Understanding Performance in Parallel Programming

Prof. Andrea Marongiu
andrea.marongiu@unimore.it

Includes slides from "*Multicore Programming Primer*" course at Massachusetts Institute of Technology (MIT) by Prof. Saman Amarasinghe and Dr. Rodric Rabbah
and from "*Parallel Programming for Multicore*" course at UC Berkeley, by prof Kathy Yelick

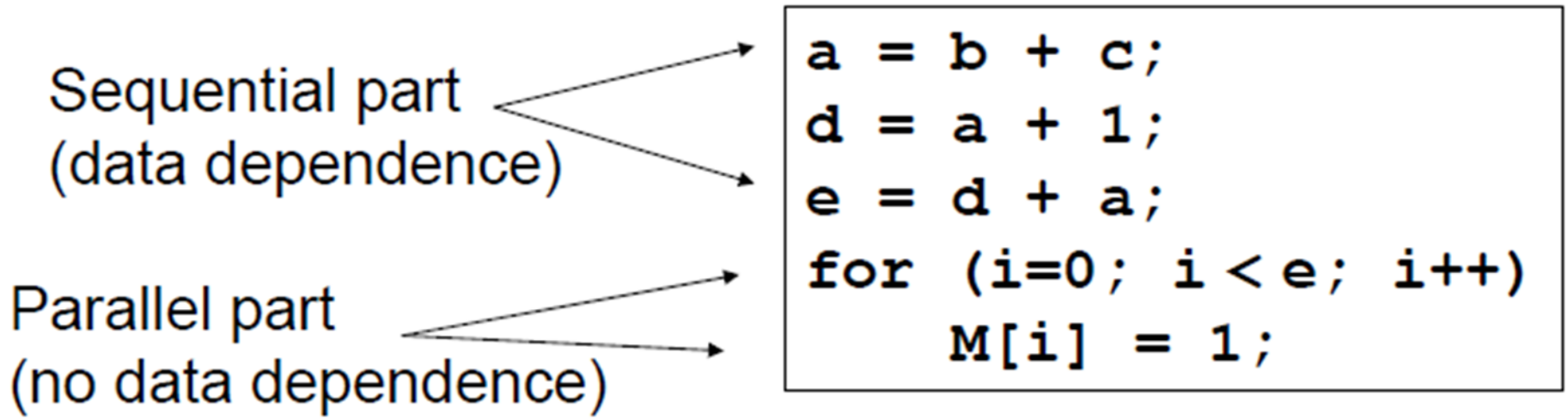
Understanding Performance

- What factors affect performance of parallel programs?
- **Coverage** or extent of parallelism in algorithm
- **Granularity** of partitioning among processors
- **Locality** of computation and communication

Coverage

- Not all programs are "embarassingly" parallel

- Programs have sequential parts and parallel parts



Challenges of parallel processing

Q1: can we get linear speedup?

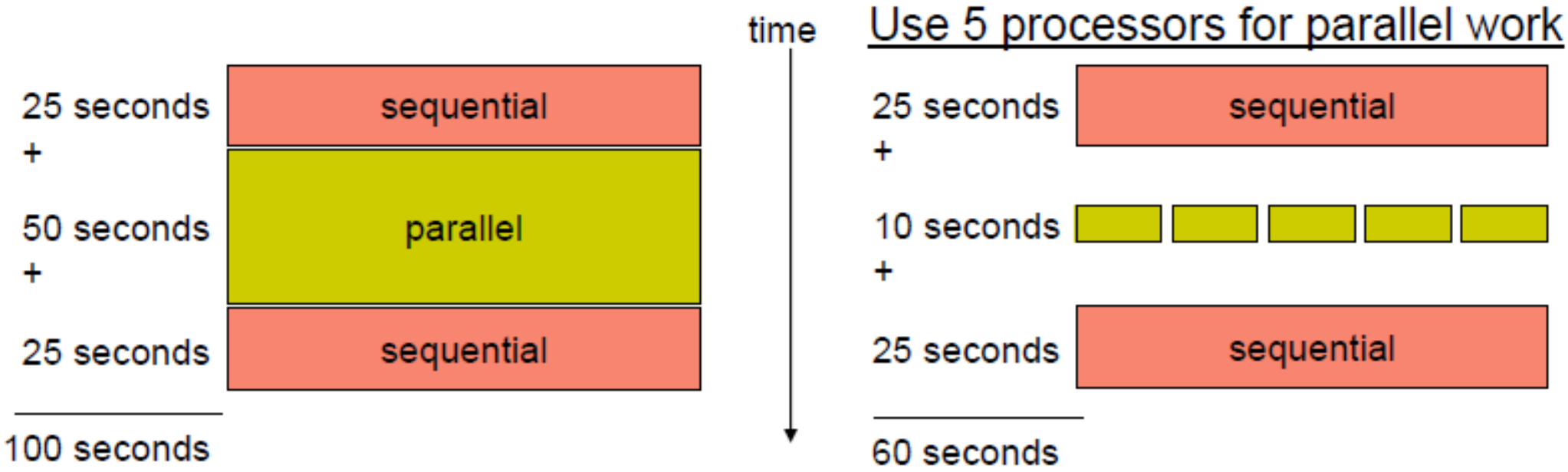
Suppose we want speedup 80 with 100 processors.
What fraction of the original computation can be sequential (i.e. non-parallel)?

Amdahl's Law

- **Amdahl's Law:** *The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used*

Amdahl's Law

- Potential program speedup is defined by the fraction of code that can be parallelized



Amdahl's Law

- Speedup = old running time / new running time
  = 100 seconds / 60 seconds
  = 1.67
  (parallel version is 1.67 times faster)

Amdahl's Law

- p = fraction of work that can be parallelized
- n = the number of processor

$$speedup = \frac{\text{old running time}}{\text{new running time}}$$

$$= \frac{1}{(1-p) + \frac{p}{n}}$$

fraction of time to complete sequential work

fraction of time to complete parallel work

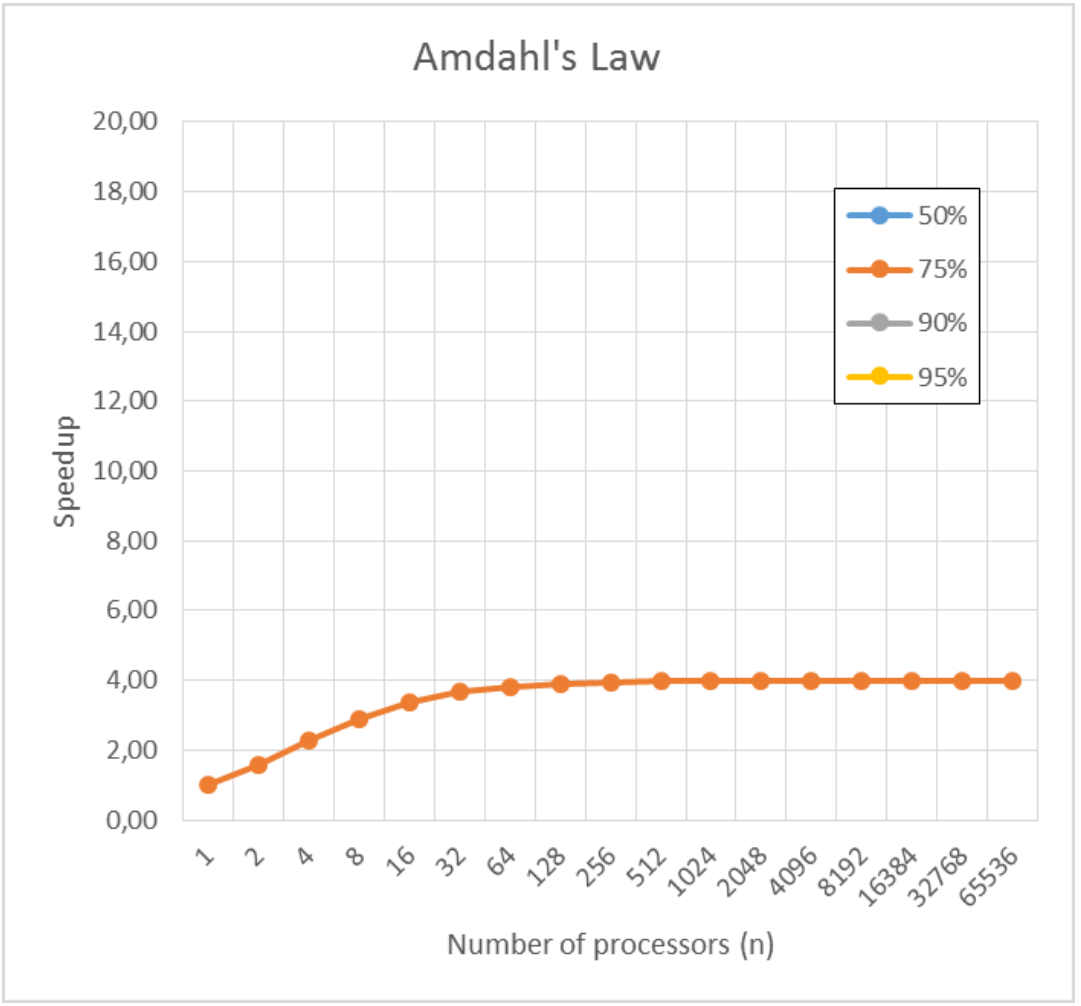
Amdahl's Law

$$S = \frac{1}{(1-p) + \frac{p}{n}}$$

fraction of time to complete sequential work

fraction of time to complete parallel work

- Suppose that 75% of a program can be parallelized.
- What is the speedup that could be achieved with 256 processors?
  - $S = \frac{1}{(1-0{,}75)+(\frac{0{,}75}{256})} = 3{,}95!$
- And for 65536 processors?
  - $S = \frac{1}{(1-0{,}75)+(\frac{0{,}75}{65536})} = \mathbf{4}!!!!!$



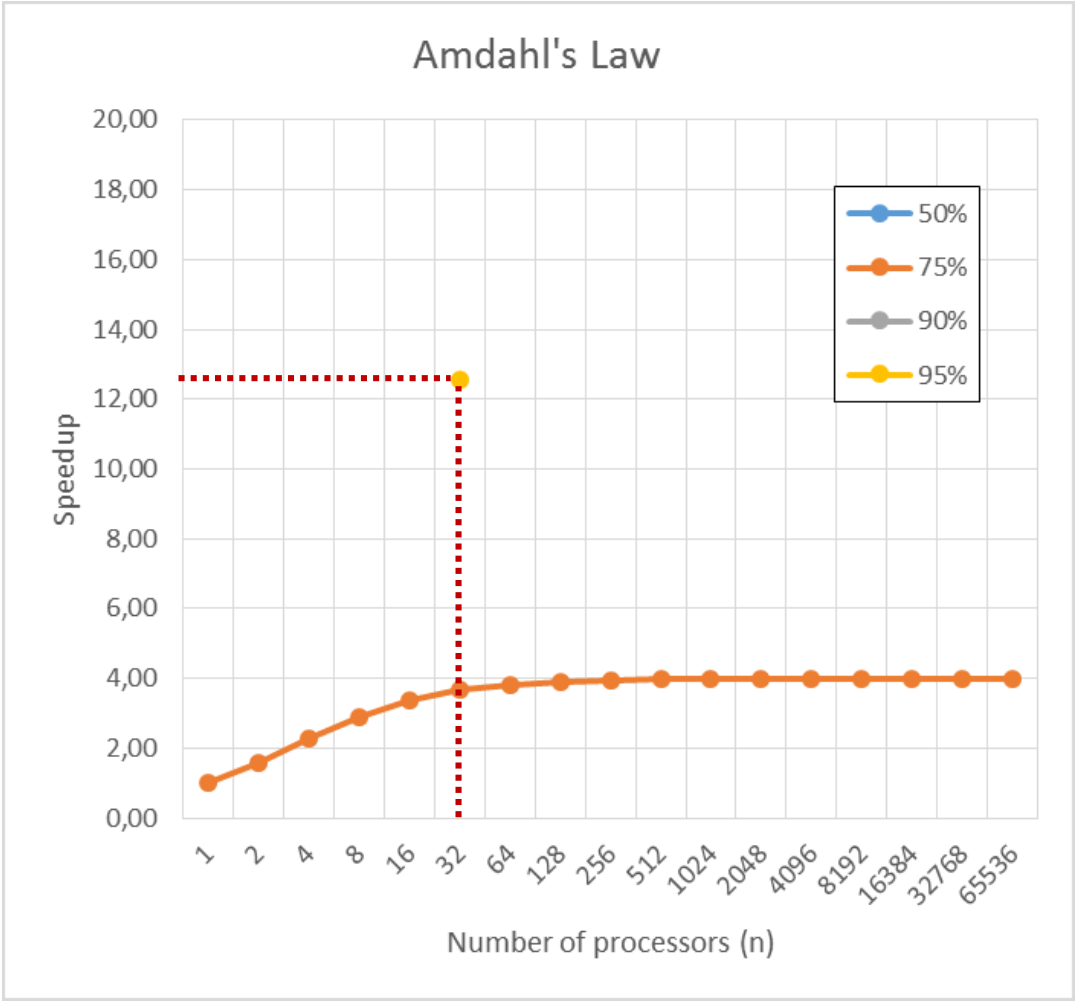
Amdahl's Law

$$S = \frac{1}{(1-p) + \frac{p}{n}}$$

fraction of time to complete sequential work

fraction of time to complete parallel work

- Suppose that n=32.
- Which p is required for S=12,55?
  - $\frac{1}{s} = 1 - p + \frac{p}{n}$ ➔
  - $\left(\frac{p}{n} - p\right) = \left(\frac{1}{s} - 1\right)$ ➔
  - $p \cdot \left(\frac{1}{n} - 1\right) = \left(\frac{1}{s} - 1\right)$ ➔
  - $p \cdot \left(\frac{1}{32} - 1\right) = \left(\frac{1}{12,55} - 1\right)$ ➔
  - $p = \frac{-11,55}{12,55} \cdot \frac{32}{-31} = \mathbf{95\%}$!!!!!



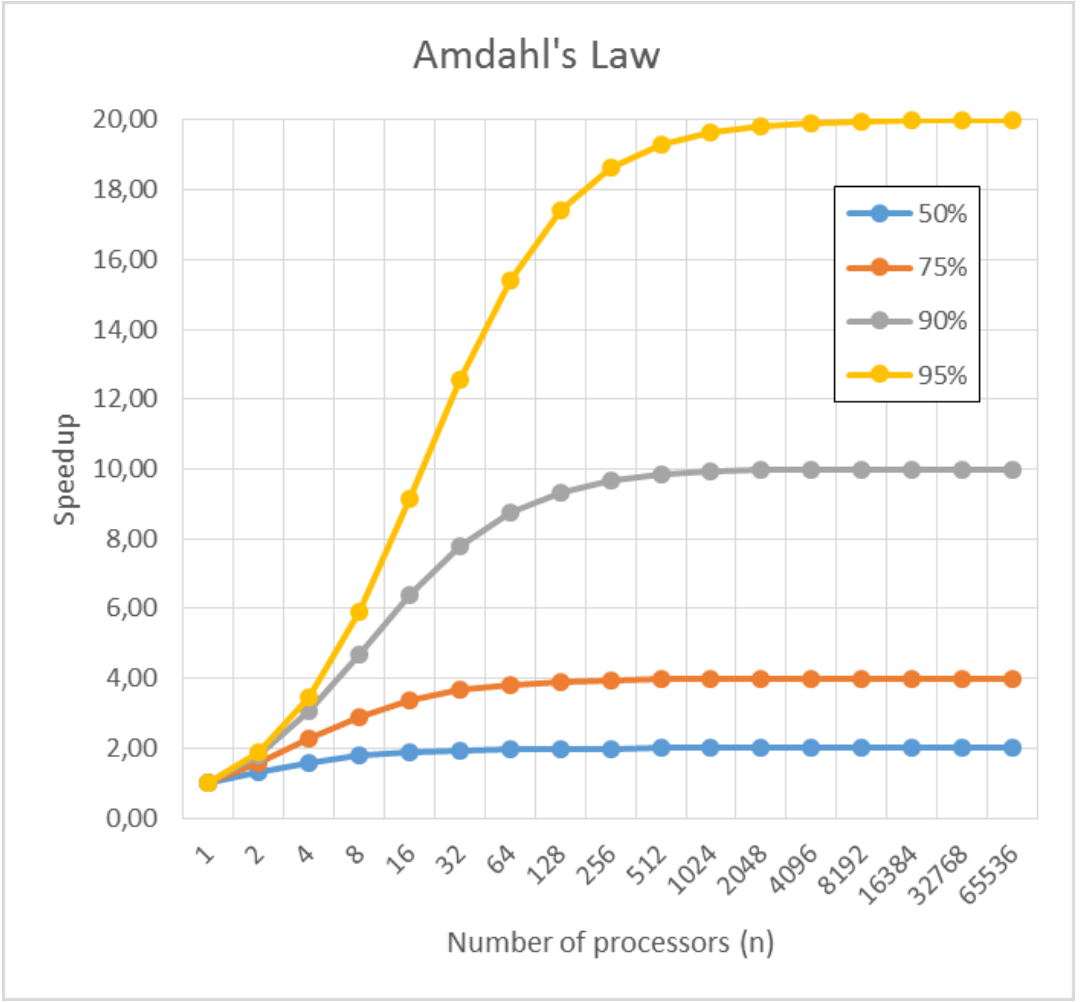
Amdahl's Law

$$S = \frac{1}{(1-p) + \frac{p}{n}}$$

fraction of time to complete sequential work

fraction of time to complete parallel work

- If the portion of the program that can be parallelized is small, then the speedup is limited
- The non-parallel portion limits the performance



Challenges of parallel processing

Q1: can we get linear speedup?

Suppose we want speedup 80 with 100 processors.
What fraction of the original computation can be sequential (i.e. non-parallel)?

Answer: fseq = 0.25%

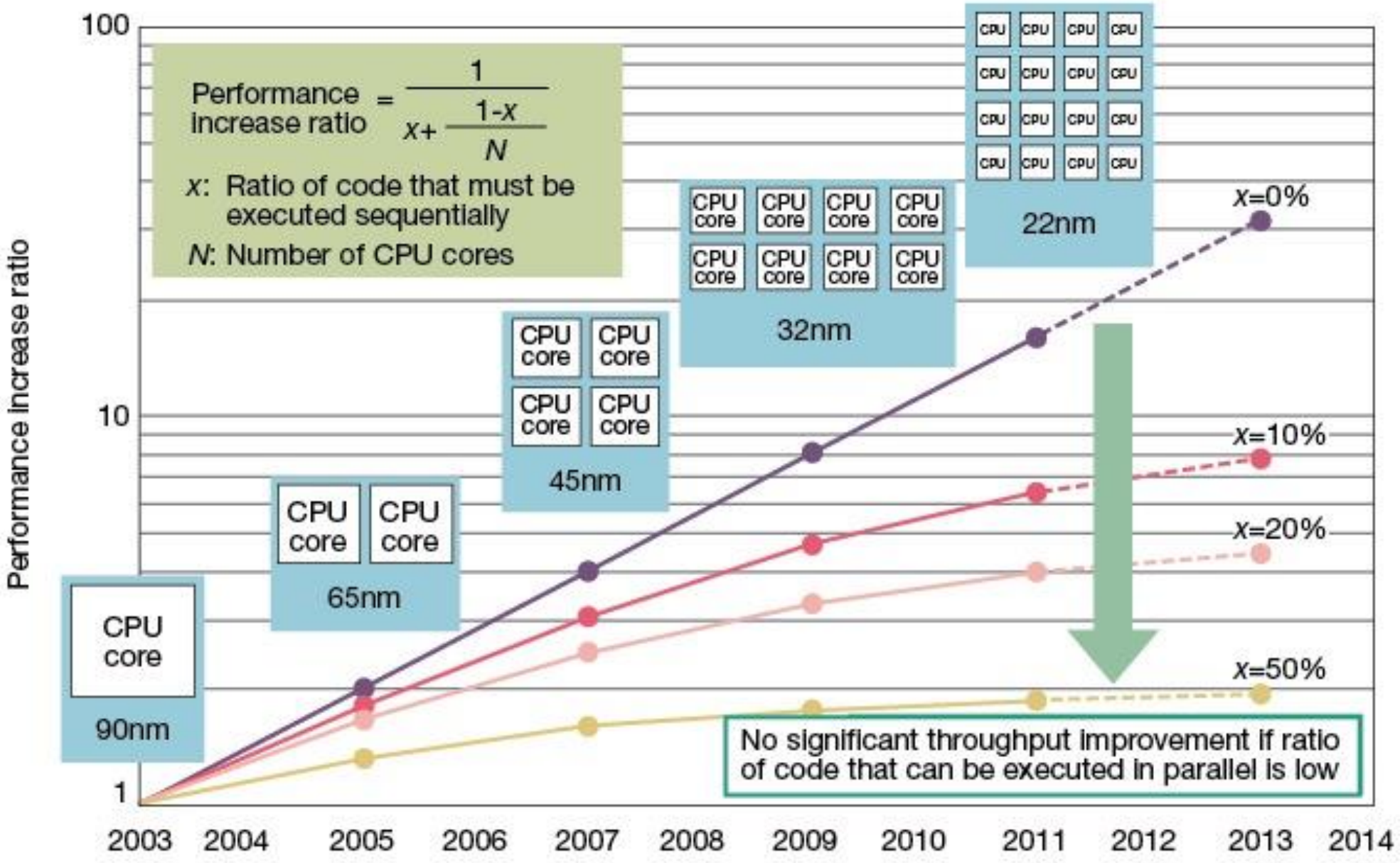
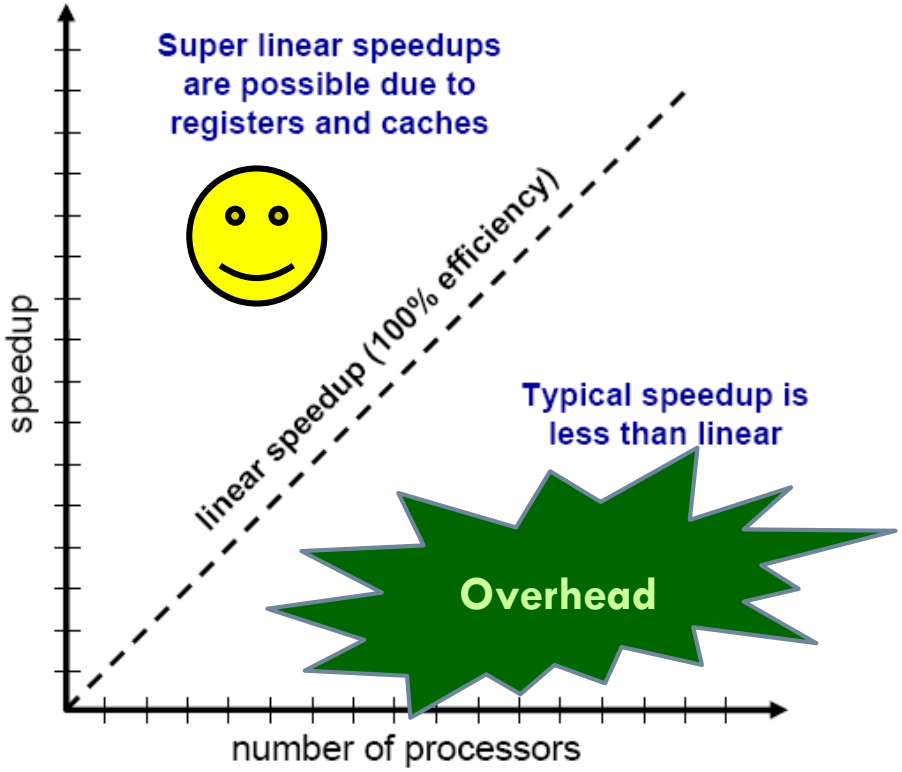


Fig 3 Amdahl's Law an Obstacle to Improved Performance Performance will not rise in the same proportion as the increase in CPU cores. Performance gains are limited by the ratio of software processing that must be executed sequentially. Amdahl's Law is a major obstacle in boosting multicore microprocessor performance. Diagram assumes no overhead in parallel processing. Years shown for design rules based on Intel planned and actual technology. Core count assumed to double for each rule generation.

Amdahl's Law

- Speedup tends to 1/(1-p) as number of processors tends to infinity
- Parallel programming is worthwhile when programs have a lot of work that is parallel in nature



Overhead of parallelism

- Given enough parallel work, this is the biggest barrier to getting desired speedup
- Parallelism overheads include:
  - cost of starting a thread or process
  - cost of communicating shared data
  - cost of synchronizing
  - extra (redundant) computation
    - Each of these can take a very large amount of time if a parallel program is not carefully designed
  - **Tradeoff:** Algorithm needs sufficiently large units of work to run fast in parallel (I.e. large **granularity**), but not so large that there is not enough parallel work

Strong and Weak Scaling

- To get good speedup on a multiprocessor while keeping the problem size fixed is harder than getting good speedup by increasing the size of the problem.
  - *Strong scaling*: when speedup can be achieved on a parallel processor without increasing the size of the problem
  - *Weak scaling*: when speedup is achieved on a parallel processor by increasing the size of the problem proportionally to the increase in the number of processors

Needed to amortize sources of **OVERHEAD** (*additional code, not present in the original sequential program, needed to execute the program in parallel*)

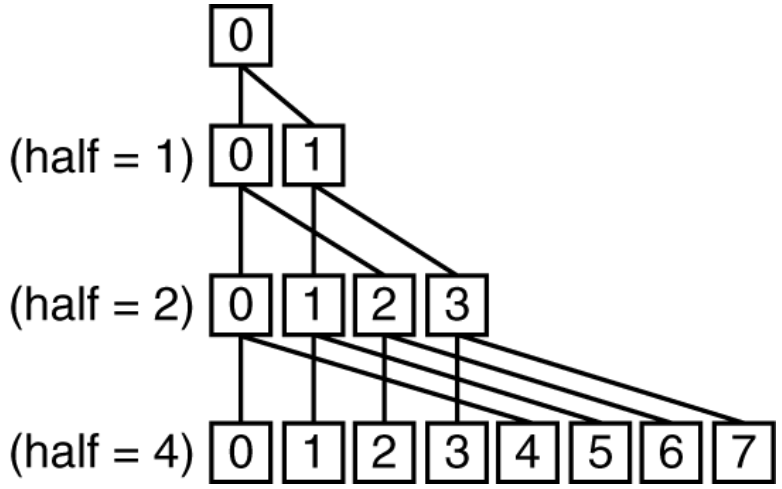
Example: Sum Reduction

- Sum 100,000 numbers on 100 processor SMP
  - Each processor has ID: 0 ≤ Pn ≤ 99
  - Partition 1000 numbers per processor
  - Initial summation on each processor

```
 sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i = i + 1)
  sum[Pn] = sum[Pn] + A[i];
```

- Now need to add these partial sums
  - Reduction: divide and conquer
  - Half the processors add pairs, then quarter, …
  - Need to synchronize between reduction steps

Example: Sum Reduction



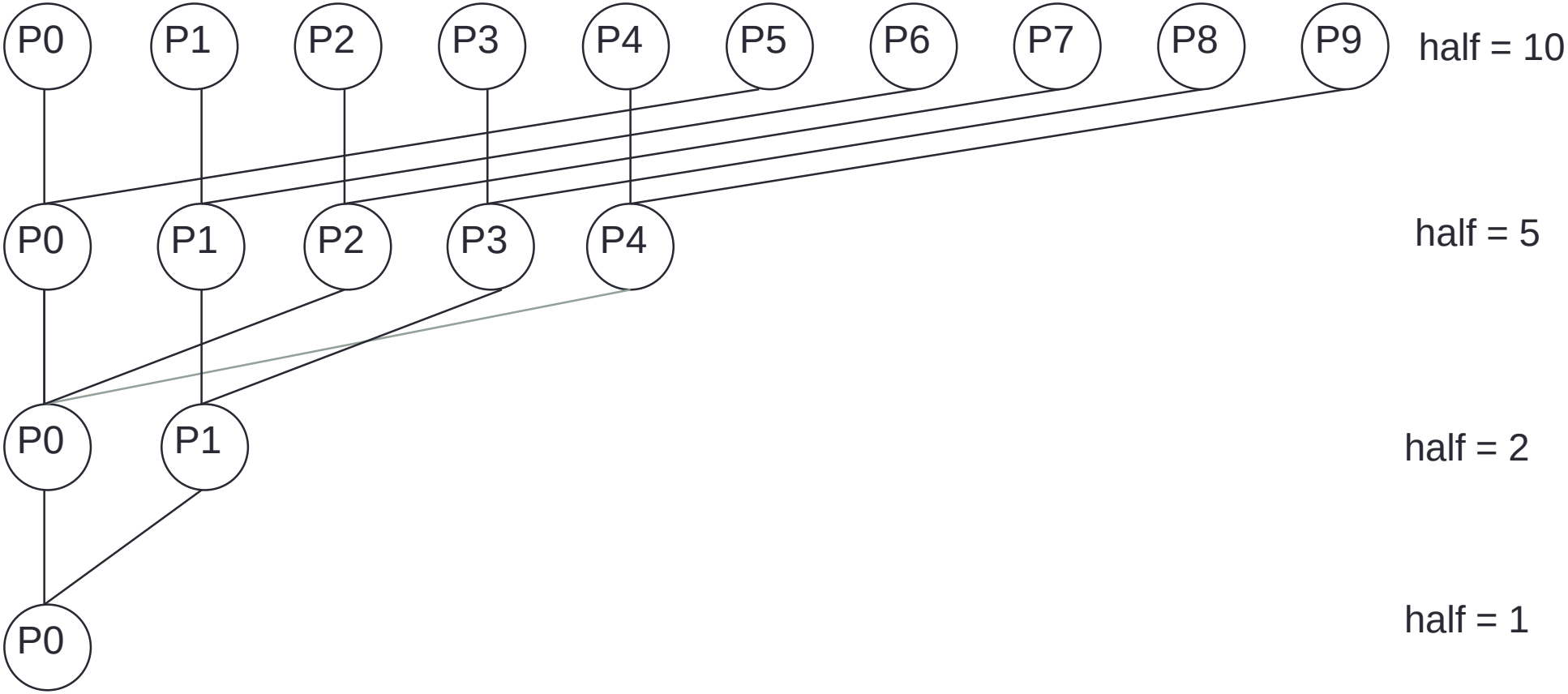

```
half = 100;
repeat
  synch();
  if (half%2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets missing element */
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```

An Example with 10 Processors

sum[P0] sum[P1] sum[P2] sum[P3] sum[P4] sum[P5] sum[P6] sum[P7] sum[P8] sum[P9]

An Example with 10 Processors

sum[P0] sum[P1] sum[P2] sum[P3] sum[P4] sum[P5] sum[P6] sum[P7] sum[P8] sum[P9]



Understanding Performance

- **Coverage** or extent of parallelism in algorithm
- **Granularity** of partitioning among processors
- **Locality** of computation and communication

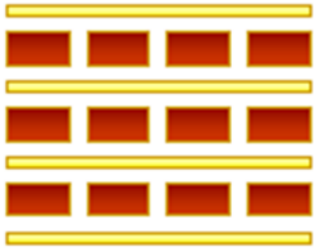
Granularity

Granularity is a (mostly qualitative) measure of the ratio between *useful work* and *additional work* implied by the parallel program:

- Computation to communication
- Communication to synchronization
  - Computation stages are typically separated from periods of communication by synchronization events
- Parallel work to parallelization overhead
- **Load balancing** is affected by granularity

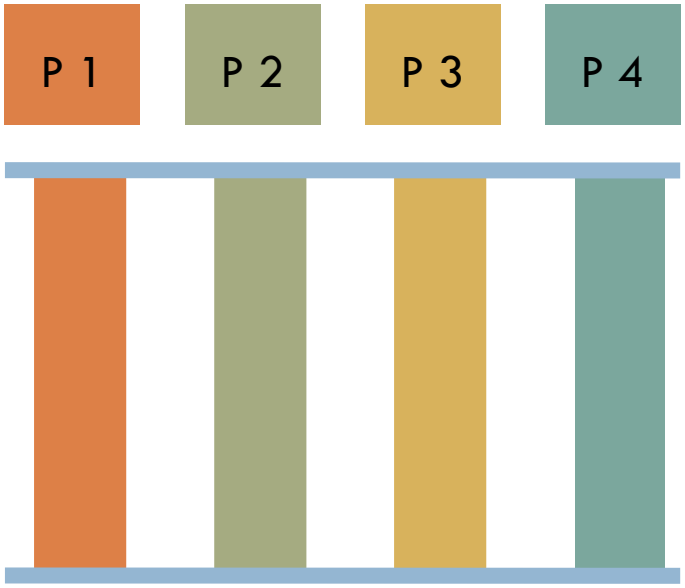
Granularity

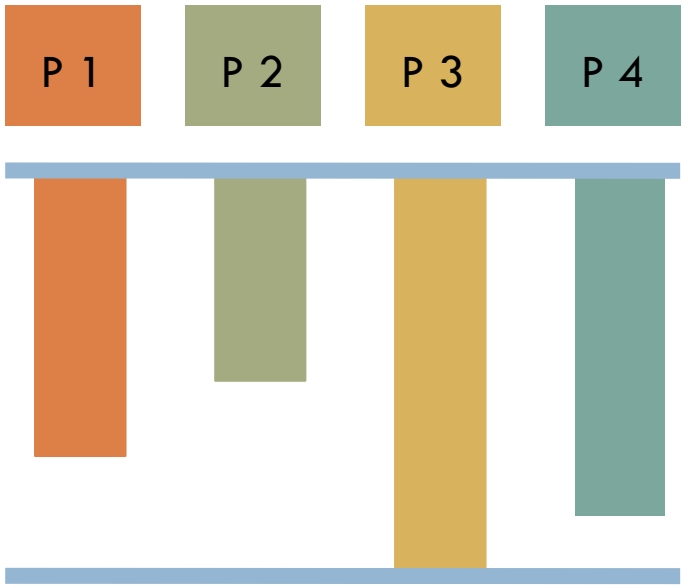
- Fine-grain Parallelism
  - Low *useful* to *additional* work ratio
  - Small amounts of *useful* work between *additional* work stages
  - High overhead
  - Better load balancing
- Coarse-grain Parallelism
  - High useful to *additional* work ratio
  - Large amounts of *useful* work between *additional* work stages
  - Overhead gets amortized
  - Harder to load balance efficiently



Load Balancing

- Processors that finish early have to wait for the processor with the largest amount of work to complete
  - Leads to idle time, lowers utilization
- Particularly urgent with barrier synchronization

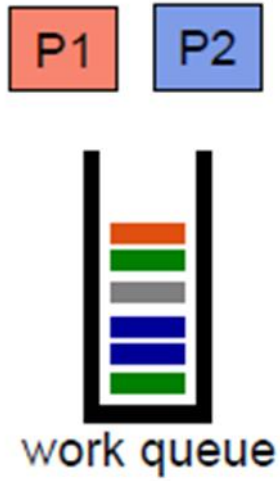




Slowest core dictates overall execution time

Static Load Balancing

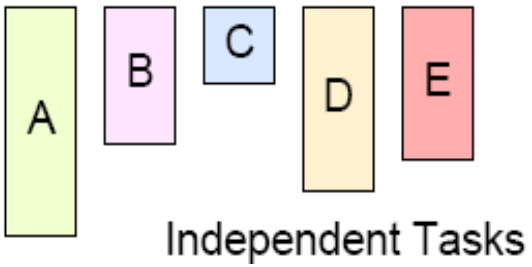
- Programmer make decisions and assigns a fixed amount of work to each processing core a priori

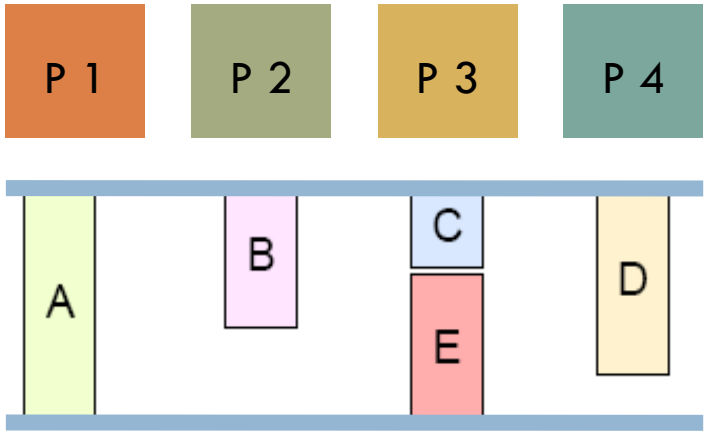


- Works well for homogeneous multicores
  - All core are the same
  - Each core has an equal amount of work
- Not so well for heterogeneous multicores
  - Some cores may be faster than others
  - Work distribution is uneven

Dynamic Load Balancing

- When one core finishes its allocated work, it takes on work from core with the heaviest workload
- Ideal for codes where work is uneven, and in heterogeneous multicore





Communication and Synchronization

- In parallel programming processors need to ***communicate*** partial results on data or ***synchronize*** for correct processing
- In **shared memory** systems
  - Communication takes place ***implicitly*** by concurrently operating on shared variables
  - Synchronization primitives must be ***explicitly*** inserted in the code
- In **distributed memory** systems
  - Communication primitives (send/receive) must be ***explicitly*** inserted in the code
  - Synchronization is ***implicitly*** achieved through message exchange

Challenges of parallel processing

Q2: how important is communication latency?

- Suppose 0.2 % of all accesses are remote, and require 100 cycles on a processor with base CPI = 0.5
- What's the communication impact?

Network: Performance metrics

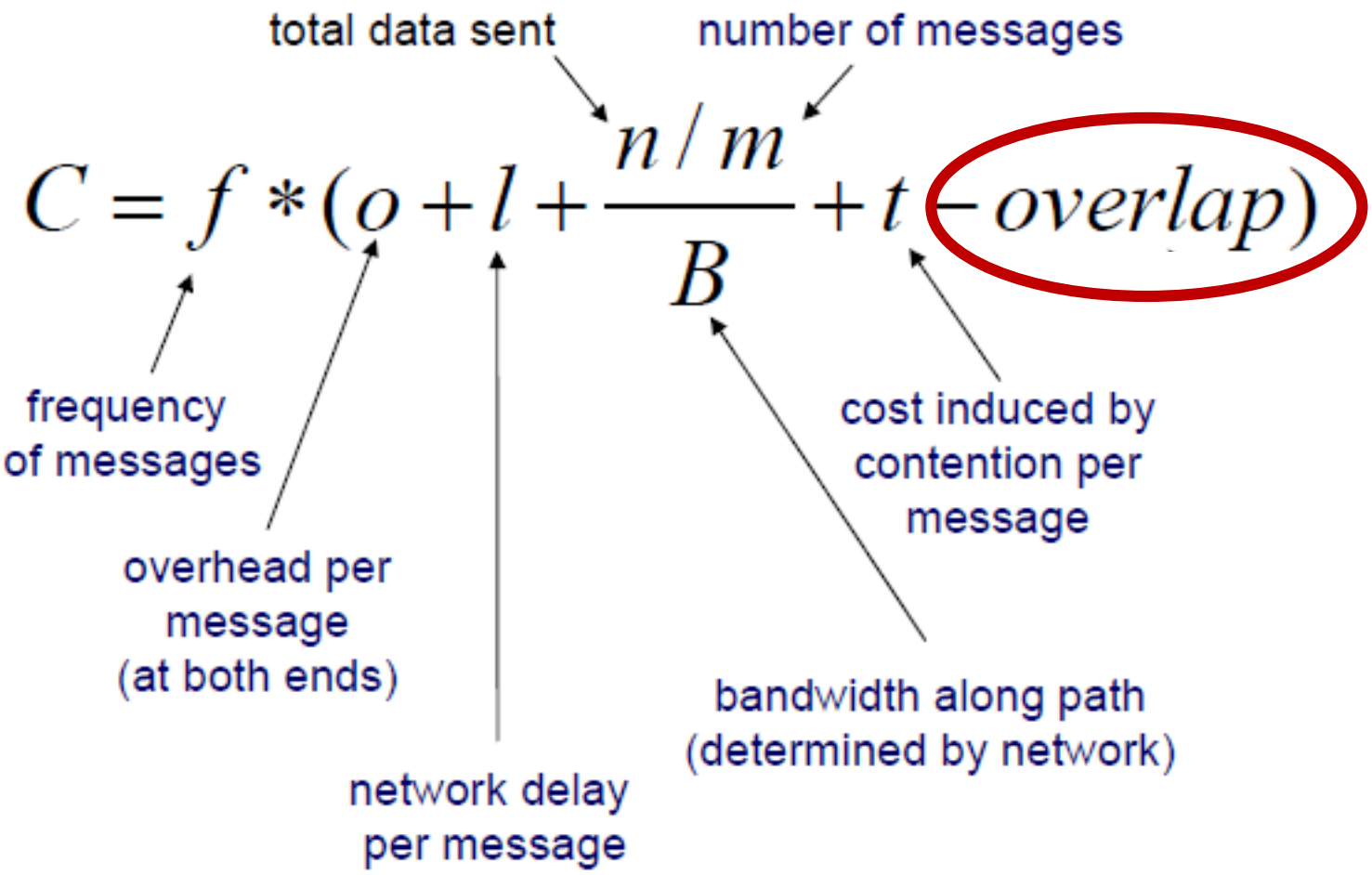
- Network Bandwidth
  - Need high bandwidth in communication
  - How does it scale with number of nodes?

- Communication Latency
  - Affects performance, since processor may have to wait
  - Affects ease of programming, since it requires more thought to overlap communication and computation

Network: Performance metrics

- How can a mechanism help hide latency?
  - overlap message send with computation,
  - prefetch data,
  - switch to other task or thread
  - pipelining tasks, or pipelining iterations

Communication Cost Model

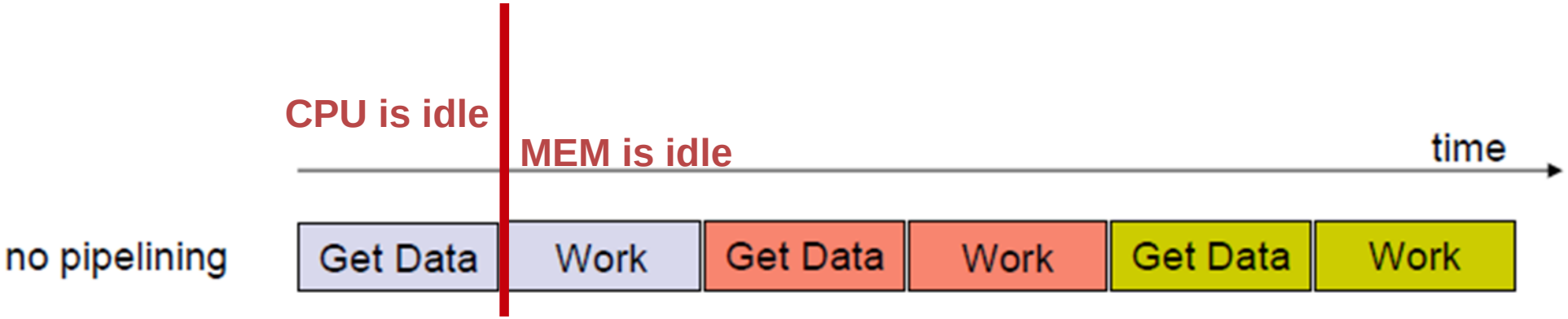


Types of Communication

- Cores exchange data or control messages
  - example: DMA
- Control messages are often short
- Data messages are relatively much larger

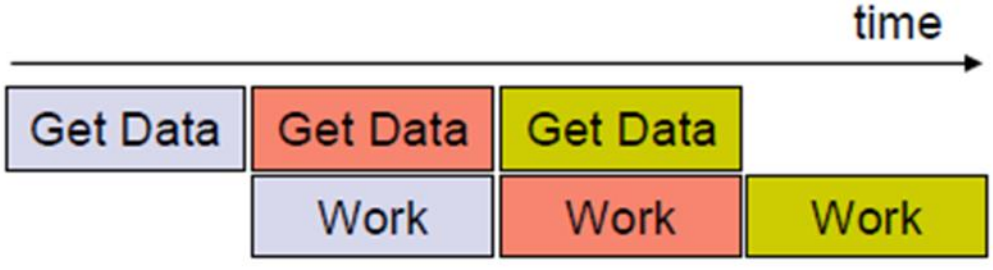
Overlapping messages and computation

- Computation and communication concurrency can be achieved with pipelining
  - Think instruction pipelining in superscalars



Overlapping messages and computation

- Computation and communication concurrency can be achieved with pipelining
  - Think instruction pipelining in superscalars
- Essential for performance on distributed memory multicores




```
// Start transfer for first buffer
id = 0;
mfc_get(buf[id], addr, BUFFER_SIZE, id, 0, 0);
id ^= 1;

while (!done) {
 // Start transfer for next buffer
 addr += BUFFER_SIZE;
 mfc_get(buf[id], addr, BUFFER_SIZE, id, 0, 0);

 // Wait until previous DMA request finishes
 id ^= 1;
 mfc_write_tag_mask(1 << id);
 mfc_read_tag_status_all();

 // Process buffer from previous iteration
 process_data(buf[id]);
}
```

Challenges of parallel processing

Q1: can we get linear speedup?

Insufficient parallelism

The problem of inadequate application parallelism must be attacked primarily in software with new algorithms that can have better parallel performance.

Q2: how important is communication latency?

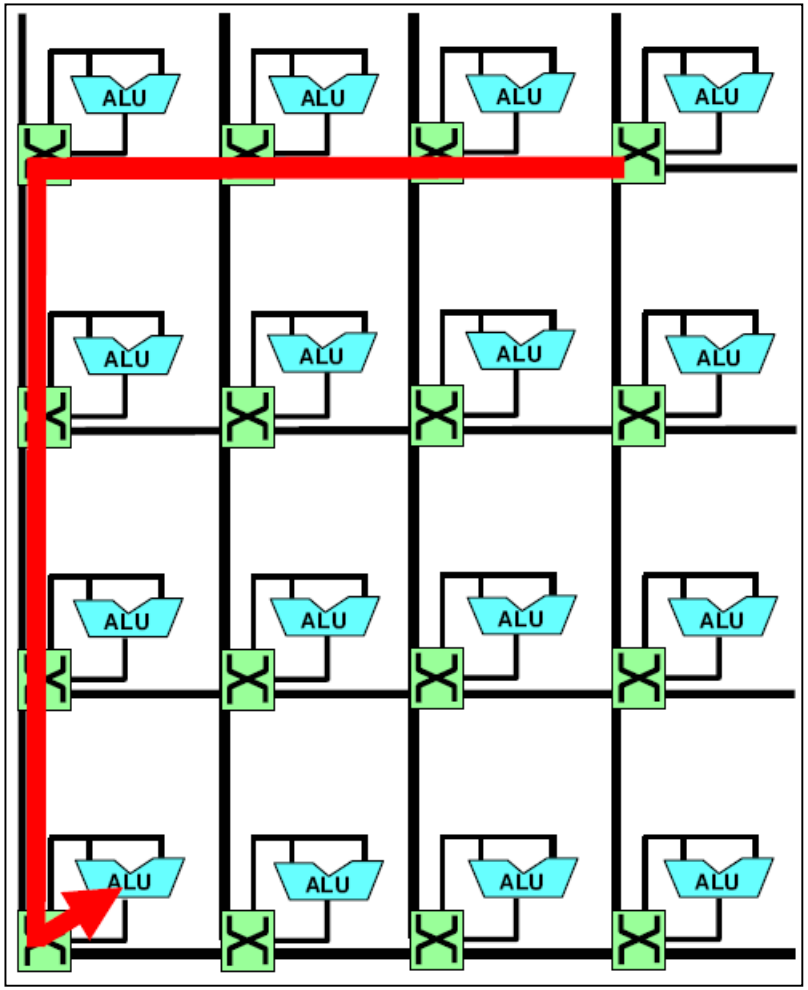
Long-latency remote communication

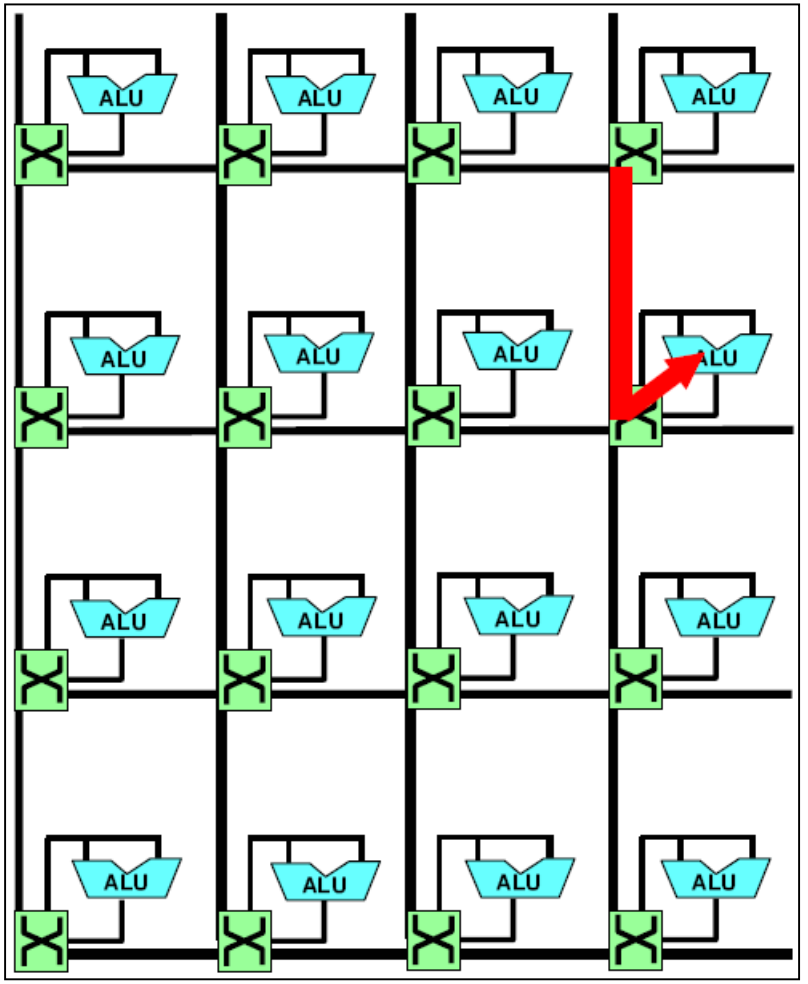
Reducing the impact of long remote latency can be attacked both by the architecture and by the programmer

Understanding Performance

- **Coverage** or extent of parallelism in algorithm
- **Granularity** of partitioning among processors
- **Locality** of computation and communication

Locality in communication (message passing)



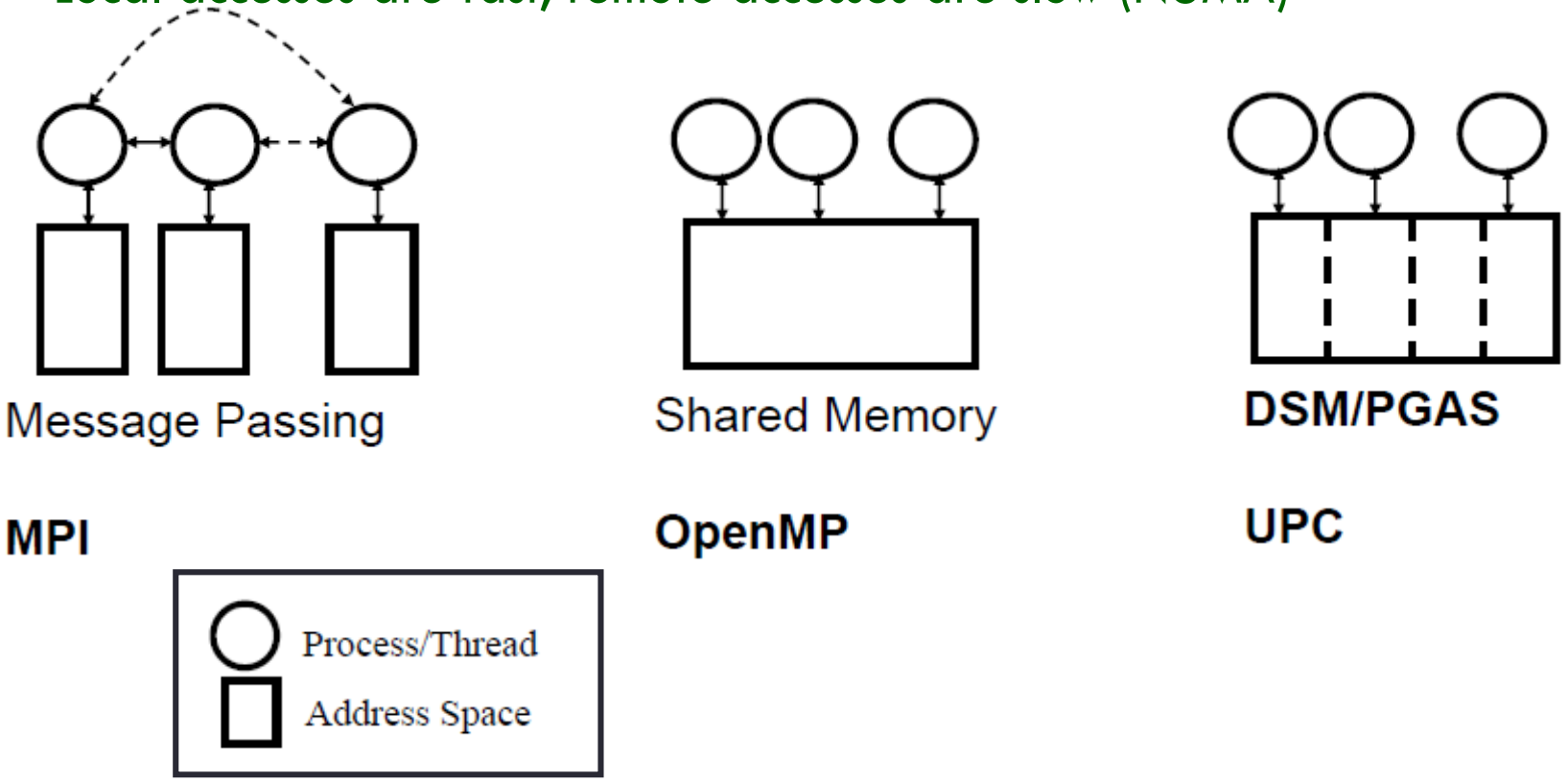


Memory access latency in shared memory architectures

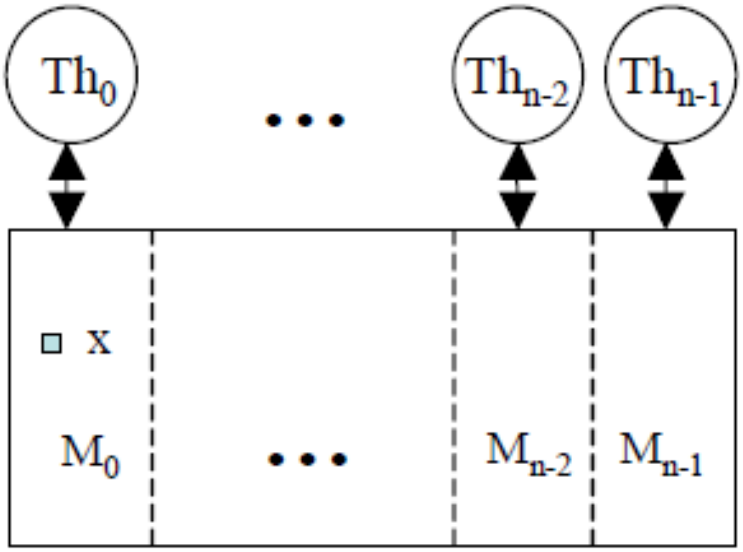
- Uniform Memory Access (UMA)
  - ▲ Centrally located memory
  - ▲ All processors are equidistant (access times)
- Non-Uniform Access (NUMA)
  - ▲ Physically partitioned but accessible by all
  - ▲ Processors have the same address space
  - ▲ Placement of data affects performance

Distributed Shared Memory

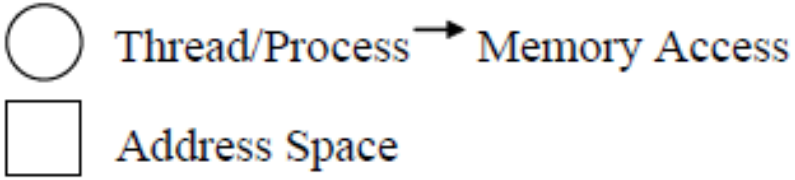
- A.k.a. **P**artitioned **G**lobal **A**ddress **S**pace (PGAS)
  - Each processor has a local memory node, globally visible by every processor in the system
  - Local accesses are fast, remote accesses are slow (NUMA)



Distributed Shared Memory



Legend



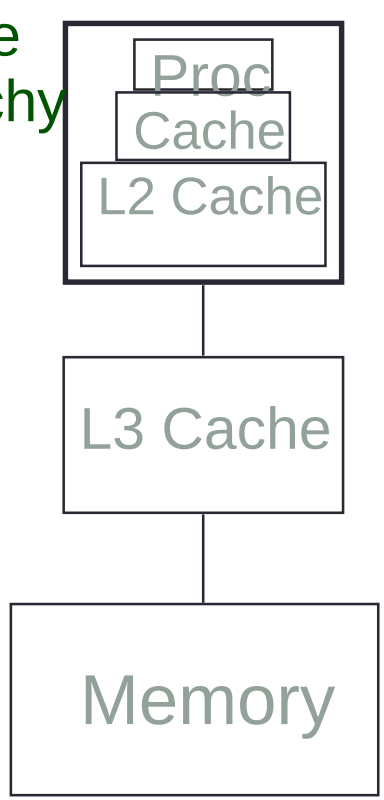
- Concurrent threads with a partitioned shared space
- Similar to the shared memory
- Memory partition Mi has affinity to thread Th_i
  - ▲ Helps exploiting locality
  - ▲ Simple statements as SM
  - ▼ Synchronization

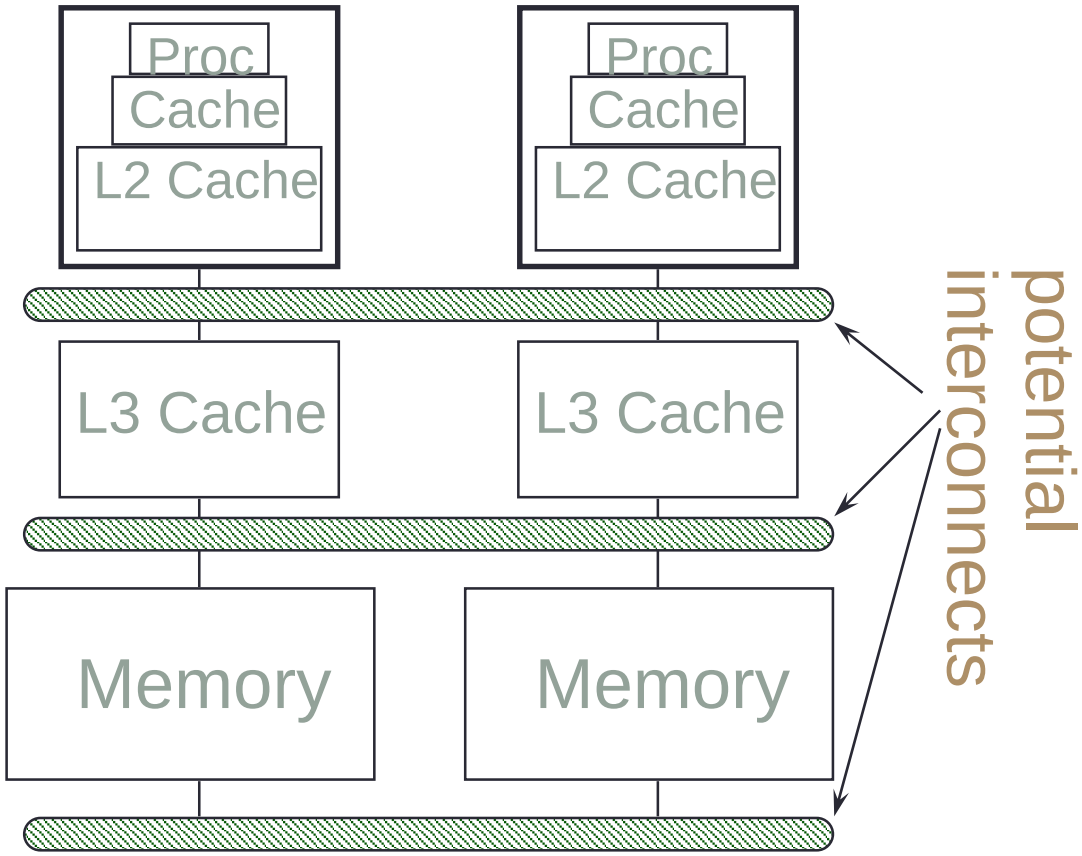
Locality – What does it mean?

- We talk about locality a lot
- What are the ways to improve memory behavior in a parallel application?
- What are the key considerations?
  - Mostly about managing caches (and registers)
  - Targets of optimizations
  - Abstractions to reason about locality
  - Data dependences, reordering transformations

Locality and Parallelism

Conventional Storage Hierarchy

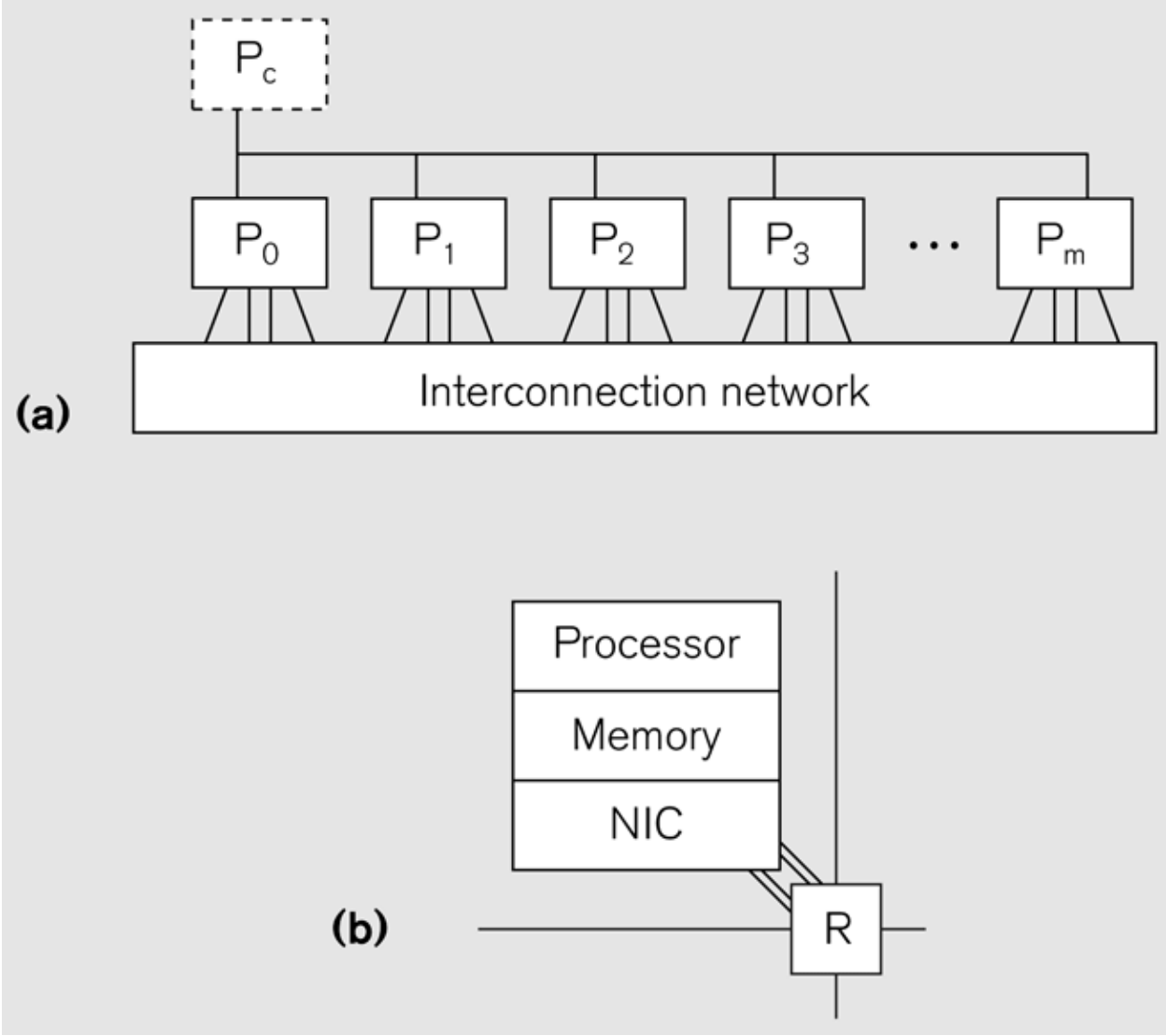




- Large memories are slow, fast memories are small
- Cache hierarchies are intended to provide illusion of large, fast memory
- Program should do most work on local data!

Interconnect

- A model with *P* standard processors, *d* degree,*λ* latency
- Node == processor + memory + NIC
- Key Property: Local memory ref is 1, global memory is λ



Targets of Memory Hierarchy Optimizations

- Reduce ***memory latency***
  - The latency of a memory access is the time (usually in cycles) between a memory request and its completion
- Maximize ***memory bandwidth***
  - Bandwidth is the amount of useful data that can be retrieved over a time interval
- Manage overhead
  - Cost of performing optimization (e.g., copying) should be less than anticipated gain

Reuse and Locality

- Consider how data is accessed
  - ***Data reuse:***
    - Same or nearby data used multiple times
    - Intrinsic in computation
  - ***Data locality:***
    - Data is reused and is present in “fast memory”
    - Same data or same data transfer
  - If a computation has reuse, what can we do to get locality?
    - Appropriate data placement and layout
    - Code reordering transformations

Cache basics

- **Cache hit:**
  - in-cache memory access—cheap
- **Cache miss:**
  - non-cached memory access—expensive
  - need to access next, slower level of hierarchy
- **Cache line size:**
  - # of bytes loaded together in one entry
  - typically a few machine words per entry
- **Capacity:**
  - amount of data that can be simultaneously in cache
- **Associativity**
  - direct-mapped: only 1 address (line) in a given range in cache
  - *n*-way: $n \geq 2$ lines w/ different addresses can be stored

Temporal Reuse in Sequential Code

- Same data used in distinct iterations I and I'

```
for (i=1; i<N; i++)
  for (j=1; j<N; j++)
    A[j]= A[j+1]+A[j-1]
```

- `A[j]` has self-temporal reuse in loop `i`

Spatial Reuse

- Same data transfer (usually cache line) used in distinct iterations I and I'

```
for (i=1; i<N; i++)
  for (j=1; j<N; j++)
    A[j]= A[j+1]+A[j-1];
```

- A[j] has self-spatial reuse in loop j
- **Multi-dimensional array note:** C arrays are stored in row-major order

Can Use Reordering Transformations!

- Analyze reuse in computation
- Apply loop reordering transformations to improve locality based on reuse
- With any loop reordering transformation, always ask
  - **Safety?** (doesn't reverse dependences)
  - **Profitablity?** (improves locality)

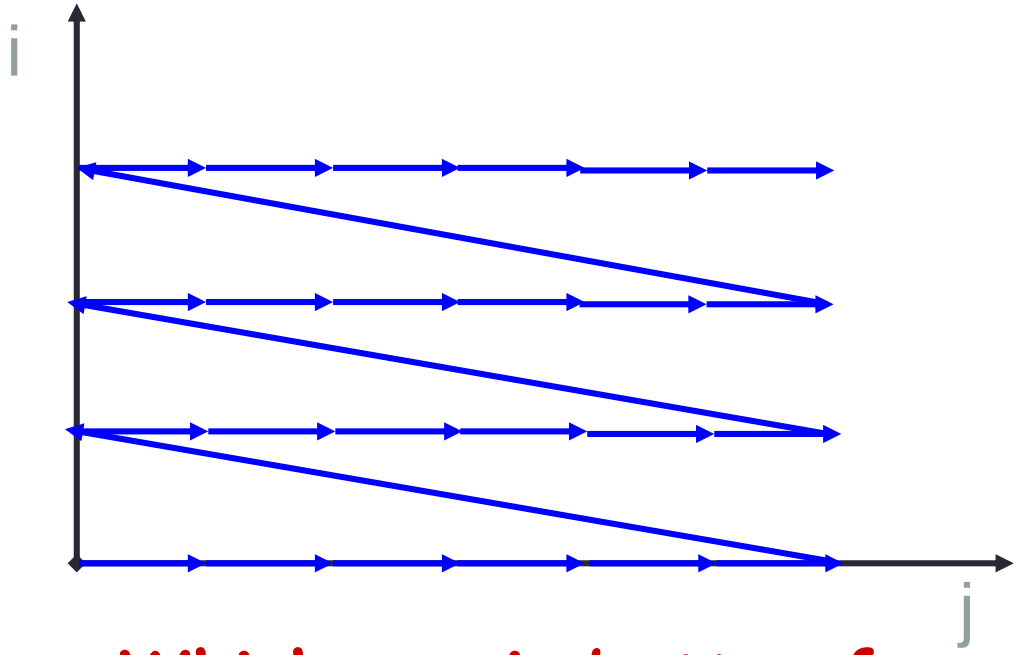
Loop Permutation:

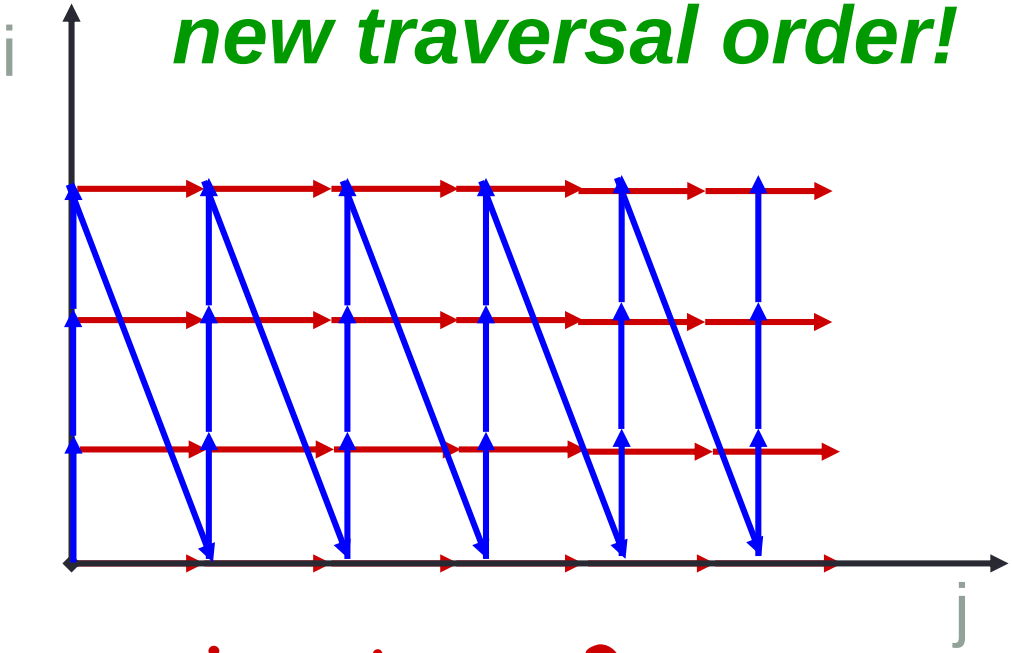
An Example of a Reordering Transformation

Permute the order of the loops to modify the traversal order

```
for (i= 0; i<3; i++)
 for (j=0; j<6; j++)
   A[i][j+1]=A[i][j]+B[j];
```

```
for (j=0; j<6; j++)
 for (i= 0; i<3; i++)
   A[i][j+1]=A[i][j]+B[j];
```





new traversal order!

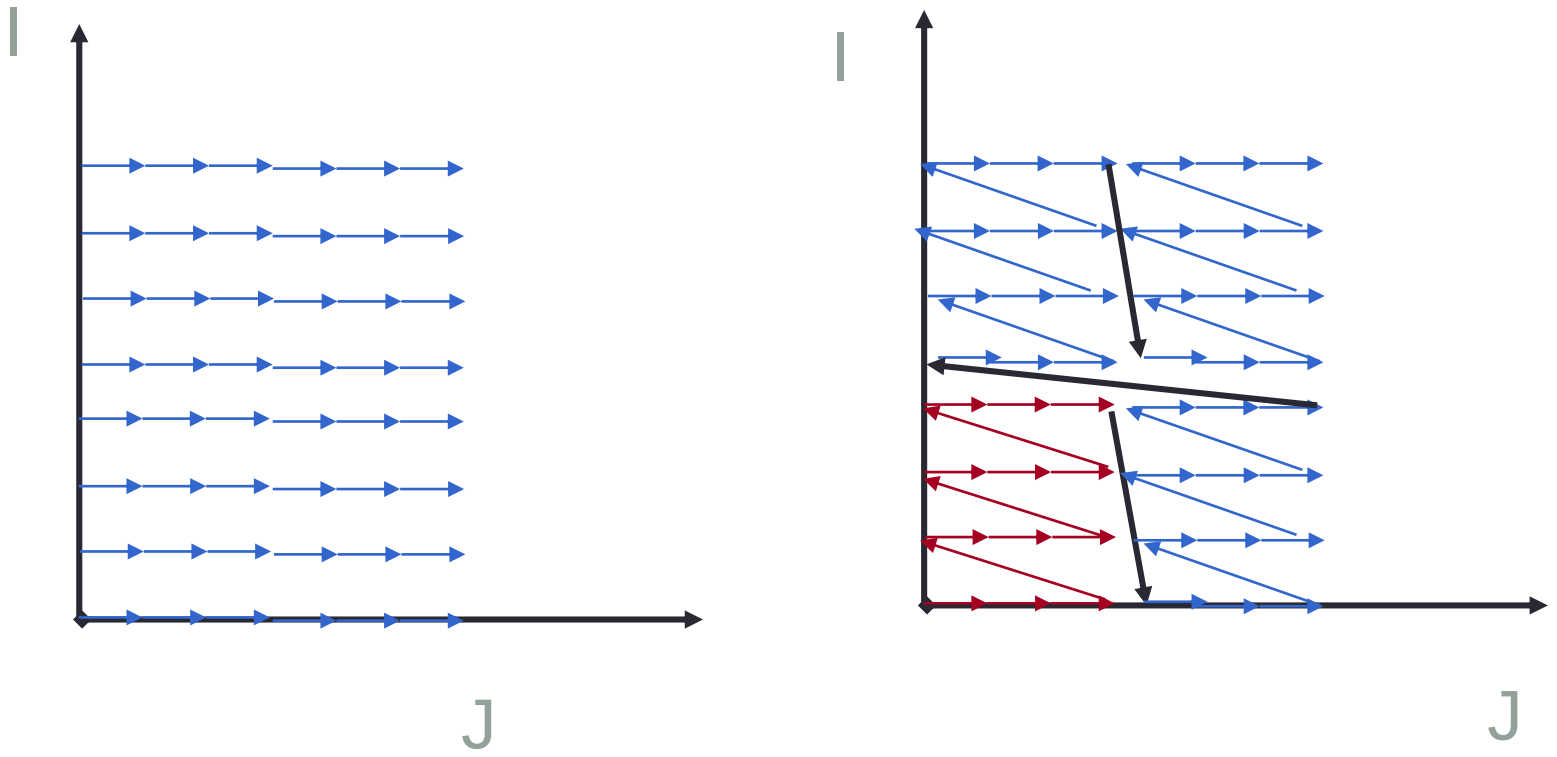
Which one is better for row-major storage?

Permutation has many goals

- Locality optimization
  - Particularly, for spatial locality (like in your SIMD assignment)
- Rearrange loop nest to move parallelism to appropriate level of granularity
  - Inward to exploit fine-grain parallelism (like in your SIMD assignment)
  - Outward to exploit coarse-grain parallelism
- Also, to enable other optimizations

Tiling (Blocking):
Another Loop Reordering Transformation

- Blocking reorders loop iterations to bring iterations that reuse data closer in time
- Goal is to retain in cache between reuse



Tiling is Fundamental!

- Tiling is very commonly used to manage limited storage
  - Registers
  - Caches
  - Software-managed buffers
  - Small main memory
- Can be applied hierarchically

Tiling Example

```
for (j=1; j<M; j++)
   for (i=1; i<N; i++)
     D[i] = D[i] +B[j,i]
```

Strip mine

```
for (j=1; j<M; j++)
   for (i=1; i<N; i+=s)
       for (ii=i; ii<min(i+s-1,N); ii++)
        D[ii] = D[ii] +B[j,ii]
```

Permute

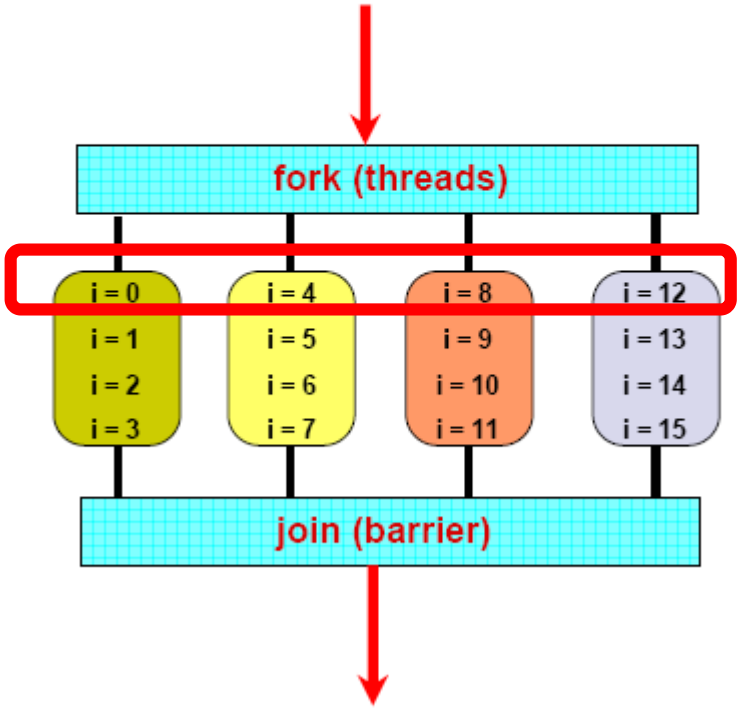
```
for (i=1; i<N; i+=s)
   for (j=1; j<M; j++)
      for (ii=i; ii<min(i+s-1,N); ii++)
        D[ii] = D[ii] +B[j,ii]
```

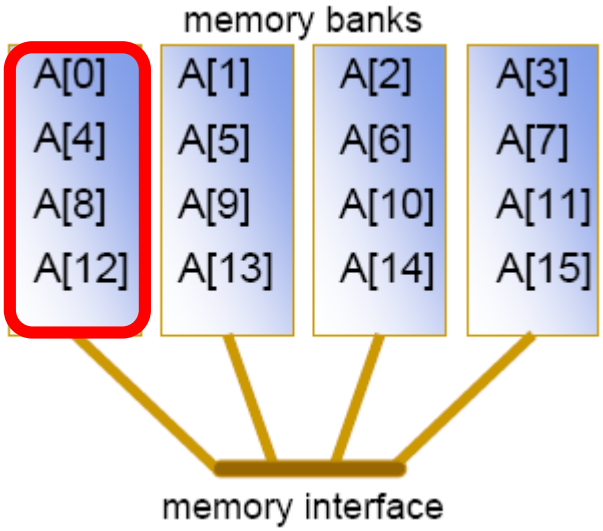
How to Determine Safety and Profitability?

- Safety
  - Notion of reordering transformations
  - Based on data dependences
- Profitability
  - Reuse analysis (and other cost models)
  - Also based on data dependences, but simpler

Locality of memory accesses (shared memory)

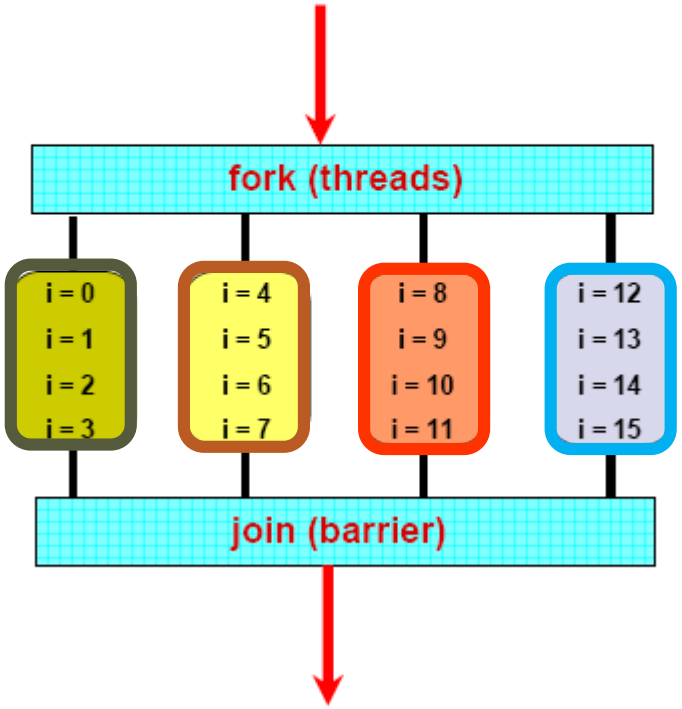
```
for (i = 0; i < 16; i++)
      C[i] = A[i] + ...;
```

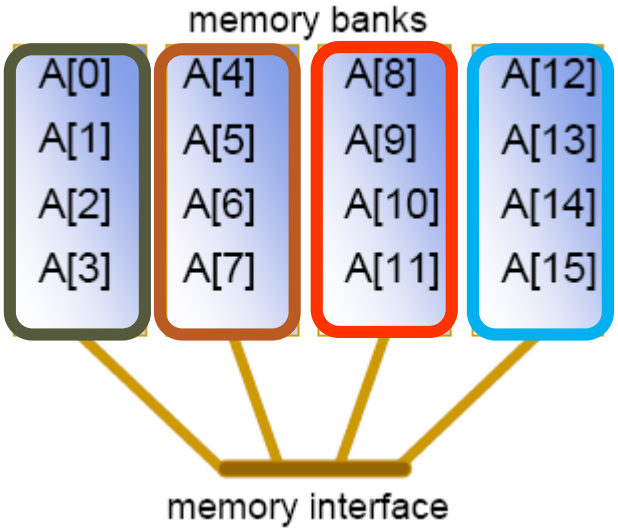




Locality of memory accesses (shared memory)

```
for (i = 0; i < 16; i++)
      C[i] = A[i] + ...;
```





Computing systems performance

Response Time and Throughput

- Response time
  - How long it takes to do a task
- Throughput
  - Total work done per unit time
    - e.g., tasks/transactions/… per hour
- How are response time and throughput affected by
  - Replacing the processor with a faster version?
  - Adding more processors?
- We'll focus on response time for now…

Relative Performance

- Define Performance = 1/Execution Time
- "X is n time faster than Y"

$$\text{Performance}_X / \text{Performance}_Y$$
$$= \text{Execution time}_Y / \text{Execution time}_X = n$$

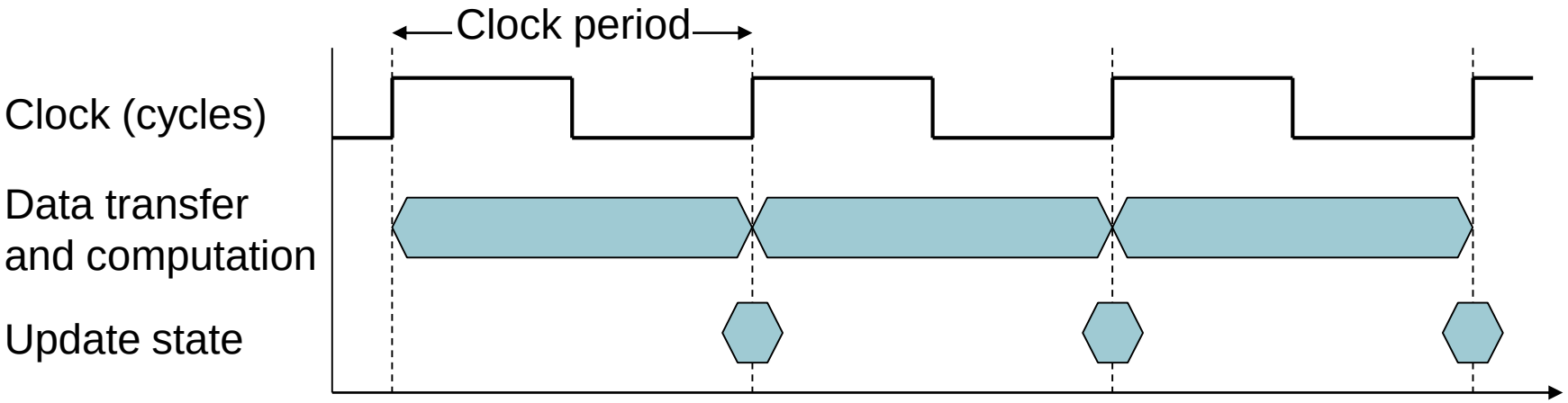
- Example: time taken to run a program
  - 10s on A, 15s on B
  - Execution Time_B / Execution Time_A
    = 15s / 10s = 1.5
  - So A is 1.5 times faster than B

Measuring Execution Time

- Elapsed time
  - Total response time, including all aspects
    - Processing, I/O, OS overhead, idle time
  - Determines system performance
- CPU time
  - Time spent processing a given job
    - Discounts I/O time, other jobs' shares
  - Comprises user CPU time and system CPU time
  - Different programs are affected differently by CPU and system performance

CPU Clocking

- Operation of digital hardware governed by a constant-rate clock



  - Clock period: duration of a clock cycle
    - e.g., 250ps = 0.25ns = 250×10^{-12}s
  - Clock frequency (rate): cycles per second
    - e.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz

CPU Time

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

- Performance improved by
  - Reducing number of clock cycles
  - Increasing clock rate
  - Hardware designer must often trade off clock rate against cycle count

CPU Time Example

- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
  - Aim for 6s CPU time
  - Can do faster clock, but causes 1.2 × clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}$$

$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A$$

$$= 10\text{s} \times 2\text{GHz} = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \frac{24 \times 10^9}{6\text{s}} = 4\text{GHz}$$

Instruction Count and CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- Instruction Count for a program
  - Determined by program, ISA and compiler
- Average cycles per instruction
  - Determined by CPU hardware
  - If different instructions have different CPI
    - Average CPI affected by instruction mix

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$

$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$

A is faster…

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$

$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2$$

…by this much

CPI in More Detail

- If different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

  - Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

CPI Example

- Alternative compiled code sequences using instructions in classes A, B, C

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

- Sequence 1: IC = 5
  - Clock Cycles
    = 2×1 + 1×2 + 2×3
    = 10
  - Avg. CPI = 10/5 = 2.0
- Sequence 2: IC = 6
  - Clock Cycles
    = 4×1 + 1×2 + 1×3
    = 9
  - Avg. CPI = 9/6 = 1.5

Performance Summary

The BIG Picture

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

Power

- In CMOS IC technology

$$\text{Pdyn} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

$\times 30$ | $5\text{V} \rightarrow 1\text{V}$ | $\times 1000$

- Dynamic power is dominant when active
- Static power: $\text{Pstat} = \text{Voltage} \times I_{static}$
- Total power P=Pstat+Pdyn

Reducing Power

- Suppose a new CPU has
  - 85% of capacitive load of old CPU
  - 15% voltage and 15% frequency reduction

$$\frac{P_{new}}{P_{old}} = \frac{C_{old} \times 0.85 \times (V_{old} \times 0.85)^2 \times F_{old} \times 0.85}{C_{old} \times V_{old}^{\ 2} \times F_{old}} = 0.85^4 = 0.52$$

- The power wall
  - We can't reduce voltage further
  - We can't remove more heat
- How else can we improve performance?

Multiprocessors

- Multicore microprocessors
  - More than one processor per chip
- Requires explicitly parallel programming
  - Compare with instruction level parallelism
    - Hardware executes multiple instructions at once
    - Hidden from the programmer
  - Hard to do
    - Programming for performance
    - Load balancing
    - Optimizing communication and synchronization

Pitfall: Amdahl's Law

- Improving an aspect of a computer and expecting a proportional improvement in overall performance

$$T_{improved} = \frac{T_{affected}}{improvement\ factor} + T_{unaffected}$$

- Example: multiply accounts for 80s/100s
  - How much improvement in multiply performance to get 5× overall?

$$20 = \frac{80}{n} + 20$$

  - Can't be done!
- Corollary: make the common case fast

Fallacy: Low Power at Idle

- i7 power benchmark
  - At 100% load: 258W
  - At 50% load: 170W (66%)
  - At 10% load: 121W (47%)
- Consider designing processors to make power proportional to load
- However today P= Pdyn+Pstat
  - Pstat is due to leakage power and residual activity
  - If Pstat >0 → careful with extreme parallelism!

Exercise

- Processor core IPC=1:
  - @1.5V, Pdyn= 0.1W, Pstat=0.01W, f=0.5GHz
  - @1.0V, f=0.4GHz
- Uncore
  - @1.5V, Pdyn+Pstat=0.1W
- Compare 2,4,8,16 core architecture with a 1 core architecture for running appl with 100GInstr
    - 50% sequential parr
    - 10% sequential part
    - 1% sequential part
- Compute power, energy