# Parallel and distributed computing

N. Kälin

March 8, 2021

# Contents

# 1 Architectures

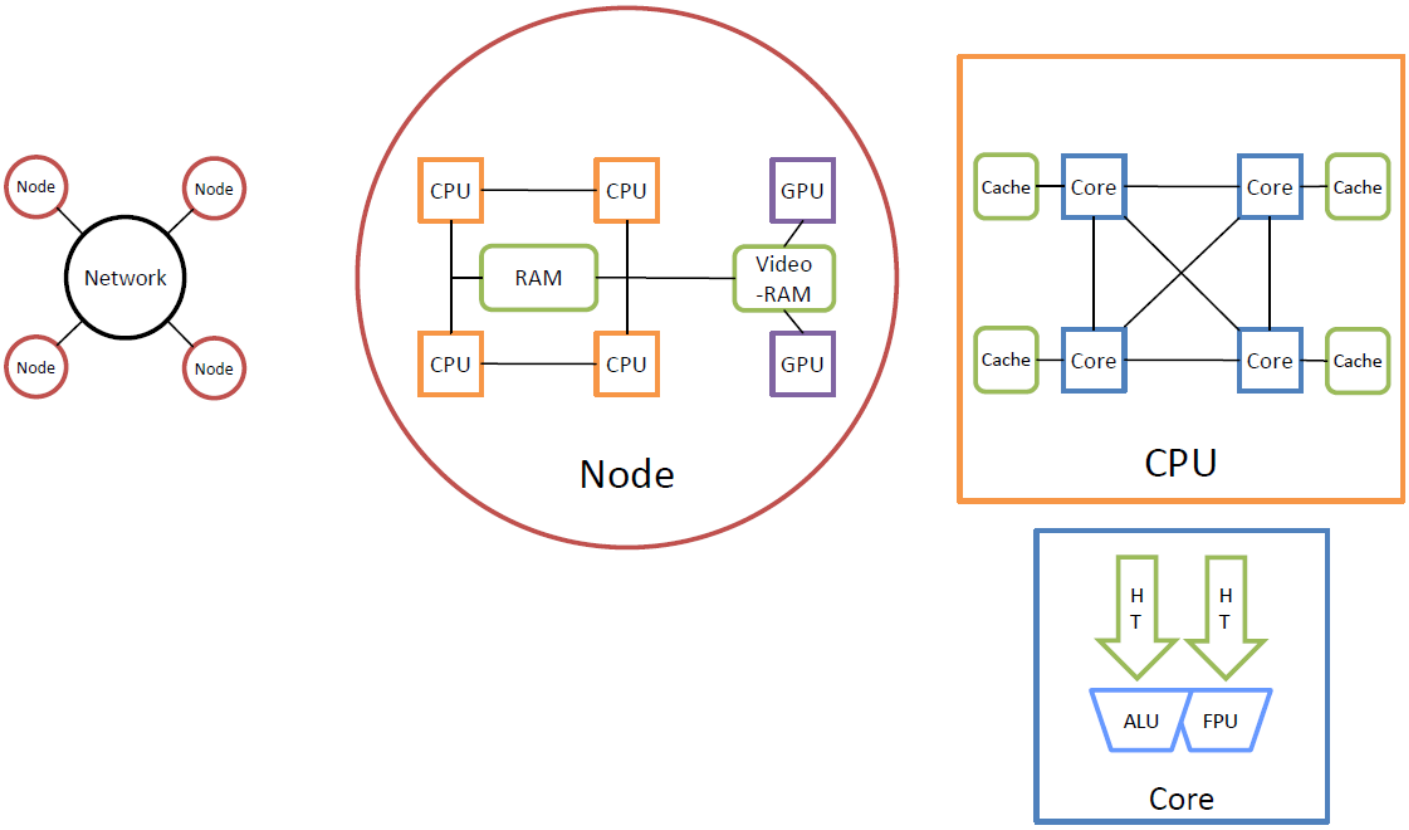



Figure 1: Parallel machine model (cluster)

## 1.1 Implicit vs. explicit parallelism

- Implicit Parallelism
  - processors have multiple functional units and execute multiple instructions in the same cycle
  - the precise way these instructions are selected and executed provides impressive diversity in Architectures
    - **pipelining**
    - **superscalar execution**
    - **very long instruction word processors**
- Explicit Parallelism
  - an explicitly parallel program must specify concurrency (**control structure**) and interaction (**communicatio model**) between concurrent subtasks

## 1.2 Parallel programming models

### 1.2.1 Overview of Programming models

- Programming models
  - provide support for expressing concurrency and synchronization
- Process based models
  - assume that all data associated with a process is private, by default, unless otherwise specified
- Lightweight processes and Threads

  - assume that all memory is global (bounded by process boundaries)
  - memory protection between threads of the same process is not necessary
  - support much faster memory access than processes with explicitly allocated shared memory
- Parallel programming language with syntax to specify parallelism
  - Examples: Ada, SR, Occam (no longer common)
- Directive based programming models
  - extend the threaded model by facilitating creation and synchronization of threads
  - Examples: Open MP, Linda, POP-C++

### 1.2.2 Parallel Machine Model

- PRAM
  - a natural extension of the Random Access Machine (RAM) serial architecture
  - consists of $p$ processors and a global memory of unbounded size that is uniformly accessible to all processors
  - procesors share a common clock but may execute different instructions in each cycle
- Handling of simultaneous memory accesses
  - Exclusive-read, exclusive-write (EREW)
  - Concurrecnt-read, exclusive-write (CREW)
  - Exclusive-read, concurrent-write (ERCW)
  - Concurrent-read, concurrent-write (CRCW)

What does concurrent write mean?

**Common:** write only if all values are identical.

**Arbitrary:** write the data from a randomly selected processor.

**Priority:** follow a predetermined priority order.

**Sum:** write the sum of all data items.

## 1.3 Different grains of parallelism

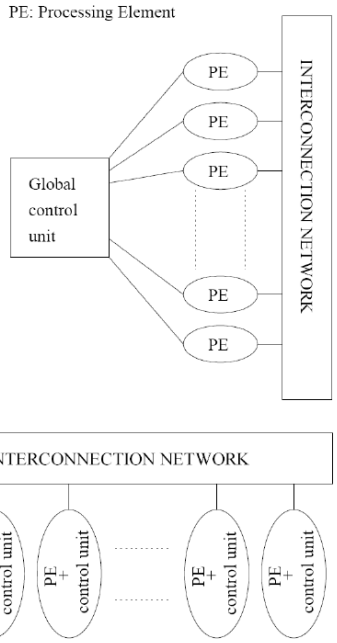
- Granularity: the ratio of computation to communication
  - periods of computation are separated from periods of communication by synchronization events
  - constrained by the inherent characteristics of the used algorithms
  - the parallel programmer must select the right granularity to benefit from the underlying platform
- Chunking
  - determining the amount of data to assign to each task (chunk or grain size)
- Which Granularity will lead to best performance?
  - depends on the algorithm and the used hardware environment
  - general rule: increase grain size if the communication overhead is too large

### 1.3.1 Trade-offs associated with chunk size

- Fine-grained parallelism
  - low arithmetic intensity
  - may not have enough work to hide long-duration asynchronous communication
  - facilitates load balancing by providing a larger number of more manageable (i.e. smaller) work units
  - too fine granularity can produce slower parallel implementation than the serial execution (too much overhead required for communication)
- Coarse-grained parallelism
  - high arithmetic intensity
  - complete applications can serve as the grain of parallelism
  - more difficult to load balance efficiently

## 1.4 Control structure of parallel platforms

- SIMD: Single Instruction stream, Multiple Data stream
  - there is a single control unit that dispatches the same instruction to various processors (that work on different data)
  - data parallelization
- MIMD: Multiple Instruction stream, Multiple Data stream
  - each processor has its own control unit
  - each processor can execute different instructions on different data items




### 1.4.1 SIMD Computers

- Hardware requirements
  - SIMD computers require less HW than MIMD computers (only one control unit)
  - SIMD computers require less memory (only one copy of the program is stored)
- Current implementations
  - Graphics Processing Units (GPUs)
  - Digital Signal Processors (DSPs) are widely used in cameras and sound equipments
  - Co-processing units in Intel CPUs: SSEx, AVX-512
- Software requirements
  - SIMD relies on the regular structure of computations (such as those in image and video processing or in deep learning)
  - it is often necessary to selectively turn off operations on certain data items

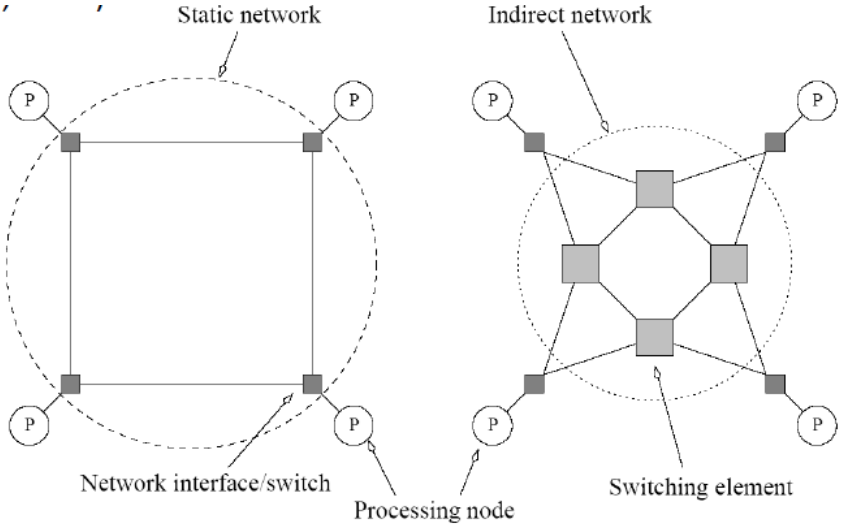
### 1.4.2 MIMD Computers

- Single Program Multiple Data (SPMD)
  - a simple variant of MIMD executes the same program on different processors
  - SPMD and MIMD are closely related in terms of programming flexibility and underlying architectural support
  - a single program consisting of several programs in a large switch block with conditions specified by the task identifiers is equivalent to the MIMD model
- Current MIMD implementations
  - SPARC servers, multiprocessor PCs, NASA Beowulf inspired workstation clusters
- Key advantages of workstation clusters
  - high performance workstations and PCs available at low cost
  - latest processors can easily be incorporated into the system as they become available
  - existing software can be used or modified

## 1.5 Communication models of parallel platforms

- Shared-Address-Space Platforms (Multiprocessors)
  - part (or all) of the memory is accessible to all processors
  - processors interact by modifying data objects stored in this shared-address-space
  - uniform or non-uniform memory access time (UMA vs. NUMA)
- Message Passing Platforms (Multicomputers)
  - comprise of a set of processors and their own (exclusive) memory
  - instances come naturally from clustered workstations (distributed systems) and non-shared-address-space multi-computers
  - are programmed using sending messages (variants of send and receive primitives)
  - libraries such as MPI (1990's) provide such primitives

## 1.6 Interconnection networks




- Interconnection Networks for Parallel Computers
  - carry data between processors and to memory
  - are made of switches and links (wires, fiber)
  - are classified as static or dynamic
    - static (direct) networks consist of point-to-point communication links among processing nodes
    - dynamic (indirect) networks are built using switches and communication links
- Network Topologies
  - a variety of network topologies have been proposed and implemented
  - tradeoff performance for cost
  - two basic categories: physical and logical topologies
  - commercial machines often implement hybrids of multiple topologies

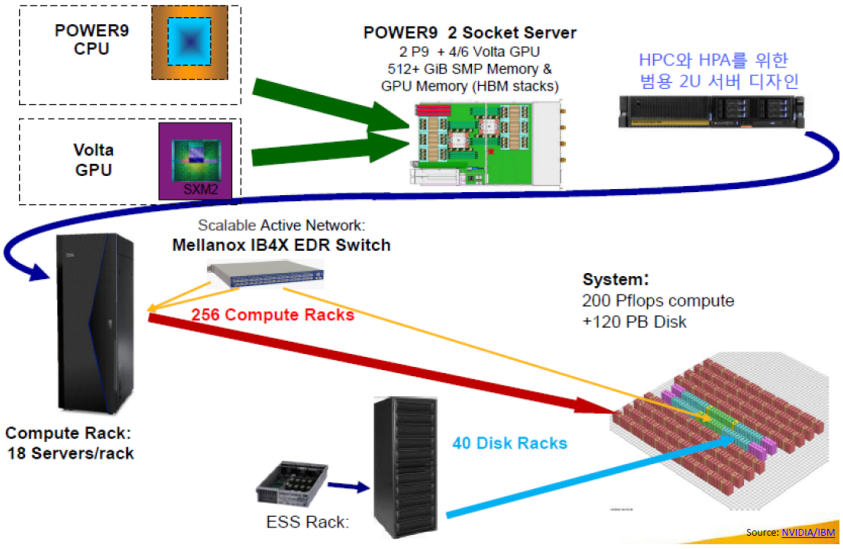
### 1.6.1 Interconnection Network for HPC

- Infiniband
  - a computer-networking communications standard used in HPC that features very high throughput and very low latency
  - it is used for data interconnect both among and within computers
  - it is also utilized as either a direct, or switched interconnect between servers and storage systems, as well as an interonnect between storage systems
  - it is designed to be scalable and uses a switched fabric network topology

| Year | FDR 2011 | EDR 2014 | HDR 2017 | NDR 2020 | XDR 2023 |
|---|---|---|---|---|---|
| Throughput, per 1x [Gbit/s] | 13.64 | 25 | 50 | 100 | 250 |
| Speed for 4x links [Gbit/s] | 54.54 | 100 | 200 | 400 | 1000 |
| Speed for 12x links [Gbit/s] | 163.64 | 300 | 600 | 1200 | 3000 |
| Latency [$\mu s$] | 0.7 | 0.5 | 0.5 | tbd | tbd |

- PCI Express Version 4
  - a high-speed serial computer expansion bus standard
  - has numerous improvements over the older standards
    - higher maximum system bus throughput
    - lower I/O pin count and smaller physical footprint
  - has been drafted with final specifications expected in 2017
  - throughput:
    - x1: 1.969 GByte/s
    - x16: 31.508 GByte/s
  - external cabling: Thunderbolt
- NVIDIA NVLink
  - is a high-bandwidth, energy-efficient interconnect
  - enables ultra-fast communication between the CPU and GPU, and between GPU
  - throughput:
    - version 1 (used in NVIDIA Pascal): x1: 20 GByte/s, x4: 80 GByte/s
    - version 2 (used in IBM Power9 chip, NVIDIA Volta GPUs): x1: 25 GByte/s, x8: 200 GByte/s

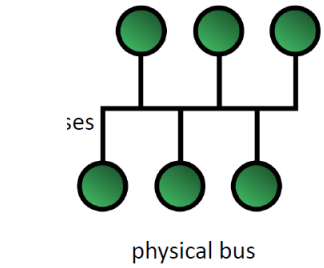
### 1.6.2 Data-Centric IT Environments

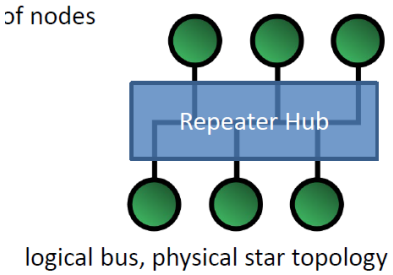


## 1.7 Network topologies

### 1.7.1 Network Topologies: Bus

- Principle and Properties
  - some of the simplest and earliest parallel machines used buses
  - all processors access a common bus for exchanging data
  - the distance between any two nodes is $O(1)$ in a bus
- Bottleneck
  - the bandwidth of the shared bus is a major bottleneck
  - typical bus-based machines are limited to dozens of nodes
- Examples
  - WLAN zone (logical bus topology)
  - PCI bus (physical bus topology)

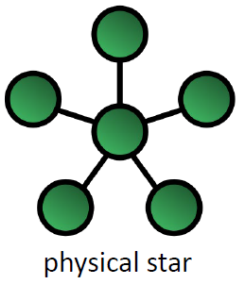


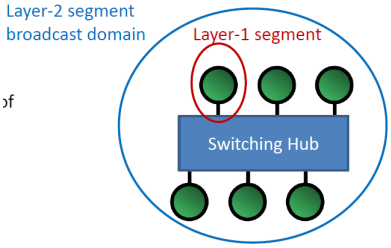

logical bus, physical star topology

### 1.7.2 Network Topologies: Star

- Principle and Properties
  - every node is connected only to a common node at the center
  - distance between any pair of node is $O(1)$
- Bottleneck
  - the central node
- Example
  - today's Ethernet based LANs with bridging hub (Bridge) or switching hub (Switch) as the center of the start topology



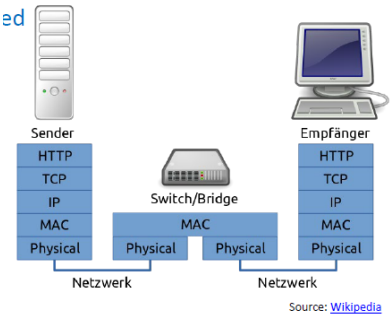
physical star




logical and physical star topology

### Network Infrastructure: Switchung Hub

- Principle and Properties
  - frame forwarding depends on learned physical device-addresses (MAC) per port (Layer-2 switching)
  - non-blocking: several input-output connections can be used in parallel without blocking
  - store-and-forward
    - the switch buffers and verifies each frame before forwarding it
    - a frame is received in its entirety before it is forwarded
    - error checking can be done before forwarding
  - cut-through
    - the switch starts forwarding after the frame's destination address is received
    - when the outgoing port is busy at the time, the switch falls back to store-and-forward operation
    - there is no error checking with this method

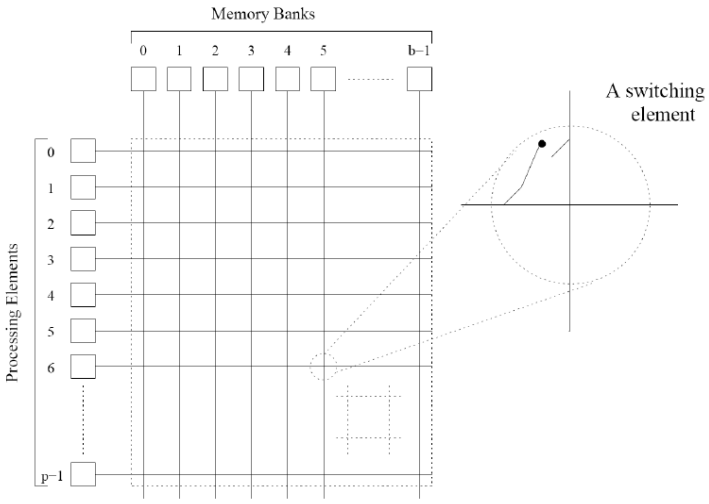


**Switching Hub: Advanced Features**

- Spanning Tree Protocol → Shortest Path Bridging
  - classic bridges may also interconnect using a spanning tree protocol that disables links so that the resulting local area network is a tree without loops
  - in contrast to routers, spanning tree bridges must have topologies with only one active path between two points
  - IEEE 802.1aq allows all paths to be active with multiple equal cost paths
    - provides much larger layer 2 topologies (up to 16 million compared to the 4096 VLANs limit)
    - improves the use of the **mesh topologies** through increased bandwidth and redundancy between all devices by allowing traffic to load share across all paths of a mesh network
- IEEE 802
  - is a family of IEEE standards dealing with local area networks and metropolitan area networkds
  - services and protocols specified in IEEE 802 map to the lower two layers (Data Link and Physical) of the sevel-layer OSI networking reference model
  - small subset of the working groups
    - 802.1: higher layer LAN protocols (bridging)
    - 802.1D: Spanning Tree Protocol (forwarding stopped while the spanning tree re-converged)
    - 802.1s: Multiple Spanning Tree Protocol
    - 802.1w: Rapid Spanning Tree Protocol
    - 802.1aq: Shortest Path Bridging (SPB) (incorporate all the older spanning tree protocols)

### 1.7.3 Network Topologies: Crossbar

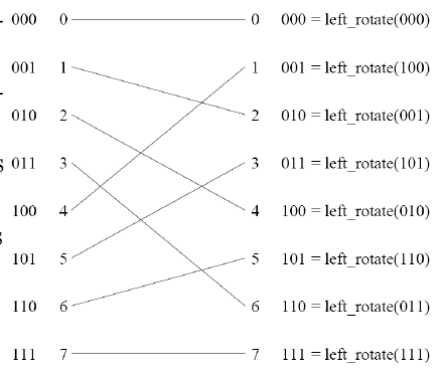
- Principle and Properties
  - a crossbar network uses an $p \cdot b$ grid of switches to connect $p$ inputs to $b$ outputs in a non-blocking manner
- Bottleneck
  - the cost of a crossbar of $p$ processors grows as $O(p^2)$ → difficult to scale for large values of $p$
- Usage
  - in non-blocking switches
  - between L2- and L2-caches



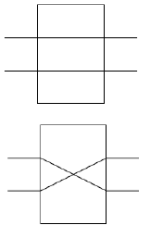

### 1.7.4 Network Topologies: Multistage Network

- Scalability
  - busses have excellent cost scalability, but poor performance scalability
  - crossbars have excellent performance scalability but poor cost scalability
  - multistage interconnects strike a compromise between these extremes
- Example: Omega Network
  - it consists of $log(p)$ stages, where $p$ is the number of inputs/outputs
  - at each stage, input $i$ is connected to output $j$ if:

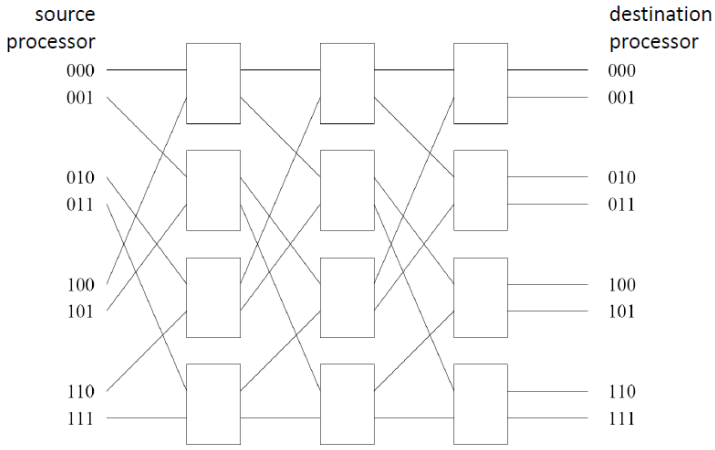
$$j = \begin{cases} 2i, & 0 \leq i \leq p/2 - 1 \\ 2i + 1 - p, & p/2 \leq i \leq p - 1 \end{cases}$$



### Omega Network

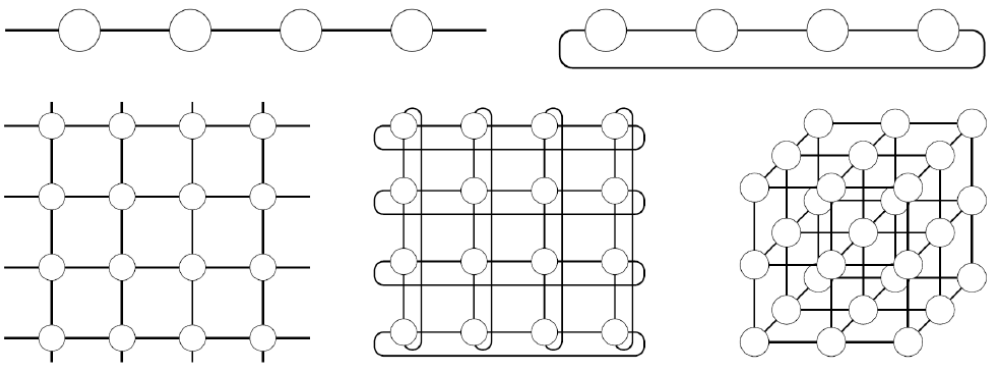


- Principle and Properties
  - the perfect shuffle patterns are connected using 2x2 switches
  - the switches operate in two modes: pass-through or cross-over



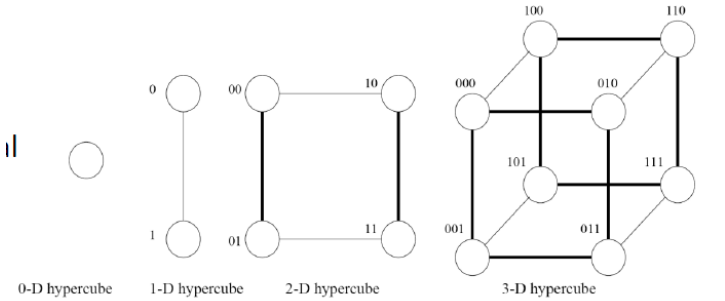

### Linear Array, Mesh, and $k-d$ Mesh

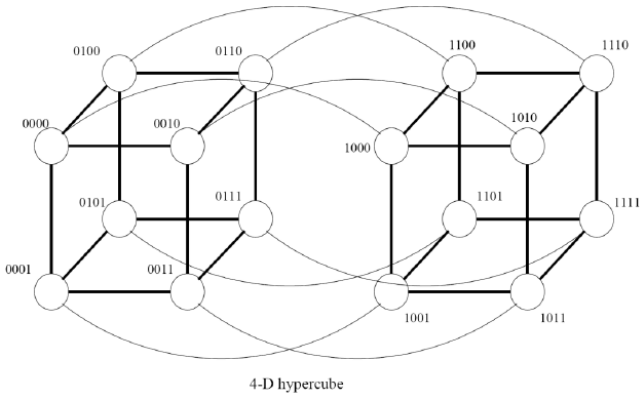
- Principle and Properties
  - in a linear array, each node has two neighbors, one to its left and one to its right
  - if the nodes at either end are connected, we refer to it as a 1-D torus or a ring
  - a generalization to 2 dimensions has nodes with 4 neighbors, to the north, south, east, and west (toroidal mesh)
  - a further generalization to $d$ dimensions has nodes with $2d$ neighbors



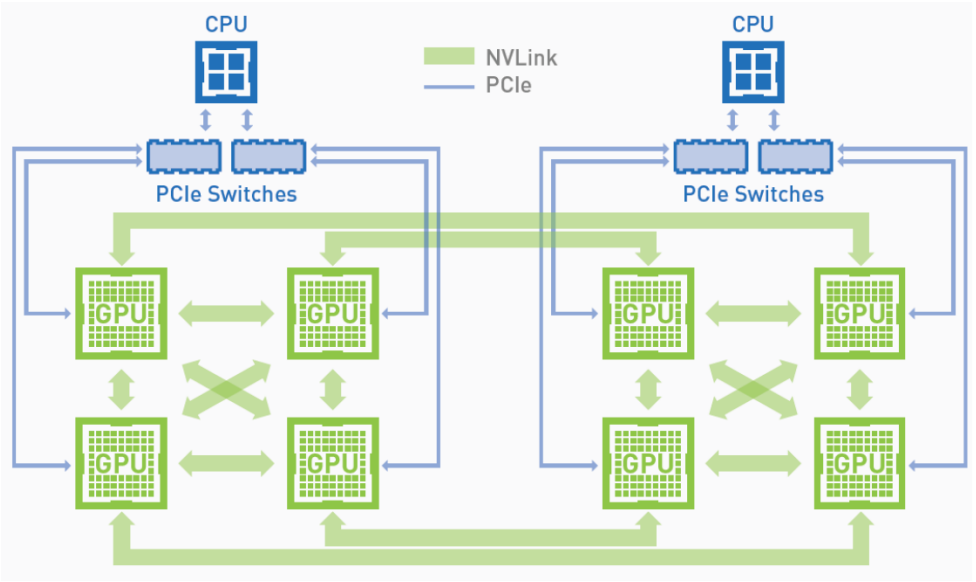
### 1.7.5 Network Topologies: Hypercube

- Principle and Properties
  - a special case of a $d$-dimensional mesh is a hypercube
  - $d = log(p)$, where $p$ is the total number of nodes
  - the distance between any two nodes is at most $log(p)$
  - each node has $log(p)$ neighbors
  - the distance between two nodes is given by the number of bit positions at which the two nodes differ



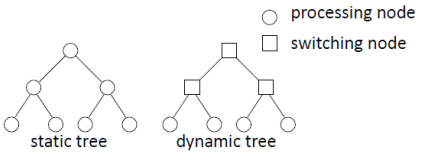




### NVIDIA NVLink: Hypercube Mesh Hybrid

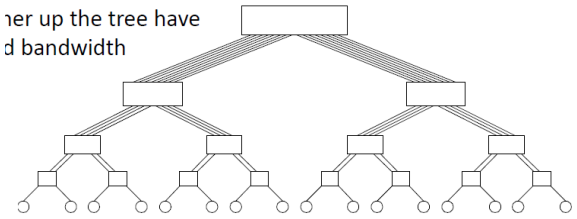



### 1.7.6 Tree-Based Networks

- Principle and Properties
  - one path between any pair of nodes
    - linear arrays and star-connected networks are special cases of tree networks
  - the distance between any two nodes is no more than $2log(p)$
  - links higher up the tree potentially carry more traffic than those at the lower levels
  - trees can be laid out in 2D with no wire crossings
- Fat-Tree



– links higher-up the tree have increased bandwidth




### 1.7.7 Evaluating Interconnection Networks

- Diameter
  - the distance between the farthest two nodes in the network
- Channel Bandwidth = channel width x channel rate
  - channel width: number of bits that can be communicated simultaneously over a link
  - channel rate: peak data transfer rate per link
- Cross-Section Bandwidth = bisection width x channel bandwidth
  - bisection width: the minimum number of wires one must cut to divide the network into two equal parts
- Cost
  - the number of links or switches (whichever is asymptotically higher) is a meaningful measure of the cost
  - the ability to layout the network
  - the length of wires
  - ...

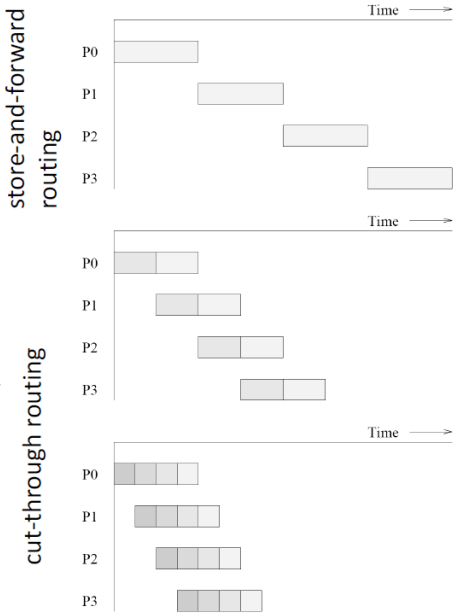
| Network | Diameter | Bisection width | Arc connectivity | Cost (No. of links) |
|---|---|---|---|---|
| Completely-connected | 1 | $p^2/4$ | $p-1$ | $p(p-1)/2$ |
| Star | 2 | * | 1 | $p-1$ |
| Complete binary tree | $2 \cdot log((p+1)/2)$ | 1 | 1 | $p-1$ |
| Linear array | $p-1$ | 1 | 1 | $p-1$ |
| 2D Mesh, no wraparound | $2(\sqrt{p}-1)$ | $\sqrt{p}$ | 2 | $2(p-\sqrt{p})$ |
| 2D wraparound Mesh | $2\lfloor\sqrt{p}/2\rfloor$ | $2\sqrt{p}$ | 4 | $2p$ |
| Hypercube | $log(p)$ | $p/2$ | $log(p)$ | $(p \cdot log(p))/2$ |
| Wraparound $k$-ary $d$-cube | $d\lfloor k/2\rfloor$ | $2k^{d-1}$ | $2d$ | $dp$ |

* depends on the node (switch) in the center, e.g. Crossbar or Omega Network

| Network | Diameter | Bisection width | Arc connectivity | Cost (No. of links) |
|---|---|---|---|---|
| Crossbar | 1 | $p$ | 1 | $p^2$ |
| Omega Network | $log(p)$ | $p/2$ | 2 | $p/2$ |
| Dynamic Tree | $2 \cdot log(p)$ | 1 | 2 | $p-1$ |

## 1.8 Communication costs in parallel systems

- Overhead in parallel programs
  - idling
  - contention
  - communication
- Communication costs depend on
  - communication model
  - the network topology
  - data handling and routing (e.g. packet routing, cut-through routing)
  - associated software protocols
  - ...




### 1.8.1 Message Passing Costs

- Total time to transfer a message over a network comprises of the following:
  - *Startup time* ($t_s$): Time spent at sending and receiving nodes (executing the routing algorithm, programming routers, etc.)
  - *Per-hop time* ($t_h$): This time is a function of number of hops and includes factors such as switch latencies, network delays, etc.
  - *Per-word transfer time* ($t_w$): This time includes all overheads that are determined by the length of the message. This includes bandwidth of links, error checking and correction, etc.

### 1.8.2 Cost Model for Communicating Messages

- Communication Costs
  - the cost of communicating a message between two nodes/hops away using cut-through routing is given by
$$t_{\text{comm}} = t_s + l \cdot t_h + m \cdot t_w$$
  - $t_h$ is typically smaller than $t_s$ and $t_w$, so the second term does not show, when $m$ is large
  - furthermore, it is often not possible to control routing and placement of tasks
- Simplified Cost Model
$$t_{\text{comm}} = t_s + m \cdot t_w$$
- Remarks
  - it is important to note that the original expression for communication time is valid for only uncongested networks
  - if a link takes multiple messages, the corresponding $t_w$ term must be scaled up by the number of messages
  - different communication patterns congest different networks to varying extents

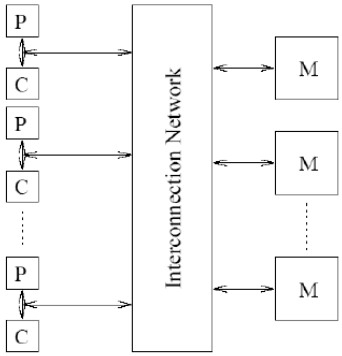
### 1.8.3 Cost Model for Shared Memory Systems

- Simplified Cost Model (still practical, but accurate cost modeling is more difficult)
  - memory layout is typically determined by the system
  - finite cache sizes can result in cache thrashing

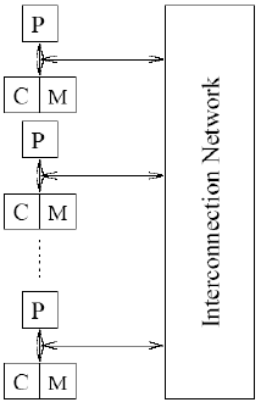
- overheads associated with invalidate and update operations are difficult to quantify
- spatial locality is difficult to model
- pre-fetching can play a role in reducing the overhead associated with data access
- false sharing and contention are difficult to model

# 2 Shared Memory Systems

## 2.1 Shared-address-space platforms



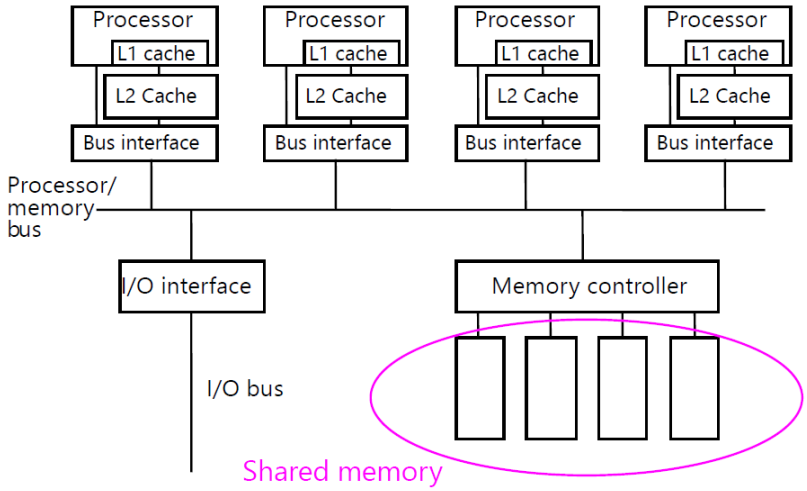

UMA (Uniform-memory-access)
shared-address-space computer with local caches and global memories → all memory access times (except cache) are identical



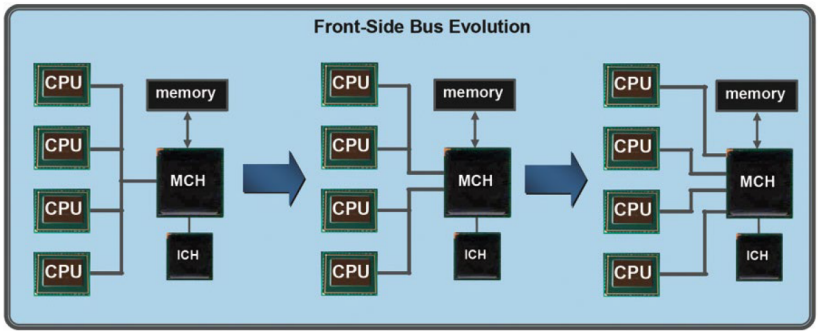

NUMA (Non-uniform-memory-access)
shared-address-space computer with local memory only
→ local memory access times are shorter

### 2.1.1 UMA Examples

- Intel Front Side Bus Architecture



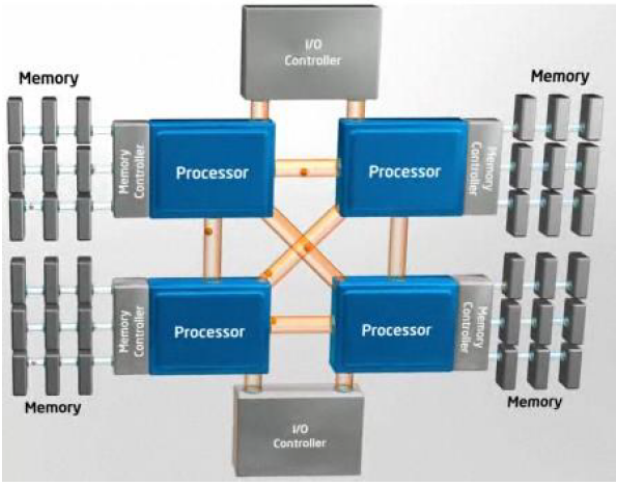

- Intel Pentium Front Side Bus Evolution




Source: Hardware LUXX

### 2.1.2 NUMA Examples

- Example: Intel Core i7



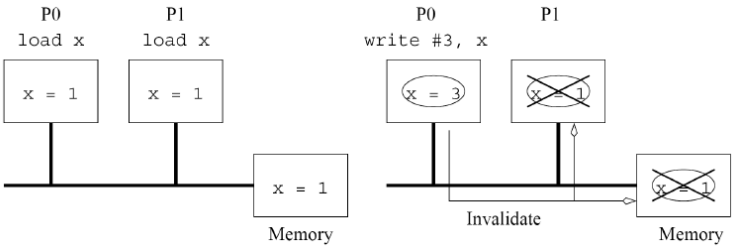

Source: University of Portsmouth

## 2.2 Cache coherence

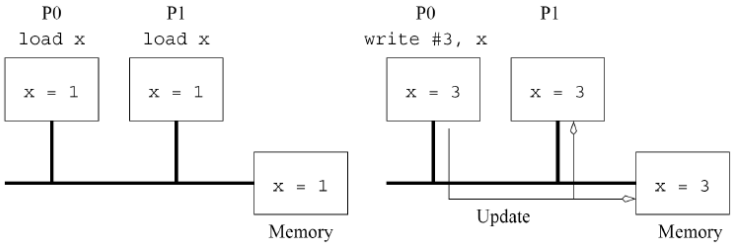
- Cache
  - principle of space and time locality
  - faster memory access
  - additional hardware is required to keep multiple copies of data consistent with each other
- Cache Coherence
  - ensuring that concurrent operations on multiple copies of the same memory word have well-defined semantics
  - this semantic is generally one of serializability
    - there exists some serial order of instruction execution that corresponds to the parallel schedule

### 2.2.1 Update and Invalidate Protocols

- Scenario
  - when a processor changes the value of its copy of a variable, one of two things must happen:
    - the other copies must be invalidated (invalidate protocol) (write-back)




    - the other copies must be updated (update protocol) (write-through)




- Pros and Cons
  - if a processor just reads a value once and does not need it again, an update protocol may generate significant overhead
  - if two processors make interleaved test and updates to a variable, an update protocol is better
  - both protocols suffer from false sharing overheads (two words that are not shared, however, they lie on the same cache line)
- Most current machines use invalidate protocols
  - each copy of a data item is associated with a state (e.g. shared, invalid, or dirty)
    - in shared state, there are multiple valid copies of the data item (therefore, an invalidate would have to be generated on an update)
    - in dirty state, only one copy exists and therefore, no invalidates need to be generated
    - in invalid state, the data copy is invalid, therefore, a read generates a data request (and associated state changes)

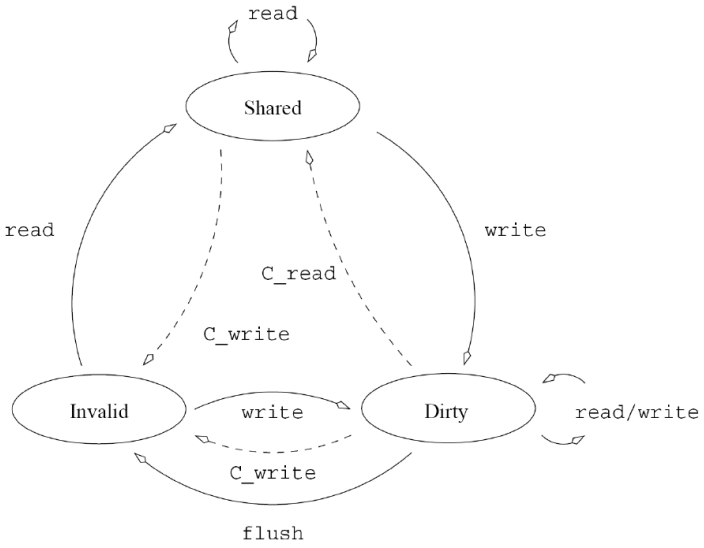



Figure 2: State Diagram of an Invalidate Protocol

## 2.3 Parallel programming in modern C++

- Explicit Threading
  - data exchange is more apparent
    - this helps in alleviating some of the overheads from data movement, false sharing, and contention
  - provides richer API in the form of condition waits, locks of different types, and increased flexibility for building composite synchronization operations
  - tools and support are easier to find
- Directives Layered on Top of Threads
  - simplify a variety of thread-related tasks
  - a programmer is rid of the tasks of initializing attributes objects, setting up arguments to threads, partitioning iteration spaces, etc.

### 2.3.1 Thread

- A thread is a single stream of control in the flow of a program. A program like

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

- can be transformed to

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

- In this case, one may think of the thread as an instance of a function that returns before the function has finished executing.

## 2.4 Threads, Futures, Tasks

### 2.4.1 C++: Threads and futures

- Thread
  - low-level
  - exchange of data must be synchronized itself
  - uncaught exceptions in the thread function lead to the termination of the entire program
  - thread_local storage class: static/global variables are created for each thread
  - is started automatically in the constructor
- Future
  - asynchronous processing: parallel execution or when calling get()
  - exceptions appear in the parent thread when the result is picked up with get()
  - when leaving the scope of the responsible future, then its destructor ensures a clear and problem-free termination of the asynchronous computation

### 2.4.2 C++: Threads as the Basis of Parallelism

- Constructor and Executor
  - thread(executable object, parameters of the executable object)
  - Executable object
    - function object (functor)
    - lambda expression
    - pointer to a function
  - the executable object and the parameters are copied by default, so the thread can work on their own data
- Example

```
void printFibs(size_t from,
    size_t to);

struct Image {
    void fill(int r, int g, int
        b);
};
```

```
int main() {
    thread th1(printFibs, 28,
        35);
    Image img;
    thread th2([&img] { img.
        fill(0,1,2); });
    th1.join(); th2.join();
}
```

### 2.4.3 C++: Futures and async

- Asynchronous Computation
  - parallel or deferred processing
  - async() initiates a computation and returns immediately
  - get() blocks until the result of the computation is available
- Return value of async()
  - future<RT>, where RT is the return type of the asynchronously executed function
- Launch Policy
  - async(launch::async, longComputation) guarantees parallel execution (default)
  - async(launch::deferred, computation) executed when calling get()
- Behind the scenes
  - a future can be produced without calling async(), by first creating a promise (some kind of transmission channel)

### C++: Futures Example

```
#include <future>

static size_t fibrec(size_t n) {
    return (n < 2) ? 1 : fibrec(n - 2) + fibrec(n - 1);
}

int main() {
    // asynchronous computation
    auto fut1 = async(launch::async, &fibrec, 35);
    // deferred computation
    auto fut2 = async(launch::deferred, &fibrec, 35);
    cout << fut2.get() << endl;     // waiting for the result of fut2
    cout << fut1.get() << endl;     // waiting for the result of fut1
}
```

What happens if the order of the two calls of get() is changed? → fut2 would only start after fut1 would have finished → serial execution

### 2.4.4 C++: Packaged Tasks

- Concept
  - a packaged_task wraps a callable element and allows its result to be retrieved asynchronously
  - it is similar to function, but transferring its result automatically to a future object
- Syntax
  - template<class Ret, class ... Args> class packaged_task<Ret(Args...)>;
- Object contains internally two elements

  - a stored task, which is some callable object whose call signature shall take arguments of the types in Args... and return a value of type Ret
  - a shared state, which is able to store the results of calling the stored task (of type Ret) and be accessed asynchronously through a future

#### C++: Packaged Tasks Example

```
// create task for calling fibrec
// argument of fibrec has to be defined later
packaged_task<size_t(size_t)> task1(&fibrec);
auto f1 = task1.get_future(); // future for getting result

// create task for calling fibrec
// argument of fibrec is bound to 35
packaged_task<size_t(void)> task2(bind(&fibrec, 35));
auto f2 = task2.get_future(); // future for getting result

// call task1 in a parallel thread (move semantic)
thread th(move(task1), 35);
// call task2 in this thread
task2();

// get results
cout << f.get() << endl;
cout << f2.get() << endl;

th.join(); // this thread waits on parallel thread th
```

### 2.4.5 C++: Synchronization Primitives

- Synchronized data access (read and write) is necessary if at least one of the parallel threads changes common data
- Synchronization Primitives
  - atomic_xyz all accesses are atomic (are not interrupted)
  - atomic_flag atomic bool but lock-free
  - once_flag used in call_once, makes sure that only one of the parallel threads executes the function
  - mutex realizes mutual exclusion
  - recursive_mutex allows a thread computing a recursive function to reenter a critical section
  - lock_guard locks a critical section; very simple usage; the only state is locked
  - unique_lock needs its unique mutex object; handles both states: locked and unlocked
  - condition_variable blocks this thread until signaled (signals or notifications can be sent by other threads)

### 2.4.6 Serial for- vs. Parallel for-Loop

- Notice
  - a lot of programming languages don't have a special parallel for-loop syntax
  - they use the keyword for in two situations
- Serial for-Loop
  - the loop notation simplifies the programming of a fixed number of repetitive, sequentially ordered steps

        * Example: reading $n$ integers from a sequential file
- Parallel for-Loop
  - the loop notation simplifies the programming of a fixed number of repetitive steps, that can be done in any order
    * Example: for each element of an array proceed the same task

**C++: Possible parallel for-each implementation**

```
template<typename It>
void parallelForEach(It begin, It end, function<void(typename It::reference)> f) {
    const ptrdiff_t len = end - begin;
    if (len == 0) {
        return;
    } else if (len == 1) {
        f(*begin);
        return;
    }

    const It mid = begin + (ptrdiff_t)(len/2);
    future<void> fut = async(parallelForEach<It>, begin, mid, f);
    try {
        parallelForEach(mid, end, f);
    } catch (...) {
        fut.wait();
        throw;
    }
    fut.get();
}
```

**C++: Usage of parallelForEach**

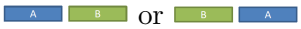
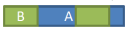
```
static size_t fibrec(size_t n) {
    return (n < 2) ? 1 : fibrec(n - 2) + fibrec(n - 1);
}

int main() {
    int vals[] = {5, 10, 15, 20, 25, 30, 35, 40};
    int size = sizeof(vals)/sizeof(int);

    parallelForEach(vals, vals + size, [](size_t n) {
        cout << fibrec(n) << endl;
    });
    return 0;
}
```

### 2.4.7 Parallel Algorithms in C++17

- Ordering
  - "sequenced-before" is an asymmetric, transitive, pair-wise relationship between evaluations within the same thread

  - A is sequenced before B
    
  - if A is not sequenced before B and B is not sequenced before A, then two possibilities exist:
    * evaluations of A and B are **indeterminately sequenced**: they may be performed in any order but may not overlap
      
    * evaluations of A and B are **unsequenced**: they may be performed in any order and may overlap (within a single thread of execution, the compiler may interleave the CPU instructions that comprise A and B)
      

- Execution Policies
  - most algorithms have overloads that accept execution policies:
  - std::execution::seq — sequential execution like calling the algorithms without an execution policy
  - std::execution::par — execution potentially using multiple threads<br>parallel instructions are indeterminately sequenced<br>is not allowed to cause data races or to cause dead-locks
  - std::execution::par_unseq — execution may be parallelized, vectorized, or migrated across threads<br>parallel instructions and ordering in the same thread: unsequenced<br>use of blocking synchronization primitives (e.g. mutex) may cause dead-lock
  - std::execution::unseq — execution may be vectorized<br>ordering in the same thread: unsequenced

#### Execution Policies and Ordering Examples

```
int a[] = {0, 1};
vector<int> v;
for_each(execution::par, begin(a), end(a), [&](int i) {
    v.push_back(i*2+1);  // Error: data race (vector isn't thread safe)
});

int x = 0;
mutex m;
for_each(execution::par, begin(a), end(a), [&](int) {
    lock_guard<mutex> guard(m);
    ++x;     // Correct, because mutual exclusion is guaranteed
});          // and sequence among parallel lambdas is irrelevant

for_each(execution::par_unseq, begin(a), end(a), [&](int) {
    lock_guard<mutex> guard(m);     // Error: calls m.lock() and several of these
    ++x;                            // calls are unsequenced and can interleave
})
```

### 2.5 OpenMP

- A Standard for Directive Based Parallel Programming
  - OpenMP is a directive-based API that can be used with
    * FORTRAN
    * C/C++

    for programming shared address space machines
- OpenMP directives provide support for
  - concurrency

  - synchronization
  - data handling

  while obviating the need for explicitly setting up
  - mutexes
  - condition variables
  - data scope
  - initialization
- standard specifications: `http://www.openmp.org/specifications`
  - Visual Studio 2019 only supports OpenMP standard 2.0

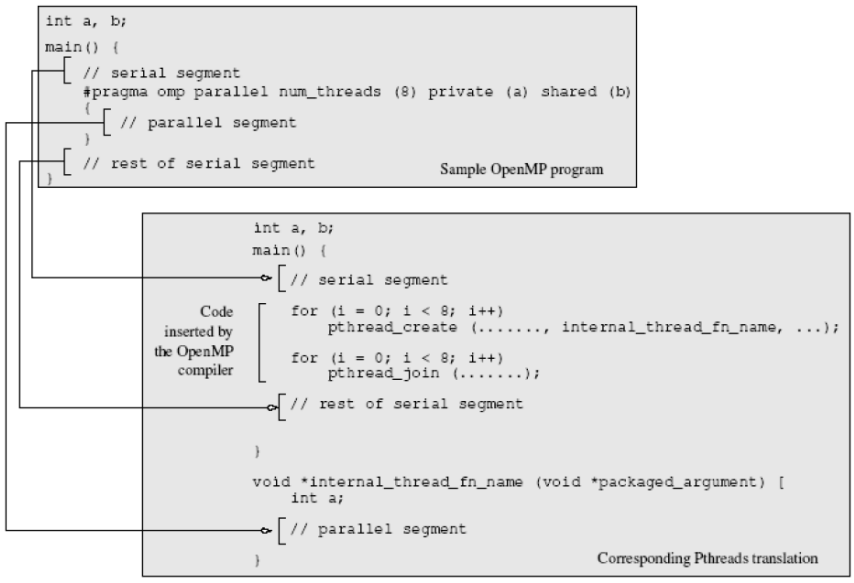
### 2.5.1 OpenMP Programming Model

- OpenMP in C/C++
  - directives are based on the #pragma compiler directives
  - a directive consists of a directive name followed by clauses
    #pragma omp directive [clause list]
- OpenMP programs
  - execute serially until they encounter the parallel directive, which creates a group of threads

```
#pragma omp parallel [clause list]
/* structured block */
```

  - the main thread that encounters the parallel directive becomes the **master** of this group of threads and is assigned the thread **id 0** within the group
  - at the end of the parallel executed block the main thread waits for all parallel threads (join)

## 2.6 OpenMP: behind the scene




### 2.6.1 Clause List in OpenMP

- Clause List specifies
  - Conditional Prallelization
    - the clause if (scalar expr) determines whether the parallel construct results in creation of threads
    - the scalar expression is evaluated at runtim
  - Degree of Concurrency
    - the clause num_threads(integer expr) specifies the number of threads that are created

  - Data Handling
    - the clause private (variable list) indicates variables local to each thread T
    - the clause firstprivate (variable list) is similar to the private, except values of variables are initialized to corresponding values before the parallel directive
    - the clause shared (variable list) indicates that variables are shared across all the threads
- Example

```
#pragma omp parallel if(is_parallel == 1) num_threads(8) \
private(a) shared(b) firstprivate(c)
{
  /* structured block */
}
```

### 2.6.2 Default Clause in OpenMP

- Syntax
  - default(shared — none)
- Semantic
  - the default clause allows the user to affect the data-sharing attributes of variables
  - omitting this clause is the same as the default(shared)
- default(shared)
  - is equivalent to explicitly listing each currently visible variable in a **shared** clause, unless it is **threadprivate** or **const**-qualified
- default(none)
  - it is usually better style to use default(none) instead of default(shared)
  - requires that at least one of the following must be true for every reference to a variable in the lexical extent of the parallel construct
    - the variable is explicitly listed in a data-sharing attribute clause
    - the variable is declared within the parallel construct
    - the variable is **threadprivate** or has a **const**-qualified type
    - the variable is the loop control variable for a **for**-loop that immediately follows a **for** or **parallel for**-directive, and the variable reference appears inside the loop

### 2.6.3 Default Clause in OpenMP

- Reduction Clause
  - specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit
  - the usage is
    reduction (operator: variable list)
  - the variables in the list are implicitly specified as being private to threads
  - the operator can be one of +, *, −, &, |, ˆ, &&, ||
- Example

```
#pragma omp parallel default(none) reduction(+: sum) num_threads(8)
{
  /* compute local sums here */
  sum = ...;
}
/* sum here contains sum of all local instances of sums */
```

### 2.6.4 OpenMP Programming: Example

```
int main() {
  const int npoints = 10000000;
  int sum = 0;
  mt19937_64 re; // random engine
  uniform_real_distribution<double> dist;
  #pragma omp parallel default(none) reduction(+: sum) num_threads(8)
  {
    #pragma omp for
    for (int i = 0; i < npoints; i++) {
      if (hypot(dist(re), dist(re)) < 1) sum++
    }
  }
  cout << setprecision(10) << 4.0*sum/npoints << endl;
}
```

### 2.6.5 OpenMP Directives

| Directive | Description |
|---|---|
| atomic | Specifies that a memory location will be updated atomically. |
| barrier | Synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier. |
| critical | Specifies that code is only executed on one thread at a time. |
| flush | Specifies that all threads have the same view of memory for all shared objects. |
| for | Causes the work done in a for-loop inside a parallel region to be divided among threads. |
| master | Specifies that only the master thread should execute a section of the program. |
| ordered | Specifies that code inside a parallelized for-loop should be executed by multiple threads in parallel. |
| sections | Identifies code sections to be divided among all threads. |
| single | Specifies that section of code should be executed on a single thread, not necessarily the master thread. |
| threadprivate | Specifies that a variable is private to a thread. |

## 2.7 Concurrent tasks in OpenMP

- The for-directive
  - specifies concurrent iterations
  - is used to split parallel iteration spaces across threads
  - the general form is

```
#pragma omp for [clause list]
/* for loop */
```

  - allowed clauses
    - private
    - firstprivate , lastprivate
    - reduction
    - schedule
    - nowait
    - ordered

- The sections-directive
  - specifies concurrent tasks
  - the general form is

```
#pragma omp sections [clause
    list]
{
  #pragma omp section
    /* structured block 1 */
  #pragma omp section
    /* structured block 2 */
  // ...
}
```

### 2.7.1 The for Directive

- Schedule Clause
  - deals with the assignment of iterations to threads
  - the general form of the schedule directive is
    schedule( scheduling_class [, parameter])
- four scheduling classes
  - static
    - splits the iteration space into equal chunks of size *parameter* and assigns them to threads in a round-robin fashion
    - when no *parameter* is specified, the iteration space is split into equally sized chunks, one chunk per thread
  - dynamic
    - the iteration space is partitioned into chunks of size *parameter* (default value: 1)
    - these chunks are assigned to threads as they become idle
  - guided
    - the chunk size is reduced exponentially as each chunk is dispatched to a thread
    - the *parameter* specifies the smallest chunk size (default value: 1)
  - runtime
    - the environment variable OMP_SCHEDULE determines at runtime the scheduling class and the chunk size

### 2.7.2 The schedule Clause: Example

```
/* static scheduling of matrix multiplication loops */
#pragma omp parallel default(none) shared(a, b, c) num_thread(4)
  #pragma omp for schedule(static)
  for (int i = 0; i < 128; i++) {
    for (int j = 0; i < 128; j++) {
      c(i,j) = 0;
      for (int k = 0; k < 128; k++) {
        c(i,j) += a(i, k)*b(k, j);
      }
    }
  }
```

### 2.7.3 The nowait Clause

- Implicit Barrier
  - at the end of the parallel for-loop all threads join
- Clause nowait
  - often, it is desirable to have a sequence of for-directives within a parallel construct that do not execute an implicit barrier at the end of each for directive
- Example

```
#pragma
{
  #pragma omp for nowait
  for (int i = 0; i < nmax; i++)
    if (isEqual(name, current_list[i])) processCurrName(name);
  #pragma omp for
  for (int i = 0; i < nmax; i++)
    if (isEqual(name, past_list[i])) processPastName(name);
```

```
}
```

### 2.7.4 The sections Directive

- Sections
  - supports non-iterative parallel task assignment using the sections directive
- Example

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
      taskA();
    }
    #pragma omp section
    {
      taskB();
    }
    #pragma omp section
    {
      taskC();
    }
  }
}
```

### 2.7.5 Merging Directives

- Remember
  - the **parallel** directive creates the group of threads
  - the for and the sections directive would execute serially (by the master thread) if no parallel directive is specified before
- Merging parallel and for directives

```
#pragma omp parallel shared(n)
{
  #pragma omp for
  for (int i = 0; i < n; i++) {
      // ...
  }
}
```

```
#pragma omp parallel for shared(n)
for (int i = 0; i < n; i++) {
  // ...
}
```

### 2.7.6 Nesting parallel Directives

- Example

```
#pragma omp parallel for shared(a, b, c) num_threads(4)
for (int i = 0; i < 128; i++) {
  #pragma omp parallel for shared(a, b, c) num_threads(4)
  for (int j = 0; j < 128; j++) {
    c(i,j) = 0;
```

```
      #pragma omp parallel for shared(a, b, c) num_threads(4)
      for (int k = 0; k < 128; k++) {
        c(i,j) += a(i, k)*b(k, j);
      }
    }
  }
```

- Remarks
  - OpenMP does not allow nested for, sections, and single directives that bind to the same parallel directive
  - each for directive brings its own parallel directive, which only generates a logical team of threads on encountering a nested parallel directive
  - the newly generated logical team is still executed by the same thread corresponding to the outer parallel directive
  - to generate a new set of threads, nested parallelism must be enabled by setting the OMP_NESTED environment variable to TRUE

## 2.8 Synchronization in OpenMP

- Synchronization Constructs
  - #pragma omp barrier
  - #pragma omp single [clause list]
    /* structured block */
  - #pragma omp master
    /* structured block */
  - #pragma omp critical [(name)]
    /* structured block */
  - #pragma omp atomic
    /* memory update instruction */
  - #pragma omp ordered
    /* structured block */
  - #pragma omp flus [(variable list)]

### 2.8.1 Example: Prefix Sums

```
cumulSum[0] = list[0];
#pragma omp parallel for default(none) shared(cumulSum, list) ordered

for (int i = 1; i < n; i++) {
  // other work
  #pragma omp ordered
  {
    cumulSum[i] = cumulSum[i - 1] + list[i];
  }
}
```

### 2.8.2 Data Handling in OpenMP

- Which data class should you use when?
  - private
    - a thread initializes and uses a variable and no other thread accesses the data
    - it is better to use a local variable in an omp block

    - if multiple threads manipulate different parts of a large data structure, the programmer should explore ways of breaking it into smaller data structures and making them private to the thread that manipulates them
  - firstprivate
    - a thread repeatedly reads a variable that has been initialized earlier in the program
  - reduction
    - if multiple threads manipulate a single piece of data, one must explore ways of breaking these manipulations into local operations followed by a single global operation (reduction)
  - threadprivate( variable list )
    - all variables in the list are local to each thread and are initialized once before they are accessed in a parallel region
    - these variables persist across different parallel regions provided dynamic adjustment of the number of threads is disabled and the number of threads is the same

## 2.9 OpenMP library functions

- Thread and Processor Count
  - #include <omp.h
  - void omp_set_num_threads (int num_threads);
  - int omp_get_num_threads();
  - int omp_get_max_threads();
  - int omp_get_thread_num();
  - int omp_get_num_procs();
  - int omp_in_parallel();
- Controlling and Monitoring Thread Creation
  - void omp_set_dynamic(int dynamic);
  - int omp_get_dynamic();
  - void omp_set_nested(int nested);
  - int omp_get_nested();
- Mutual Exclusion
  - void omp_init_lock(omp_lock_t *lock);
  - void omp_destroy_lock(omp_lock_t *lock);
  - void omp_set_lock(omp_lock_t *lock);
  - void omp_unset_lock(omp_lock_t *lock);
  - int omp_test_lock(omp_lock_t *lock);

### 2.9.1 Environment Variables in OpenMP

- Environment Variables
  - OMP_NUM_Threads
    - specifies the default number of threads created upon entering a parallel region
  - OMP_SET_DYNAMIC
    - determines if the number of threads can be dynamically changed
  - OMP_NESTED
    - turns on nested parallelism
  - OMP_SCHEDULE
    - scheduling of for-loops if the clause specifies runtime
- Common Mistakes in OpenMP programs
  - `http://michaelsuess.net/publications/suess_leopold_common_mistakes_06.pdf`

### 2.9.2 OpenMP Standard

- Version 3.1

  - major change since version 2.5
    * tasks added to execution model
- Version 4.0
  - major changes since version 3.1
    * array syntax extended to support array sections
    * proc_bind clause to support thread affinity policies
    * SIMD contructs to support SIMD parallelism
    * device constructs to support execution of devices (e.g. GPU)
    * user defined reductions
    * depend clause to support task dependencies
- Version 5.1
  - major changes since version 4.5
    * extended memory model to distinguish different types of flush operations
    * support of modern C++20

# 3 Performance Metrics for Parallel Systems

## 3.1 Analytical Modeling

### 3.1.1 Analytical Modeling: Basics

- Sequential Runtime
  - A sequential algorithm is evaluated by its runtime (in general, asymptotic runtime as a function of input size).
  - The asymptotic runtime of a sequential program is identical on any serial platform.
- Parallel Runtime
  - The parallel runtime of a program depends on
    - the input size **n**
    - the number of processing elements **p**
    - and the communication parameters of the machine.
  - An algorithm must therefore be analyzed in the context of the underlying platform.
- Parallel System
  - A parallel system is a combination of a parallel algorithm and an underlying parallel platform.

## 3.2 Intuitive Performance Measures

- Wall-clock time
  - the time from the start of the first processor to the end of the last processor in a parallel ensemble
  - Problem: How does this scale when the number of processors is changed or the program is ported to another machine?
- How much faster is the parallel version?
  - Answering this question depends on the answers of other questions
    - What's the baseline serial version with which we compare?
    - Can we use a suboptimal serial program to improve our parallel program?
- Raw floating-point operations (Flop) count
  - Problem: What good are Flop counts when they don't solve a problem?

## 3.3 Sources of Overhead in Parallel Programs

- If I use two processors, shouldn't my program run twice as fast?
- **No!**
    - Several overheads, including wasted computation, communication, idling, and contention cause degradation in performance.
  - interprocess interactions
    - Processors working on any non-trivial parallel problem will need to talk to each other.
  - idling
    - Processes may idle because of load imbalance, synchronization, or serial components.
  - excess computation
    - This is computation effort not performed by the serial version.
      This might be because the serial algorithm is difficult to parallelize, or that some computations are repeated across processors to minimize communication.

## 3.4 Performance Metrics for Parallel Systems

- Execution Time
  - Serial runtime $T_S$
    - the time elapsed between the beginning and the end of its execution on a sequential computer
  - Parallel runtime $T_P$

    - the time that elapses from the moment the first processors starts to the moment the last processor finishes execution (wall-clock time)
- Total Parallel Overhead $T_O = pT_p - T_S$
  - the total time spent by all processors combined in non-useful work
- Speedup $S = \frac{T_S}{T_P}$
  - the ratio of the serial runtime of *the best sequential algorithm* for solving a problem to the time taken by the parallel algorithm to solve the same problem on a parallel computer with $p$ identical processing elements
- Speedup per processor = Efficiency $E = \frac{S}{p} = \frac{T_S}{pT_P}$

### 3.4.1 Speedup Example

- Problem: adding $n$ numbers by using $p = n$ processing elements
  - each processing element owns the number
  - for simplicity: $n$ is a power of two
- Sequential Algorithm
  - best algorithm has to read the entire input: $T_S = \mathcal{O}(n)$
- Parallel Algorithm
  - we can perform this operation in $\mathcal{O}(log(n))$ steps by propagating partial sums up a logical binary tree of processors
  - the addition and the communication of a single word can be performed in constant time
  - $T_P = \mathcal{O}(log(n))$
- Speedup and Efficiency
  - $S = \mathcal{O}(\frac{n}{log(n)})$
  - $E = \frac{S}{p} = \frac{S}{n} = \mathcal{O}(\frac{1}{log(n)})$

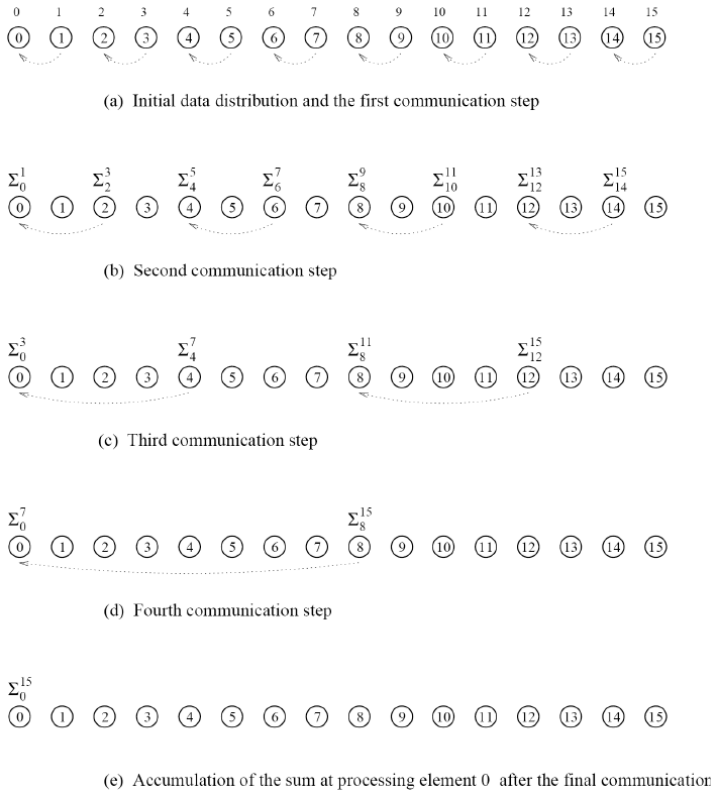



Figure 3: Computing the global sum of 16 partial sums using 16 processing elements. $\sum_i^j$ denotes the sum of numbers with consecutive labels from $i$ to $j$.

### 3.4.2 Speedup Bounds

- Lower bound
  - speedup can be as low as 0 (the parallel program never terminates)
- Upper bound
  - in theory: "the more processors, the more speedup"
    - should be by $p$
  - in practice superlinear speedup is sometimes observed
    - due to cache effects
    - due to exploratory decomposition
  - Example of superlinear speedup with exploratory decomposition

### 3.4.3 Image Processing: Edge Detection

- Problem: edge detection in image processing
  - a possible parallelization partitions the image equally into horizontal segments, each with $\frac{n^2}{p}$ pixels
  - the boundary of each segment is $2n$ pixels
  - boundaries must be communicated in time $2(t_s + t_w n)$
  - convolution: 3x3 template is applied to all $n^2/p$ pixels in time $T = \frac{9t_c n^2}{p}$, where $t_c$ is the time for one multiply-add operation
- The total time for the algorithm is therefore given by:

$$T_P = 9t_c \frac{n^2}{p} + 2(t_s + t_w n)$$

The corresponding values of speedup and efficiency are given by:

$$S = \frac{9t_c n^2}{9t_c \frac{n^2}{p} + 2(t_s + t_w n)}$$

and

$$E = \frac{1}{1 + \frac{2p(t_s + t_w n)}{9t_c n^2}}$$

### 3.4.4 Cost of a Parallel System

- Cost (amount of total work)
  - is the product of parallel runtime and the number of processing elements used
  - Cost $= p \cdot T_P$
- Cost-Optimal system
  - a parallel system is said to be cost-optimal if the cost of solving a problem on a parallel computer is **asymptotically identical** to serial cost
  - since $E = \frac{T_S}{p \cdot T_P}$, for cost-optimal systems: $E = \mathcal{O}(1)$
- Example: adding $n$ numbers on $p = n$ processors
  - $T_P = log(n)$
  - Cost $= p \cdot T_P = nlog(n)$
  - $E = \frac{T_S}{\text{Cost}} = \frac{\mathcal{O}(n)}{\mathcal{O}(nlog(n))} = \mathcal{O}(\frac{1}{log(n)}) \neq \mathcal{O}(1)$
  - since the serial runtime of this operation is $\mathcal{O}(n)$, the algorithm is **not cost-optimal**

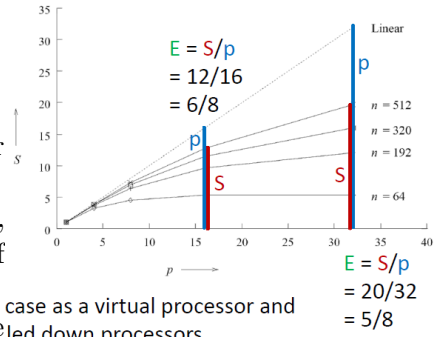
## 3.5 Impact of Non-Cost-Optimality

- Example: Sorting a list of $n$ numbers
- using $p = n$ processors
  - we assume a sorting algorithm that takes time $(log(n))^2$ (e.g. Bitonic Sort)
  - $S = \frac{nlog(n)}{log^2(n)} = \frac{n}{log(n)}$
  - $E = \frac{S}{p} = \frac{S}{n} = \frac{1}{log(n)}$
  - Cost $= pT_p = nlog^2(n)$
    $\rightarrow$ the algorithm is not cost-optimal but only by a factor of $log(n)$
- using $p < n$ processors
  - $T_p = \frac{nlog^2(n)}{p}$
  - $S = \frac{nlog(n)}{T_P} = \frac{p}{log(n)}$
  - $E = \frac{S}{p} = \frac{1}{log(n)}$

  $\rightarrow$ speedup goes down as the problem size $n$ is increased for a given $p$
  $\rightarrow$ efficiency doesn't depend on $p$, but goes down as the problem size is increased

## 3.6 Effect of Granularity on Performance

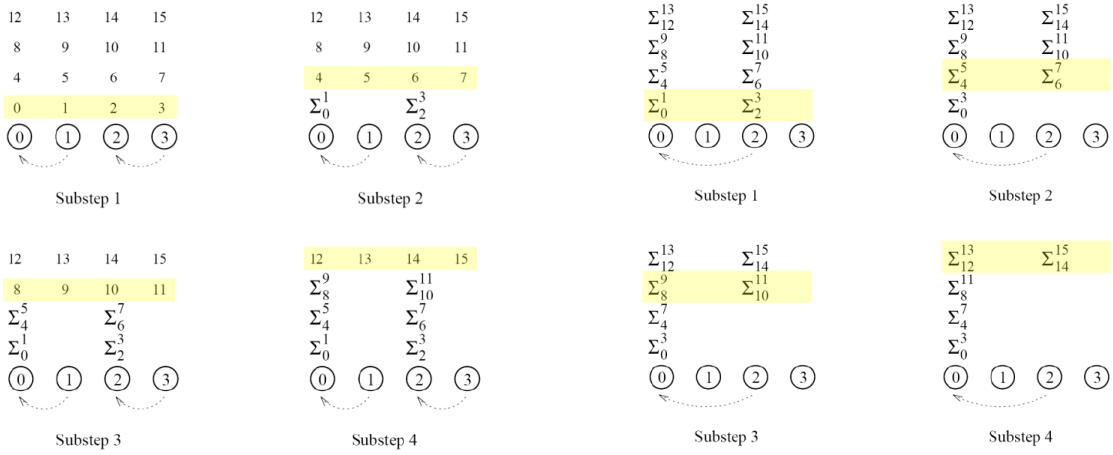
- Scaling-Down a parallel system
  - using fewer than the maximum possible number of processing elements to execute a parallel algorithm
- Observation
  - often, scaled-down parallel systems have an improved efficiency
- A naive way of scaling down is
  - to think of each processor in the original case as a virtual processor and to assign virtual processors *equally* to scaled-down processors
  - since the number of processing elements decreases by a factor of $n/p$, the computation at each processing element increases by a factor of $n/p$
  - the communication cost should not increase by this factor since some of the virtual processors assigned to a physical processor might talk to each other



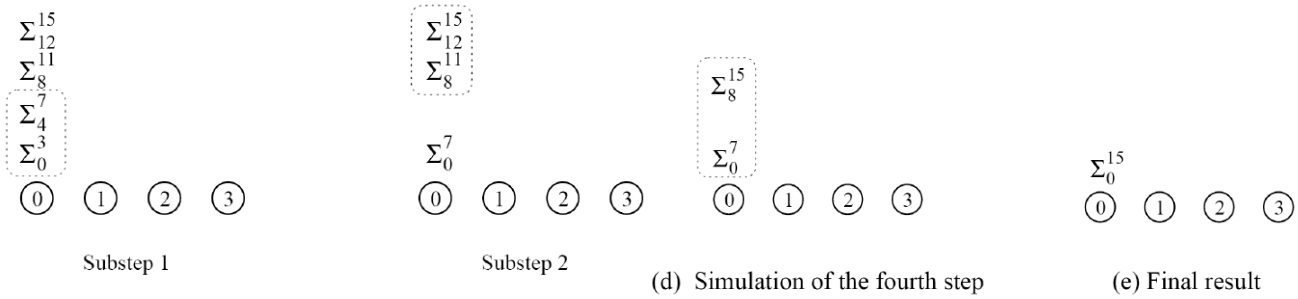

case as a virtual processor and
led down processors

### 3.6.1 Effect of Granularity: Example

- Problem
  - adding $n$ numbers on $p$ processing elements such that $\mathbf{p} < \mathbf{n}$ both $n$ and $p$ are powers of 2
- Naive Approach
  - use the parallel algorithm for $n$ processors, except, in this case, we think of them as virtual processors
  - each of the $p$ processors is now assigned $\mathbf{n/p}$ virtual processors
  - the first $log(p)$ of the $log(n)$ steps of the original algorithm are simulated in $(\mathbf{n/p})\mathbf{log(p)}$ steps on $p$ processing elements
  - subsequent $(log(n) - log(p))$ steps of the original algorithm do not require any communication and are processed in $(\mathbf{n/p})$ steps
    $\rightarrow T_P = \mathcal{O}(\frac{n}{p}log(p)) + \mathcal{O}(\frac{n}{p}) = \mathcal{O}(\frac{n}{p}log(p))$
    $\rightarrow$ Cost $= \mathcal{O}(nlog(p))$, it is not cost-optimal by a factor of $log(p)$




(a) Four processors simulating the first communication step of 16 processors (b) Four processors simulating the second communication step of 16 processors




(c) Simulation of the third step in two substeps

(d) Simulation of the fourth step

(e) Final result

- Cost-optimal Approach
  - each processing element locally adds its $n/p$ numbers in time $\mathcal{O}(n/p)$
  - the $p$ partial sums on $p$ processing elements can be added in time $log(p)$
    $\rightarrow T_P = \mathcal{O}(\frac{n}{p} + log(p))$
    $\rightarrow$ Cost $= \mathcal{O}(n + plog(p))$
    $S = \frac{n}{\frac{n}{p}+log(p)}$
    $E = \frac{1}{1+p\frac{log(p)}{n}}$
  - this is cost-optimal, if $plog(p) = \mathcal{O}(n)$ or if $n = \mathcal{O}(plog(p))$