionality?

Global virtual
Shared address space?

Message transaction
DMA?

Node:
$O(10)$ Bus-based SMP

Custom-designed CPU?
Node/System integration level?
How far? Cray-on-a-Chip?
SMP-on-a-Chip?

Multi-stage interconnection network (MIN)? Custom-designed?

Scalable network

Switch

Cache coherence
Protocols.
Network Latency Scaling Example

\[ O(\log_2 n) \] Stage MIN using switches:

- **Max distance:** \( \log_2 n \)
- **Number of switches:** \( \alpha n \log n \)
- **overhead = 1 us, BW = 64 MB/s, 200 ns per hop**
- **Using pipelined or cut-through routing:**
  - \( T_{64}(128) = 1.0 \text{ us} + 2.0 \text{ us} + 6 \text{ hops} \times 0.2 \text{ us/hop} = 4.2 \text{ us} \)
  - \( T_{1024}(128) = 1.0 \text{ us} + 2.0 \text{ us} + 10 \text{ hops} \times 0.2 \text{ us/hop} = 5.0 \text{ us} \)

- **Only 20% increase in latency for 16x size increase**

- **Store and Forward**
  - \( T_{64}^{sf}(128) = 1.0 \text{ us} + 6 \text{ hops} \times (2.0 + 0.2) \text{ us/hop} = 14.2 \text{ us} \)
  - \( T_{64}^{sf}(1024) = 1.0 \text{ us} + 10 \text{ hops} \times (2.0 + 0.2) \text{ us/hop} = 23 \text{ us} \)
Physical Scaling

• Chip-level integration:
  – Integrate network interface, message router I/O links.
  – Memory/Bus controller/chip set.
  – IRAM-style Cray-on-a-Chip.
  – Future: SMP on a chip?

• Board-level:
  – Replicating standard microprocessor cores.
    • CM-5 replicated the core of a Sun SparkStation 1 workstation.
    • Cray T3D and T3E replicated the core of a DEC Alpha workstation.

• System level:
  • IBM SP-2 uses 8-16 almost complete RS6000 workstations placed in racks.
# Spectrum of Designs

None: Physical bit stream
- blind, physical DMA nCUBE, iPSC, ...

User/System
- User-level port CM-5, *T
- User-level handler J-Machine, Monsoon, ...

Remote virtual address
- Processing, translation Paragon, Meiko CS-2

Global physical address
- Proc + Memory controller RP3, BBN, T3D

Cache-to-cache
- Cache controller Dash, KSR, Flash
Scalable Cache Coherent Systems

- Scalable distributed shared memory machines Assumptions:
  - Processor-Cache-Memory nodes connected by scalable network.
  - Distributed shared physical address space.
  - Communication assist must interpret network transactions, forming shared address space.

- For a system with shared physical address space:
  - A cache miss must be satisfied transparently from local or remote memory depending on address.
  - By its normal operation, cache replicates data locally resulting in a potential cache coherence problem between local and remote copies of data.
  - A coherency solution must be in place for correct operation.

- Standard snoopy protocols studied earlier may not apply for lack of a bus or a broadcast medium to snoop on.

- For this type of system to be scalable, in addition to latency and bandwidth scalability, the cache coherence protocol or solution used must also scale as well.
**Scalable Cache Coherence**

- A scalable cache coherence approach may have similar cache line states and state transition diagrams as in bus-based coherence protocols.
- However, different additional mechanisms other than broadcasting must be devised to manage the coherence protocol.
- Two possible approaches:
  - Approach #1: Hierarchical Snooping.
  - Approach #2: Directory-based cache coherence.
  - Approach #3: A combination of the above two approaches.
Approach #1: Hierarchical Snooping

• Extend snooping approach: A hierarchy of broadcast media:
  – Tree of buses or rings (KSR-1).
  – Processors are in the bus- or ring-based multiprocessors at the leaves.
  – Parents and children connected by two-way snoopy interfaces:
    • Snoop both buses and propagate relevant transactions.
    • Main memory may be centralized at root or distributed among leaves.

• Issues (a) - (c) handled similarly to bus, but not full broadcast.
  – Faulting processor sends out “search” bus transaction on its bus.
  – Propagates up and down hierarchy based on snoop results.

• Problems:
  – High latency: multiple levels, and snoop/lookup at every level.
  – Bandwidth bottleneck at root.

• This approach has, for the most part, been abandoned.
Hierarchical Snoopy Cache Coherence

Simplest way: hierarchy of buses; snoopy coherence at each level.

– or rings.

• Consider buses. Two possibilities:
  
  (a) All main memory at the global (B2) bus.
  
  (b) Main memory distributed among the clusters.

(a) (b)
Scalable Approach #2: Directories

(a) Read miss to a block in dirty state

(b) Write miss to a block with two sharers

Many alternatives exist for organizing directory information.
Organizing Directories

Directory Schemes

- Centralized
- Distributed

How to find source of directory information

- Flat
- Hierarchical

How to locate copies

- Memory-based
- Cache-based

Let’s see how they work and their scaling characteristics with P
Flat, Memory-based Directory Schemes

- All info about copies co-located with block itself at home.
  - Works just like centralized scheme, except distributed.
- Scaling of performance characteristics:
  - Traffic on a write: proportional to number of sharers.
  - Latency a write: Can issue invalidations to sharers in parallel.
- Scaling of storage overhead:
  - Simplest representation: full bit vector, i.e. one presence bit per node.
  - Storage overhead doesn’t scale well with P; a 64-byte cache line implies:
    - 64 nodes: 12.7% overhead.
    - 256 nodes: 50% overhead.; 1024 nodes: 200% overhead.
  - For M memory blocks in memory, storage overhead is proportional to $P \times M$.
Flat, Cache-based Schemes

- How they work:
  - Home only holds pointer to rest of directory info.
  - Distributed linked list of copies, weaves through caches:
    - Cache tag has pointer, points to next cache with a copy.
  - On read, add yourself to head of the list (comm. needed).
  - On write, propagate chain of invalidations down the list.

- Utilized in Scalable Coherent Interface (SCI) IEEE Standard:
  - Uses a doubly-linked list.
Approach #3:
A Popular Middle Ground

• Two-level “hierarchy”.
• Individual nodes are multiprocessors, connected non-
hierarchically.
  – e.g. mesh of SMPs.
• Coherence across nodes is directory-based.
  – Directory keeps track of nodes, not individual processors.
• Coherence within nodes is snooping or directory.
  – Orthogonal, but needs a good interface of functionality.
• Examples:
  – Convex Exemplar: directory-directory.
  – Sequent, Data General, HAL: directory-snoopy.
Example Two-level Hierarchies

(a) Snooping-snooping

(b) Snooping-directory

(c) Directory-directory

(d) Directory-snooping
Advantages of Multiprocessor Nodes

- Potential for cost and performance advantages:
  - Amortization of node fixed costs over multiple processors:
    - Applies even if processors simply packaged together but not coherent.
  - Can use commodity SMPs.
  - Less nodes for directory to keep track of.
  - Much communication may be contained within node (cheaper).
  - Nodes prefetch data for each other (fewer “remote” misses).
  - Combining of requests (like hierarchical, only two-level).
  - Can even share caches (overlapping of working sets).
  - Benefits depend on sharing pattern (and mapping):
    - Good for widely read-shared: e.g. tree data in Barnes-Hut.
    - Good for nearest-neighbor, if properly mapped.
    - Not so good for all-to-all communication.
Disadvantages of Coherent MP Nodes

- Bandwidth shared among nodes.
- Bus increases latency to local memory.
- With local node coherence in place, a CPU typically must wait for local snoop results before sending remote requests.
- Snoopy bus at remote node increases delays there too, increasing latency and reducing bandwidth.
- Overall, may hurt performance if sharing patterns don’t comply with system architecture.