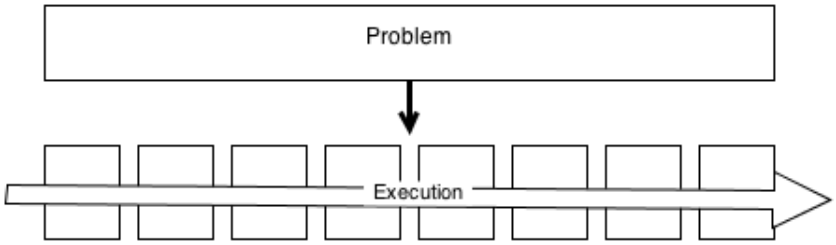
Intro to Parallel Computing

1 Intro

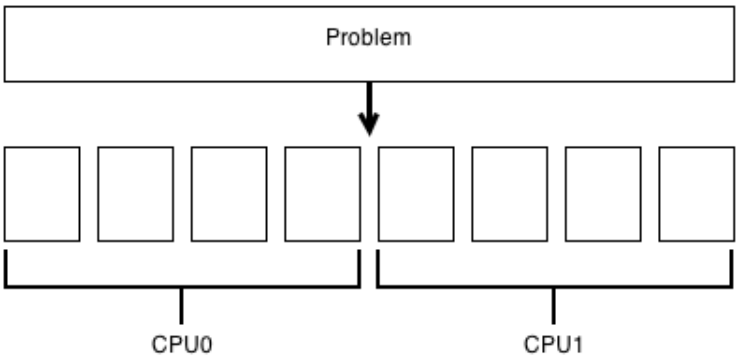
- **Parallel Processing**: simultaneous use multiple processing elements to solve problem fast
- **Processing Elements (PE)**:
  - Single Processor, multiple Cores
  - Simple Computer, multiple processors
  - Many computers, connected via network
  - Combination of above
- Problem need to be **partitioned** into sufficient **independent parts** for execution on **parallel PE's**

2 Serial Computing



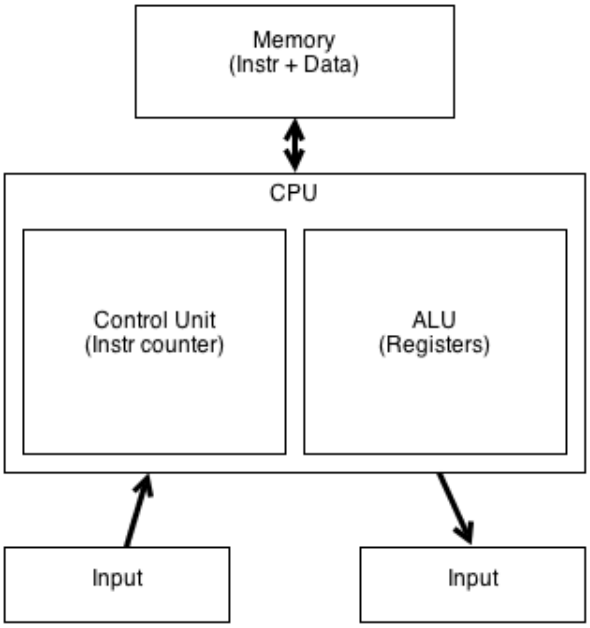
- **Problem ⇒ discrete series of instructions**
- Executed in series (1 at a time)

3 Parallel Computing



- **Problem ⇒ Discrete tasks**
- Solved concurrently
- Instructions of each part execute in parallel on different PEs

4 Von Neumann Computation Model



5 Why Parallel Computing

Exploits large number of PEs that communicate and coorperate to **solve large problems fast**

Primary reasons

- Overcome limits of serial computing
- Save (wall clock) time
- Solve large problems

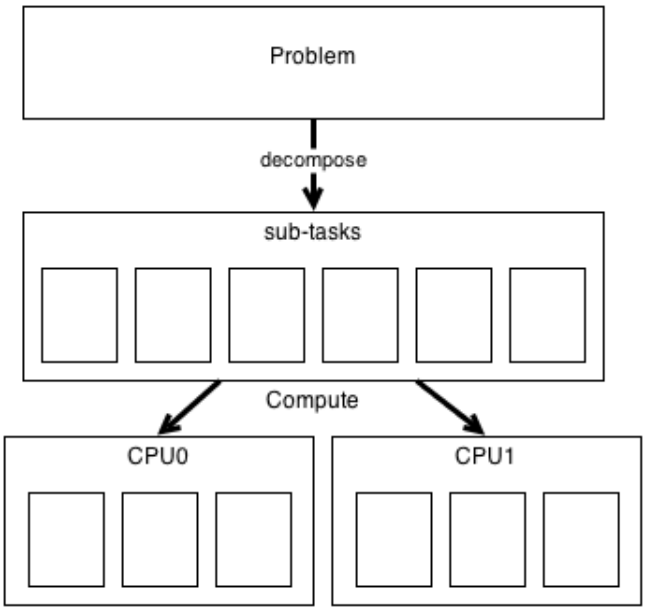
Secondary reasons

- Take advantage of non-local
- Cost savings - use multiple cheaper commoditized computing resources
- Overcome memory constraints

6 How?

- Work hard (serial)
- Work smart (optimize/algorithms)
- Get help (go parallel)

7 Parallel Computing Basics



7.1 Decomposition

- **Potential parallelism** in problem dictates how it should be split
- **Granulatity**: size of tasks

7.2 Tasks

- **Scheduling**: tasks ⇒ processes/threads
- **Mapping**: processess/threads ⇒ cores/processors

7.3 Dependencies & Coordination

- Dependencies constrain scheduling
- To execute correctly, syncronization & coordination needed

7.4 Performance

- Throughput vs time
- Parallel execution time = Computation time + (data exchange & scynronization time)

7.5 Challenges

- Optimize execution resources (less idle)
- How to share memory correctly
- Automate extraction of parallelism: compilers, language tooling
- Verification of correctness
- Debugging
- Performance monitoring

CS3210 - 1 - Processor Architecture & Memory

1 Processor Architecture & Trends

- **Goal**: ↓ average time for executing instr.
- **Trends**:
  - ↑ Transistors
    - ↑ clk. speed ⇒ ↑ speed
    - Moore's law: number of transistors doubles every 18-24 months
  - Parallelism w/in single
    - At bit level
      - Word size ↑ to 64bits
      - Improved floating point accuracy
      - ↑ address space
    - By pipelining
      - Partition instructions into steps (FETCH, DECODE, EXECUTE, WriteBack)
      - Steps performed by dedicated hardware units (pipeline stages)
      - Units work in parallel as long as theres no dependencies between instructions
      - Throughput: 1 instr. per clock cycle
    - By Multiple Functional Units
      - Processor use multiple, independent functional units (ALU, FPU, Load/Store, Branch)
      - Independent instr. executed in parallel by different functional unit
    - Process/Thread Level
      - Multi-core on single chip
      - "Hyper Threading": Processor appear to OS as set of logical processor (LP), which stores processor state in separate process resource. Cache, Bus, Functional & Control units shared between LP, but memory accesses must be coordinated

  - Multi**processors** (Shared memory)
  - Multi**computers** (Distributed memory)

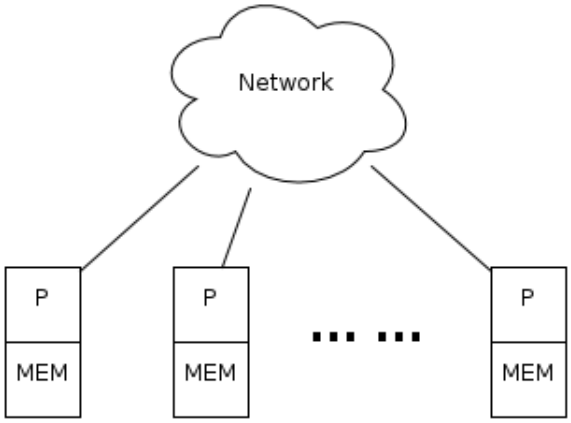
2 Flynn's Taxonomy of Parallel Architectures

- **Parallel Computer**: collection of processing elements that communicate & cooperate to solve large problems fast
- **MIMD**: Multiple Instruction, Multiple Data
  - * CU → * PU → 1 Shared MEM
- **SIMD**: Single Instruction, Multiple Data
  - 1 CU → * PU → * Local MEM
- ~~**SISD**~~: Single Instruction, Single Data
  - 1 CU → 1 PU → 1 MEM
- ~~**MISD**~~: Multiple Instruction, Single Data
  - * CU → * PU → 1 MEM

3 Memory Organization

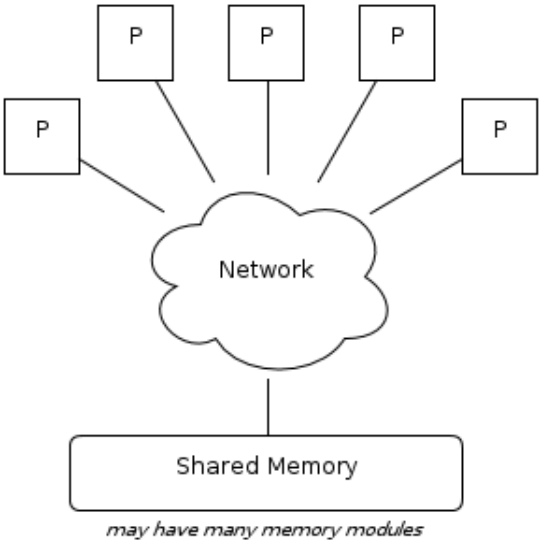
- Distributed (Multicomputers)
- Shared (Multiprocessors)
  - Uniform Memory Access (UMA)
  - Non-Uniform Memory Access (NUMA)
  - Cache-only Memory Access (COMA)

3.1 Distributed Memory (Multi-Computers)



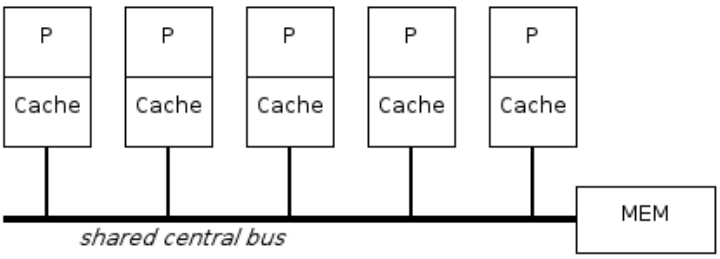
- Node contains Processor (P), Memory and perhaps other periphery elements
- Physically distributed memory. Memory on each node is private
- Data exchange via Message Passing

3.2 Shared Memory (Multiprocessors)



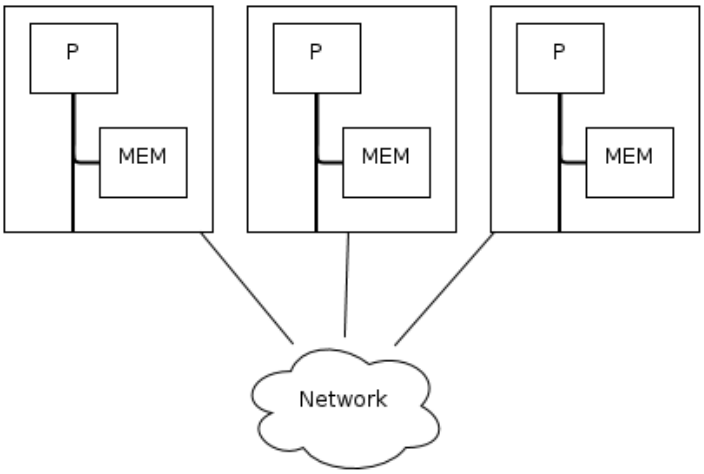
- Global Memory, Common address space
- Data exchange of shared variables via threads
- Problems: Race conditions, fast access to global memory

3.2.1 Uniform Memory Access



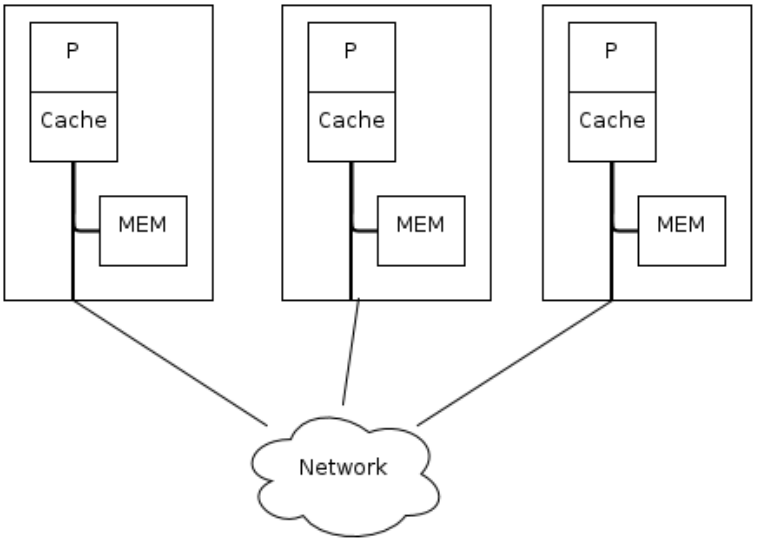
- Memory equidistant to processors
- Processors connect to memory via Central Bus

3.2.2 Non Uniform Memory Access



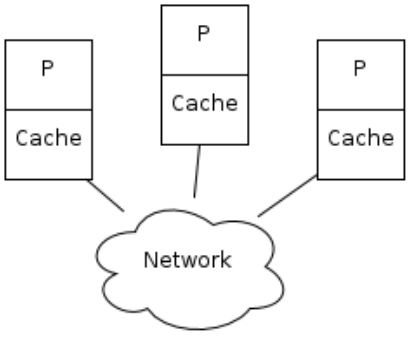
- Memory physically distributed
- aka Distributed Shared Memory

3.2.3 Cache Coherent NUMA



- NUMA with Caches that are kept in sync by cache coherence protocol

3.2.4 Cache Only Memory Access



4 Reduce Memory Access Time

- Main reason: Memory Wall (Memory access time: 100-1000 cycles)
- Techniques
  - Multithreading
  - Cache

4.1 Threading

- Thread is a separate control flow which shares data with other threads via global address space
- Interleaved threads hide latency of memory access

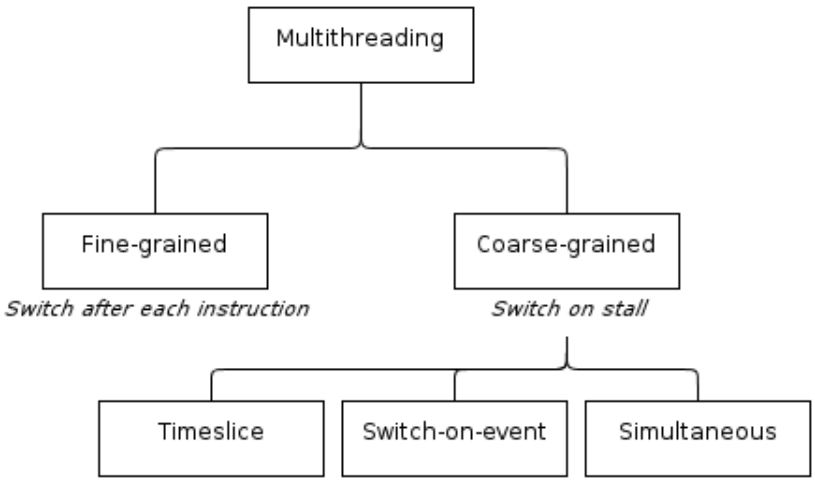
4.2 Caching

- Reduce access to slow memory
- In multiprocessors, multiple caches to reduce latency, but each processor should have coherent
- Cache coherancy problem - read should return most recently written value

5 Thread Level Parallelism

- Exploit multiple threads to efficiently use resources on single processor chip
- Architecture organization: Chip Multiprocessing.
  - Example: multiple execution cores on single chip (multicore processor)

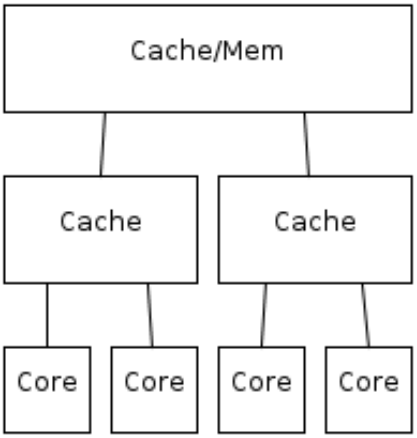
5.1 Multithreading Classification



5.2 Architecture

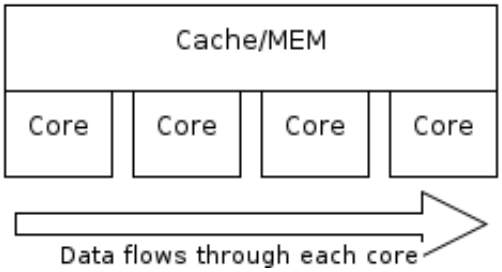
- Hierarchical
- Pipelined
- Network

5.2.1 Hierarchical



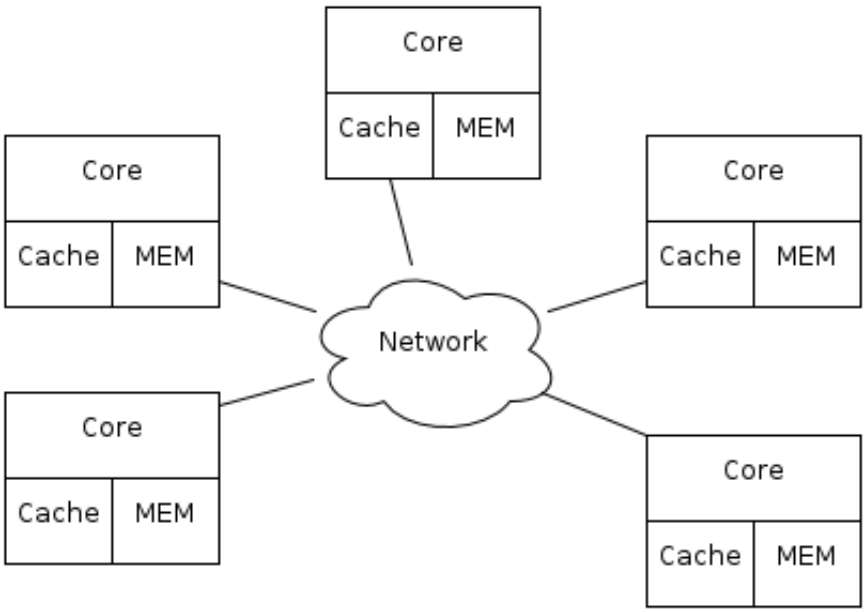
- Multiple cores share multiple caches
- Cache size ↑ from leaves (near processor) to root (near memory)
- L1 private, L2 shared
- All cores share common external memory
- Usage:
  - Desktop
  - Server
  - GPU

5.2.2 Pipelined



- Data elements processed by multiple execution cores in pipelined manner
- Useful when same computation steps applied to long sequence of data elements
- Usage: Network processors (routers/switches)

5.2.3 Networked



- Cores, local cache & memory connected via network

6 Future Trends

- Efficient on chip interconnection
  - Bandwidth for data transfer between cores
  - Scalable
  - Robust to failures
  - Energy efficient energy management
  - Reduce memory access time

CS3210 - 3 - Memory Hierarchy & Networks

1 Caching

- **Why**: Memory Wall
- Issues:
  - Shared cache: cache coherency
  - Shared memory: memory consistency
- Levels
  - L1: usually not shared
  - L2: may/may not be shared
  - L3: usually shared

1.1 Terminology

- Cache Hit: requested memory belongs to a cache line
- Cache Miss: not in cache, fetch from main memory and block replacement if necessary
- Locality of references:
  - Spatial: often access neighbouring memory at successive points in time
  - Temorial: often access same memory at successive points in time

1.2 Decisions & Implications

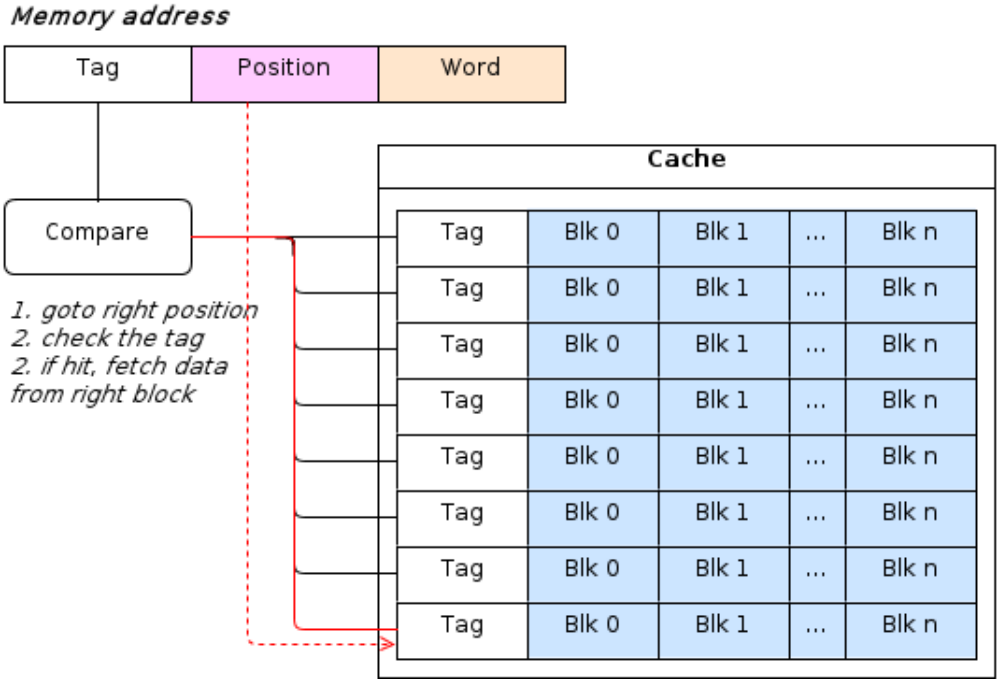
- **Larger Cache size**: ↑ access time, ↓ cache miss
- **Block size**: data transfer between main memory and cache in fixed size
  - **Large Block**: ↓ blocks, ↑ block replacements
  - time(transfer block of x words) $<$ (x $\times$ time(transfer 1 word))

1.3 Mapping Memory to Cache Blocks

- Types:

- Direct Mapped: 1 to 1
- Fully Associative: 1 to Any
- Set Associative: 1 to Some

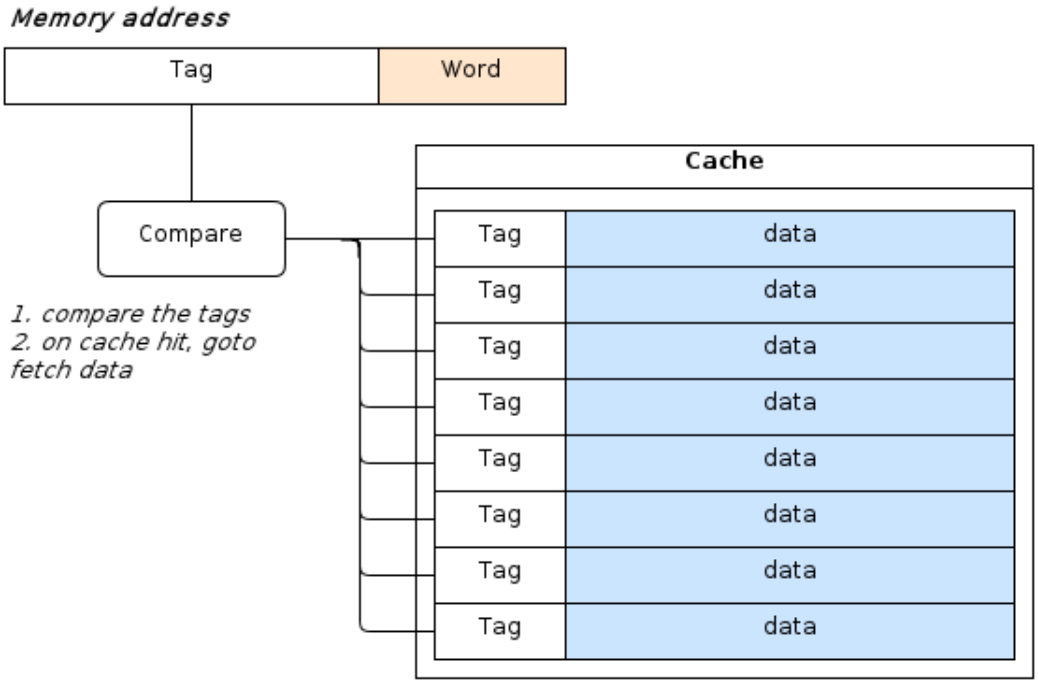
1.3.1 Direct Mapped



$$\boxed{\text{i} = \text{block} \ \% \ \text{size of cache}}$$

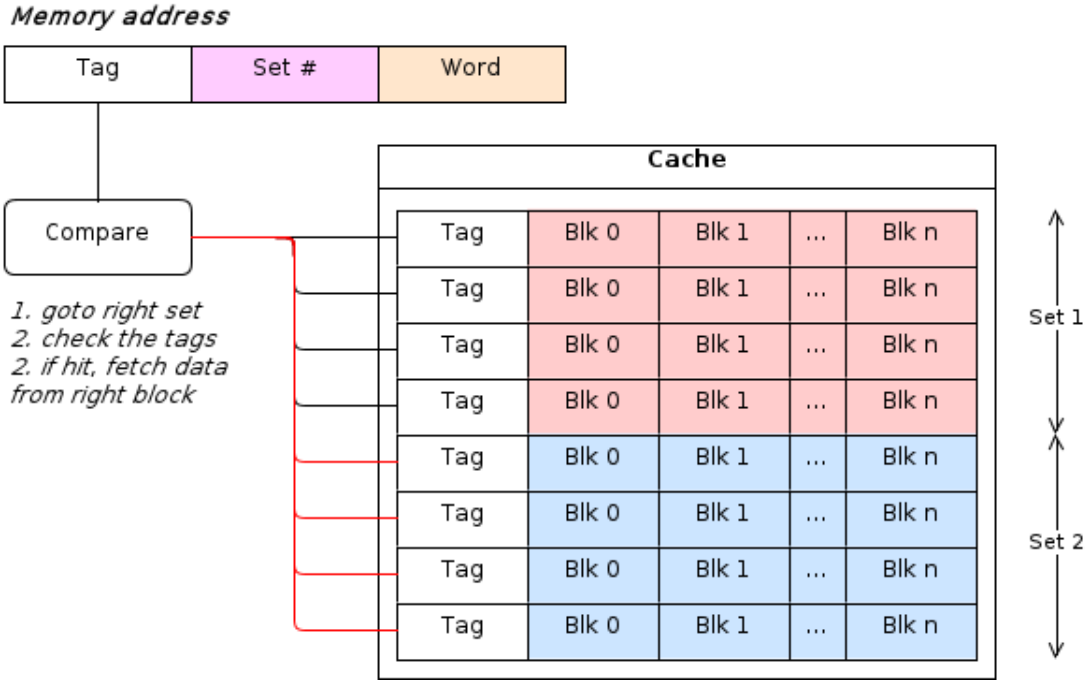
Trashing when memory addresses in different memory blocks are mapped to the same cache position are accessed

1.3.2 Fully Associative



- Memory stored in any cache position
- Flexibility when loading memory blocks
- Need to search all cache positions

1.3.3 Set Associative



- A memory blocked can be mapped to a fixed number of positions
- Cache is partitioned into v sets. Where each set consists of $k = m/v$ blocks

1.4 Block Replacement

- LRU: replace block not used for longest time
- LFU: replace block used the least

1.5 Write Policy

- Memory in cache
  - Write Through
  - Write Back
- Memory NOT in cache
  - Write Allocate

1.5.1 Write Through

- Modification immediately written to memory
- Write buffer used to store pending writes
- Cache & Memory consistent
- When write buffer is full, need to wait

1.5.2 Write Back

- Cache changed but memory NOT updated immediately
- Update only on replacement
- Use dirty bit to determine if write necessary
- Fewer write operations to memory
- Memory can contain outdated entries

1.5.3 Write Allocate

- When memory not in cache
- Memory first brought into cache
- then use write through/back

2 Cache coherency problem

Cache coherency ensures processors has same consistent view of memory via local cache

Classification

- **Snooping protocols**: shared broadcast & write through
  - Cache controllers observe all writes to update/invalidate cache block
- **Directory based cache coherence protocols**: no shared broadcast medium

2.1 Sequential consistency model

- Every processor issues memory operations in **program order**

2.2 Relaxed consistency model

- R → W: anti-dependence: first uses variable later used by another to store result
- W → W: output dependence: 2 instructions use same variable to store result
- W → R: flow dependence

- Relax W → R: Processor can execute R even if preceding W has not completed if theres no data dependency
- Relax W → W & W → R: Writes can be completed in different order if theres no data dependency
- Relax everything but provide additional sync mechanisms

3 Interconnection Networks

Topology

- Direct: static/point2point. Nodes connected directly via fixed physical links
- Indirect: dynamic. Nodes connected via switches/links

Routing: What path to take
Switching strategy: how to transfer message. How to cut message, how message is forwarded through path

3.1 Direct

To reduce transmission time

- Buffers can be used to decouple send/recv operations
- DMA (direct memory access) controller to decouple communication and processor operations

To reduce communication time

- Add routers
- which forwards message without interaction from processors (in nodes)

3.2 Properties

3.2.1 Diameter

Max distance between any pair of nodes. Ensure less hops for transmission

3.2.2 Degree

Number of direct neighbours. Less hardware overhead

3.2.3 Bisection bandwidth

Min edges that can be removed to divide network into equal halves. Larger data throughput

3.2.4 Node connectivity

Min nodes that must fail to disconnect network

3.2.5 Edge connectivity

Min edges that must fail to disconnect network

Larger connectivity → higher reliability

Network G with n nodes	Degree $g(G)$	Diameter $\delta(G)$	Edge- connectivity $ec(G)$	Bisection bandwidth $B(G)$
Complete graph	$n-1$	1	$n-1$	$\left(\frac{n}{2}\right)^2$
Linear array	2	$n-1$	1	1
Ring	2	$\lfloor \frac{n}{2} \rfloor$	2	2
d-Dimensional mesh $(n = r^d)$	$2d$	$d(\sqrt[d]{n} - 1)$	d	$n^{\frac{d-1}{d}}$
d-Dimensional torus $(n = r^d)$	$2d$	$d \left\lfloor \frac{\sqrt[d]{n}}{2} \right\rfloor$	$2d$	$2n^{\frac{d-1}{d}}$
k-Dimensional hyper-cube $(n = 2^k)$	$\log n$	$\log n$	$\log n$	$\frac{n}{2}$
k-Dimensional CCC network $(n = k2^k$ for $k \geq 3)$	3	$2k - 1 + \lfloor k/2 \rfloor$	3	$\frac{n}{2k}$
Complete binary tree $(n = 2^k - 1)$	3	$2 \log \frac{n+1}{2}$	1	1
k-ary d-cube $(n = k^d)$	$2d$	$d \lfloor \frac{k}{2} \rfloor$	$2d$	$2k^{d-1}$

3.3 Indirect Networks

Reduce hardware costs by sharing switches and links

- **Bus**: Only 1 communication at a time
- **Crossbar**: Very expensive
- **Multistage (eg. Omega)**: Balanced

4 Routing

Determines a path from source to destination

Desirable properties

- Fast message transmission
- Load balancing

Issues

- Contention: too many messages transmitted via same link
- Congestion: too many messages assigned to a resource → message loss

4.1 Routing Algorithm Classification

4.1.1 Path length

	Path length	**Congestion**
Minimal routing	↓	↑
Non-minimal routing	↑	↓

4.1.2 Network utilization

	Factors	**Network Load**
Deterministic	src & dest	Unbalanced
Non-minimal routing	dynamic	Balanced/fault torlerant

5 Switching

- Determine whether and how message is split into packets/flits (flow control units)
- How messages/packets forwarded from input to output channel of swicth/router

Sending processor

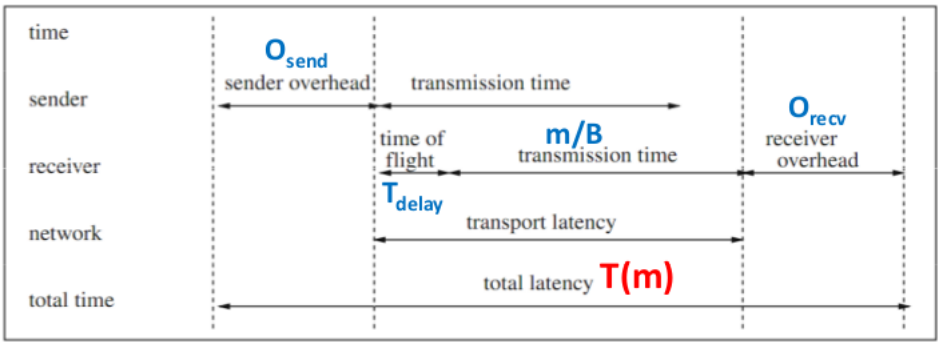
- Message copied to system buffer
- Checksum computed
- Header added to message
- (After sending message) If acknowledgement message arrives, release system buffer. Else resent

Receving processor

- Message copied from network interface to system buffer
- Compare checksum. If identical, copy into user buffer (send ACK). Else discard message, message will be resent by sender

Measure	Definition	Unit
Bandwidth	Maximum rate at which data can be sent	bits (bytes) per second
Byte transfer time	Time to transmit a single byte	Seconds/byte
Time of flight	Time the first bit arrived at the receiver (channel propagation delay)	second
Transmission time	Time to transmit a message	second
Transport latency	Total time to transfer a message = transmission time + time of flight	second
Sender overhead	Time of computing the checksum, appending the header, and executing the routing algorithm	second
Receiver overhead	Time of checksum comparison and generation of an acknowledgment	second
Throughput	Effective bandwidth	bits (bytes) per second

Total Latency of a Message of Size m

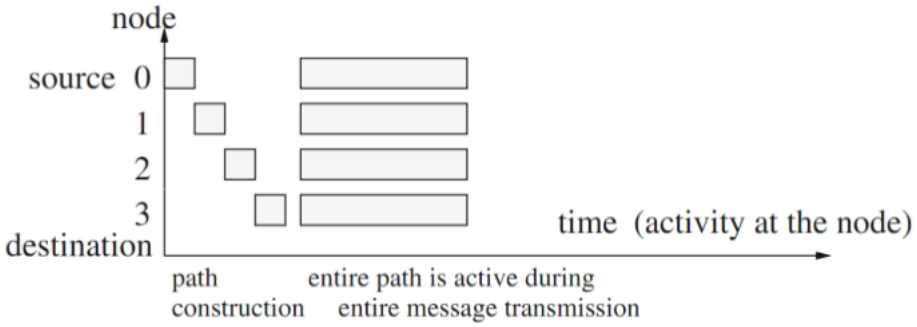


$$T(m) = O_{send} + T_{delay} + m/B + O_{recv}$$
$$= T_{overhead} + m/B = T_{overhead} + t_B * m$$

where B is network bandwidth,
no checksum error and network contention and congestion,
$T_{overhead}$ $(= O_{send} + T_{delay} + O_{recv})$ is independent of the message size;
t_B $(=1/B)$ is the byte transfer time

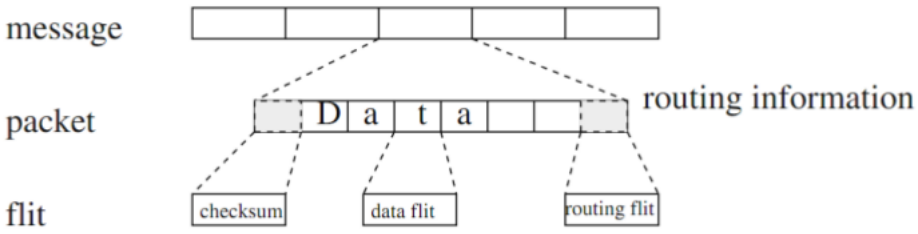
5.1 Switching strategies

- **Circuit**: path reserved until end of transmission

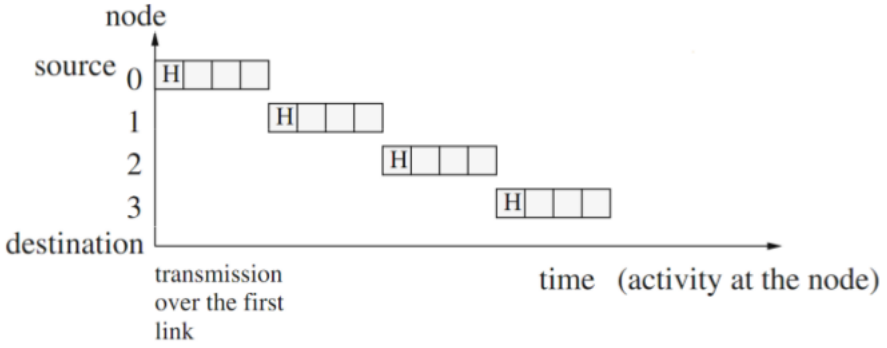


  - $T(m,l) = T_{overhead} + \underbrace{t_B \cdot m_c \cdot l}_{\text{path construction}} + t_B \cdot m$
  - l is length of path
  - m_c is control message size
  - t_B is byte transfer time
  - If $m_c << m$, $T_{cs} = T_{overhead} + t_B \cdot m$

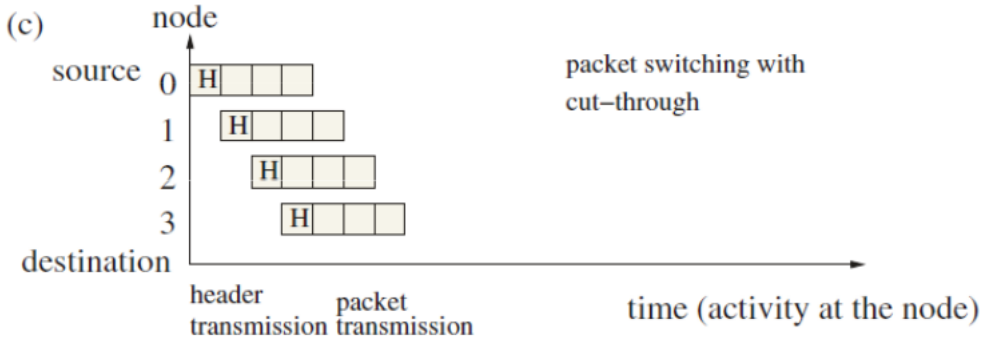
- **Packet** (store & forward routing): message partitioned into sequence of packets



  - with **Store and Forward**: entire packet (received and) stored by a switched before its forwarded to next switch
  - Connection between switches released as soon as packet stored at destination
  - **Advantages**: bandwidth utilization, reduce danger of deadlock
  - **Disadvantages**: Transmission time increases with number of switches. Memory demands on switches
  - $T(m,l) = T_{overhead} + l(t_h + t_B \cdot m)$
  - t_h is time to store packet in buffer
  - Since t_h is typically small, $T \approx T_{overhead} + l \cdot t_B \cdot m$
  - With deterministic routing, transmission time is sum of time to transmit all packets
  - With adaptive routing, transmission time can be overlapped



- **Cut-through routing**: Pipeline each packet through different paths



CS3210 - 4 - Performance

1 Response time

- User CPU time: time for executing program
- System CPU time: time for OS routines
- Waiting time: IO waiting, time sharing

1.1 User CPU Time

$T_{U\_CPU} = n_{cycle}(A) \cdot t_{cycle}$

- t_{cycle} = clock cycle time = 1/clock rate

$c_{cycle}(A) = \sum_{i=1}^{n} n_i(A) \cdot CPI_i$

$$\boxed{T_{U\_CPU} = n_{instr}(A) \cdot CPI(A) \cdot t_{cycle}}$$

1.2 MIPS (Million Instructions Per Instruction

$$\boxed{MIPS(A) = \frac{n_{instr}(A)}{T_{U\_CPU}(A) \cdot 10^6}}$$

$$\boxed{MIPS(A) = \frac{r_{cycle}}{CPI(A) \cdot 10^6}}$$

1.3 MFLOPS

$$\boxed{MFLOPS(A) = \frac{n_{flt\_op}(A)}{T_{U\_CPU}(A) \cdot 10^6} \cdot \frac{1}{s}}$$

2 Memory Access Time with Caches

$$T_{U\_CPU} = (n_{cycle}(A) + n_{mem\_cycle}(A)) \cdot t_{cycle}$$

Considering 1 level cache miss

$$_{mem\_cycle} = n_{read\_cycle}(A) + n_{write\_cycle}(A)$$

$$n_{read\_cycle}(A) = n_{read}(A) \cdot r_{read\_miss}(A) \cdot n_{miss\_cycle}$$

n_{miss_cycle}: additional cycles required for loading new cache line

Combined

$$T_{U\_CPU} = n_{instr}(A) \cdot (CPI(A) + \underbrace{n_{rw} \cdot r_{miss} \cdot n_{miss\_cycles}}_{\text{additional cycles due to miss}}) \cdot T_{cycle}$$

Multi level cache

$$t_{read\_access} = \underbrace{t_{read\_hit} + r_{read\_miss} \cdot t_{read\_miss}}_{\text{L1}}$$

$$t_{read\_miss} = \underbrace{t_{read\_hit} + r_{read\_miss} \cdot t_{read\_miss}}_{\text{L2}}$$

Global read miss rate

$$r_{read\_miss\_L1} \cdot r_{read\_miss\_L2} \cdots$$

3 Parallel execution time

Parallel runtime, $T_p(n)$, where n refers to problem size and p means execution on p processors. Refers to time between start and end of **all** p processors. Consists of

- Computation
- Data exchange
- Syncronization
- Waiting time (unequal load, wait for shared resources)

3.1 Cost of parallel program

$$C_p(n) = p \cdot T_p(n)$$

Cost optimal if executes same number of operations as fastest sequential program

3.2 Speedup

Saving in execution time (benefit of parallelism)

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

- $T_p(n)$: parallel execution time
- $T^*(n)$: best sequential execution time

3.3 Efficiency

Actual degree of speedup acheived compared to maximum

$$E_p(n) = \frac{T^*(n)}{C_p(n)} = \frac{S_p(n)}{p} = \frac{T^*(n)}{p \cdot T_p(n)}$$

Ideal speedup, $S_p(n) = p$ corresponds to $E_p(n) = 1$

4 Amdahl's Law (Fixed problem size)

Improvement in performance in parallel algorithm over sequential algorithm is limited by fraction which cannot be parallelized (f, sequential fraction)

$$S_p(n) \leq \frac{1}{f}$$

If 10% of program is sequential, max speedup is $1/0.1 = 10$

5 Gustafson's Law (Fixed execution time)

As problem size gets large and assuming parallel program is perfectly parallelizable without overheads

$$\lim_{n \to \infty} S_p(n) = p$$

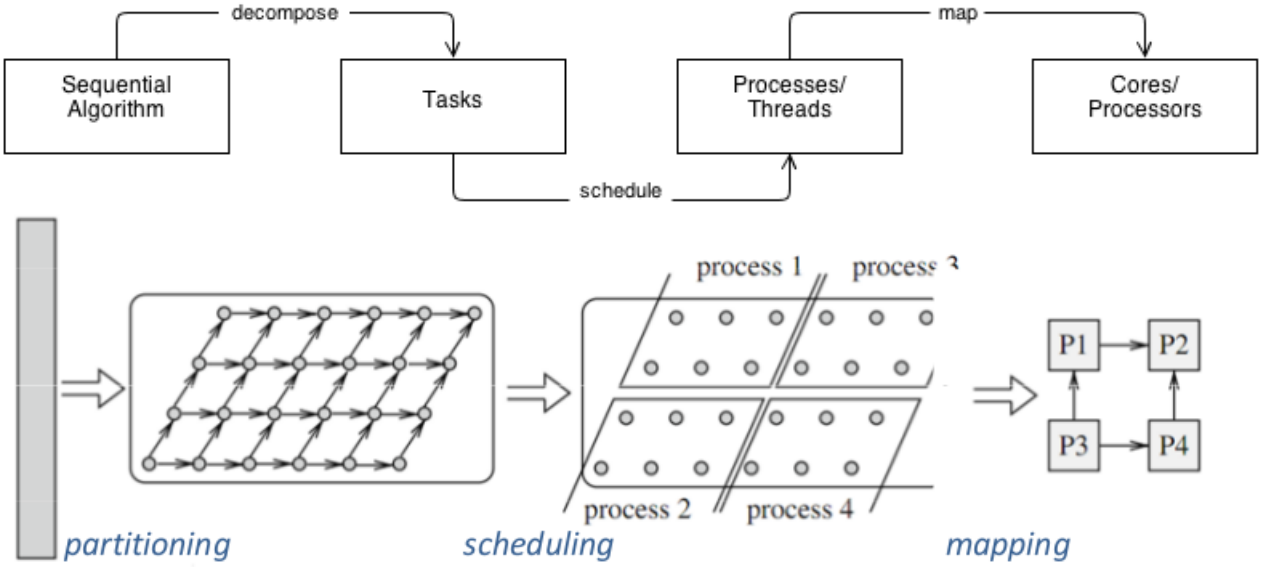
Parallel Programming Models

1 Parallel Programming Models

1.1 Classification

- **Level of parallelism**: instruction, statement, procedural levels or parallel loops
- **Implicit/Explicit**
- **Processes/Threads**
- **Execution Mode**: SIMD, SPMD, Sync/Async
- **Communication modes**: Message passing/Shared variables
- **Syncronization mechanisms**

1.2 Program Parallelization



1.2.1 Decomposition

- Sequential algorithm split into tasks/dependencies. Where task is a sequence of computations executed by single processor/core
- Static decomposition: program start/compile time
- Dynamic decomposition: during execution

- Goal
  - enough tasks to keep cores busy
  - tasks $\geq$ cores
  - Size of task >> overhead of parallelism

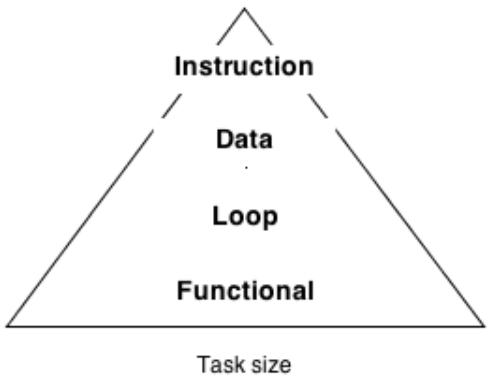
1.2.2 Scheduling

- Assignment of tasks to processes/threads (static/dynamic)
- Goal
  - Load balancing (of computations/memory access (shared mem.)/communications (distributed mem.)) among processes/threads
  - Efficient execution order

1.2.3 Mapping

- Assignment of processes/threads to execution units (cores/processors)
- Goal
  - Balance utilization of execution units
  - Minimize communications among processors

2 Levels of Parallelism



2.1 Instruction parallelism

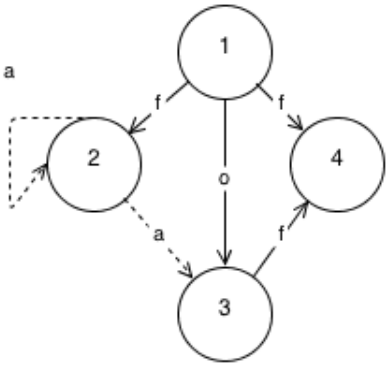
Multiple instruction, but data dependencies inhibits parallel execution

- **Flow dependency** ($W \rightarrow R$)
- **Sequential consistency**: see lecture 3: sequential consistency

Data dependency graph

1. x=a
2. y=y+x
3. x=z
4. b=x

f: flow dep
a: anti-dep
o: output dep



2.2 Data parallelism

Same operation used for different data (SIMD)

MIMD/SPMD means one program executed by all processors in parallel using shared/distibuted memory

2.3 Loop parallelism

When no dependency between iterations of loop

2.3.1 forall

Think: each assignment in its only loop, executed before next assignment is run

```
forall (i = 1:4)
    a(i) = a(i) + 1
    b(i) = a(i-1) + b(i+1)
```

becomes

```
for (i = 1:4)
    a(i) = a(i) + 1
for (i = 1:4)
    b(i) = a(i-1) + a(i+1)
```

2.3.2 dopar

Iterations executed in parallel. Updates in 1 iteration transparent to others.

2.3.3 doall

2.4 Functional/Task Parallelism

- Independent tasks can be executed in parallel

3 Parallel Programming Patterns

3.1 Fork-join

- Create children threads, which work in parallel, with `fork`
- (Parent) Thread waits for children using `join`

3.2 parbegin, parend

- Similar to fork/join?

3.3 SPMD, SIMD

- SIMD: single instruction executed sequentially by different threads on different data
- SPMD: Same program on different processors, but on different data. Threads work asynchronously with each other

3.4 Master-Slave/Worker

- Master assigns work to workers
- Master responsible for coordination, initialization, output etc (bottleneck)

3.5 Client-Server

- "Multiple program, multiple data"
- Server compute requests from multiple clients concurrently

3.6 Pipelining

- Data partitioned into stream of data elements flows through pipeline threads

3.7 Task pools

- Threads fixed (created statically)

3.8 Producer-Consumer

- Producer produce data used as input to consumer

Parallel Programming Models

1 Information Exchange

1.1 Shared address space

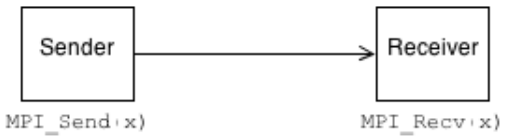
- Processes/Threads
- Syncronization ensure concurrent access to same variables are syncronized
- **Race conditions**: multiple threads accessing/modifing same shared variable (use locks)
- **Critical section**: a part where concurrent access to shared variables may occur (mutal exclusion - 1 thread at a time)

1.2 Distributed address space

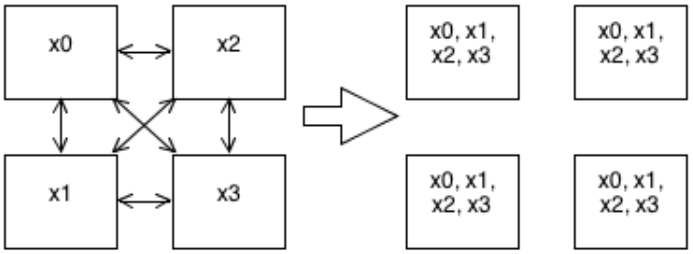
- Message passing: send/recv
- Types: point to point, collective

1.2.1 Communication Types

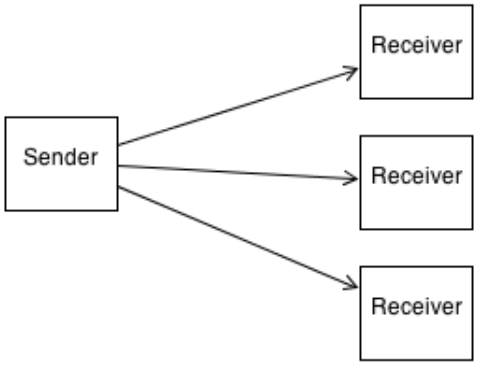
Point to point



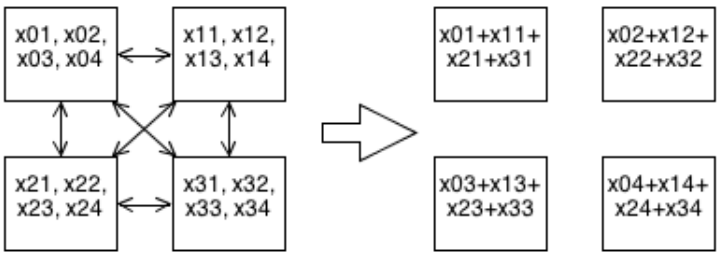
Multi broadcast



Single broadcast

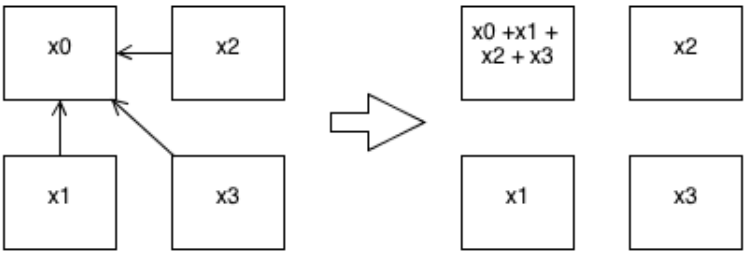


Multi accumulation

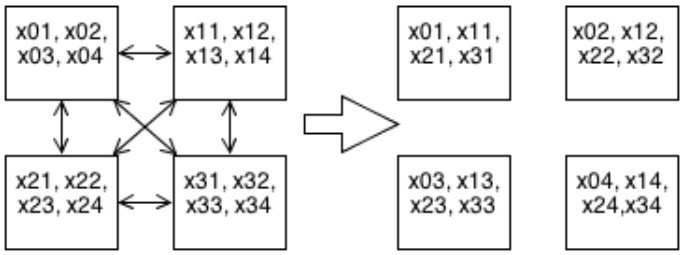


Single accumiulation

Reduction operation associative and commutative

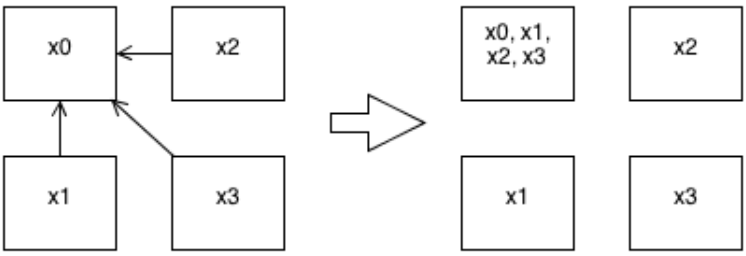


Total exchange

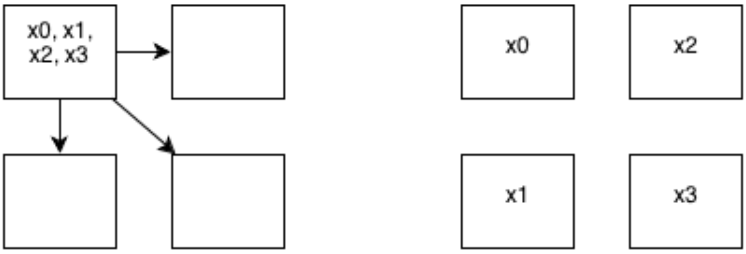


Gather

No reduction operation



Scatter

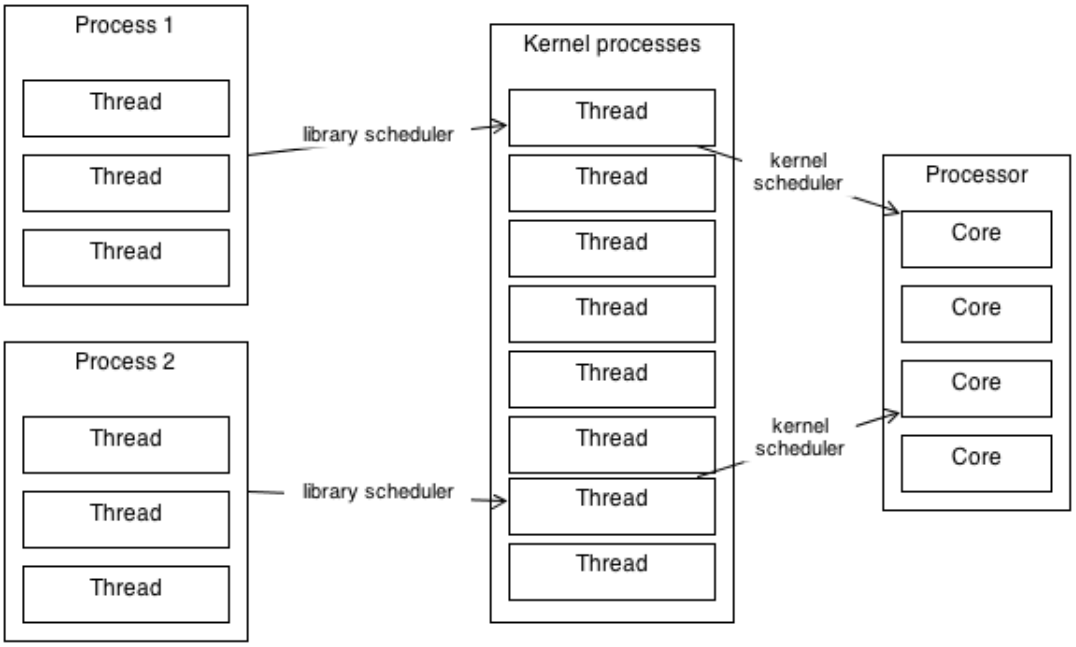


2 Abstractions of Control Flows

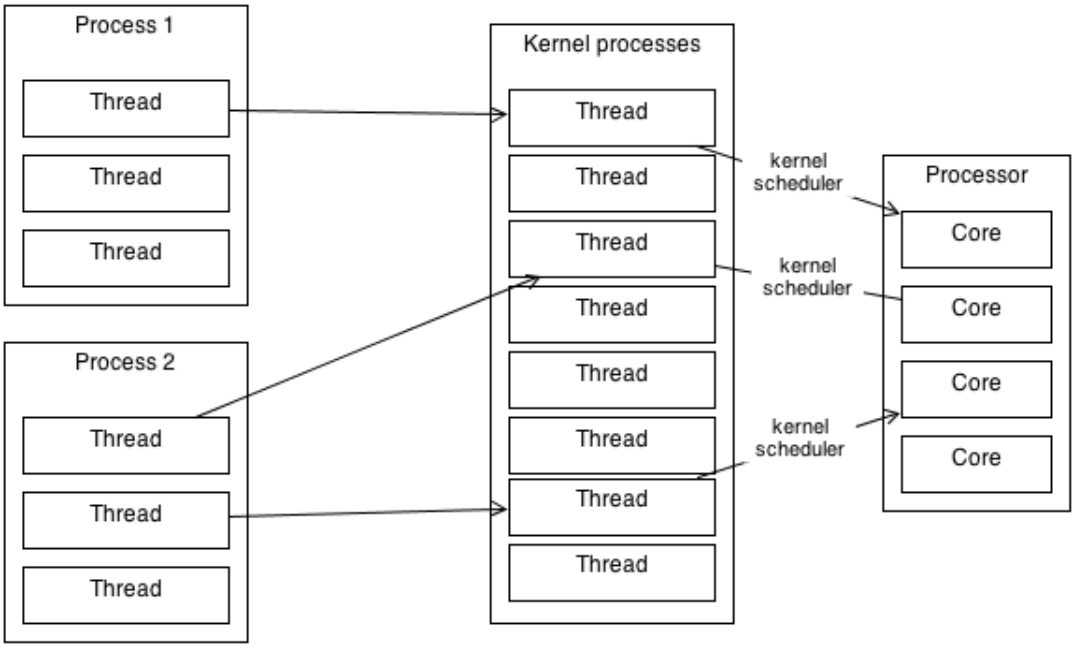
	Process	**Threads**
Address space	Own	Shared (with process)
Communication	Explicit	
Assigned to	Cores	Cores
Others	Need context switch (overhead) - round-robin, parallel execution. Fork: child gets copy of parents address space	User level threads are managed by thread library. OS cannot map user level threads of same process to different execution resources - no parallelism. OS cannot switch to another threads if 1 blocks. Advantage is switching is fast

2.1 Mapping from User to Kernel Level Threads

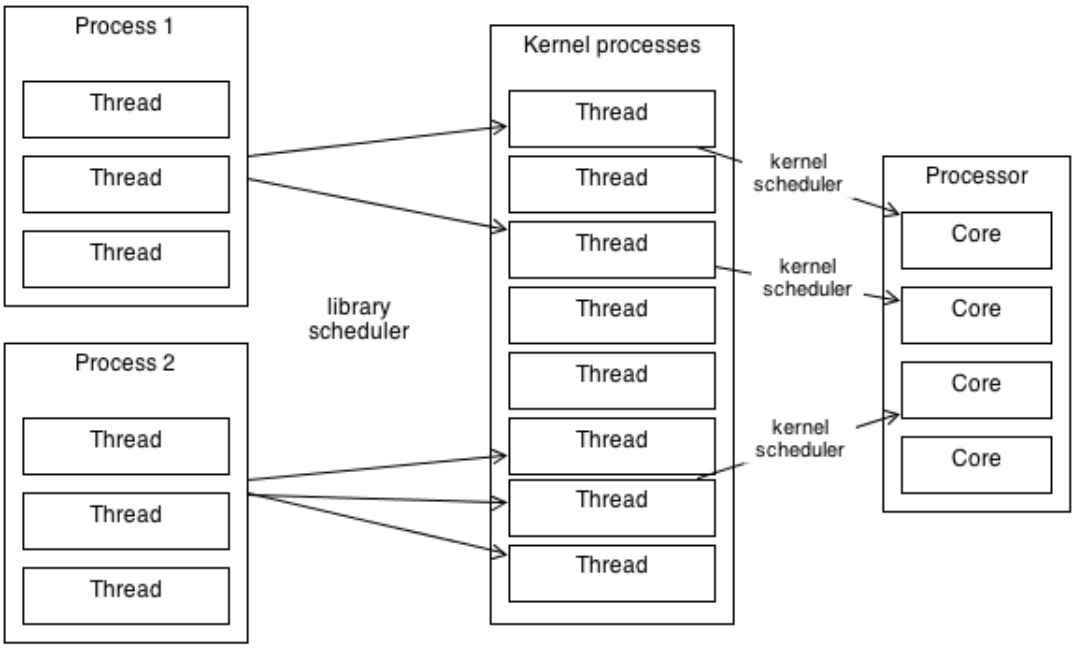
2.1.1 Many (threads) to One (process) Mapping



2.1.2 One (thread) to One (thread) Mapping



2.1.3 Many (thread) to Many (thread) Mapping



threads may be mapped to different kernel thread at different point in times

MPI

1 Message Passing Programming Model

- Distributed address space
- Processes exchange local data
- Point-to-Point or Global/Collective
- Explicit Parallelism - Programmer specify parallelism
- SPMD

1.1 Views

Local	**Blocking** - user allowed to refuse resources specified in call	**Non-Blocking** - procedure returns before operation completes, and before user is allowed to reuse resources specifed in call
Global	**Synchronous** - Operation does not complete until both processes started communication operation	**Asyncronous** - No coordination between parties

Parallel Algorithm Design

1 Overheads

- Cost of starting thread/process
- Communication
- Syncronization

In general, need for large granularity (units of works) to run fast (less overheads), but not so much there is too little parallel work

1.1 Communication

Tasks interact by sending messages through channels.
Channels are message queues connecting a task's output to another task's input

2 Foster's Design Methodology

- **Partitioning**: tasks
- **Communication**: provide data to tasks
- **Agglomeration**: Reduce communications
- **Mapping**: Map tasks to processors. Goal: minimize total time

2.1 Partitioning

- **Data**: divide data into pieces of approx. same size. Then determine how to associate tasks to data
- **Functional**: Divide computation into pieces. Determine how to associate data with these computations.

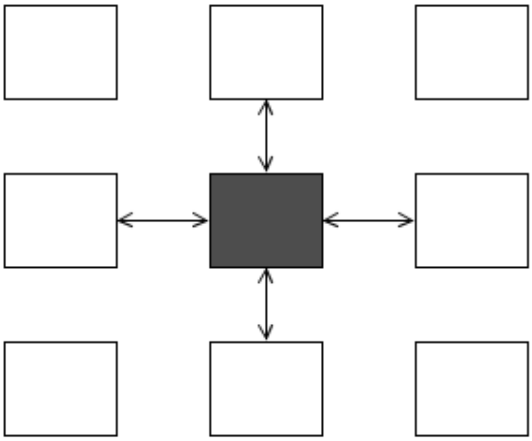
2.1.1 Goals

At least 10x primitive tasks	Ensure sufficient parallelism later
Minimize redundent computations and data	Complicates things as problem size increases
Primitive tasks roughly same size	Load balancing
Tasks should increase as problem size increases	ensure scalability

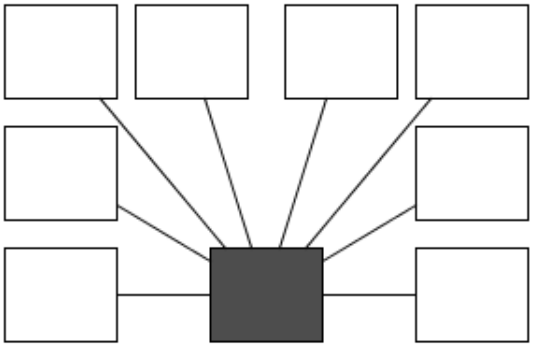
3 Communication

In general, communication required to pass data between tasks

- **Local**: Small amounts of data from neighbours
- **Global**: Significant number of tasks contribute data to perform computation



Local



Global

3.1 Goals

- Communication balanced between tasks (tree-like ($O(\log n)$))
- Task communicate with small number of neighbours
- Overlap communication
- Overlap computation

4 Agglomeration

- Combine tasks into larger tasks (reduce communication)
- Improve performance

- Maintain scalability
- Simplify programming
- 1 agglomerated task per processor

4.1 Goals

- Locality increased
- Number of tasks increase with task size

5 Mapping

- Assign task to processor
- Maximize pricessor utilization (tasks on different prorcessors)
- Minimize inter processor communication (assign tasks that communicate frequently to same processor)