Parallel Computing for Beginners

Chapter 0: Computing

Joel C. Adams, Richard A. Brown, Suzanne J. Matthews, and Elizabeth Shoop (CSinParallel)

Fall Semester 2025/26

Welcome to Parallel Computing

- Welcome to Parallel Computing for Beginners!
- This chapter lays the conceptual foundations needed for the chapters that follow
- Many concepts to cover - get ready to learn many new things!
- Starting your parallel computing journey

What is Computing?

Definition

Computing can be defined as:

"Designing, writing, and/or running software in order to solve a problem"

Let's examine each component of this definition in detail.

Designing Software

- Consists of carefully specifying what must be done to solve the problem
- Involves devising an **algorithm**

Algorithm

A series of well-defined steps that solves the problem

This is the conceptual phase where we plan our approach to the problem.

Writing Software

- Consists of implementing an algorithm in a particular programming language
- Results in a **computer program**
- The program must be thoroughly tested to ensure it actually solves the problem
- If testing fails, it may be necessary to:
  - Revisit the design
  - Start anew

Running Software

- Consists of having a computer perform the program's statements
- We must first be confident that the program correctly solves the problem
- Once confident, we can solve the problem by running the program on a computer

This is where the actual problem solving happens!

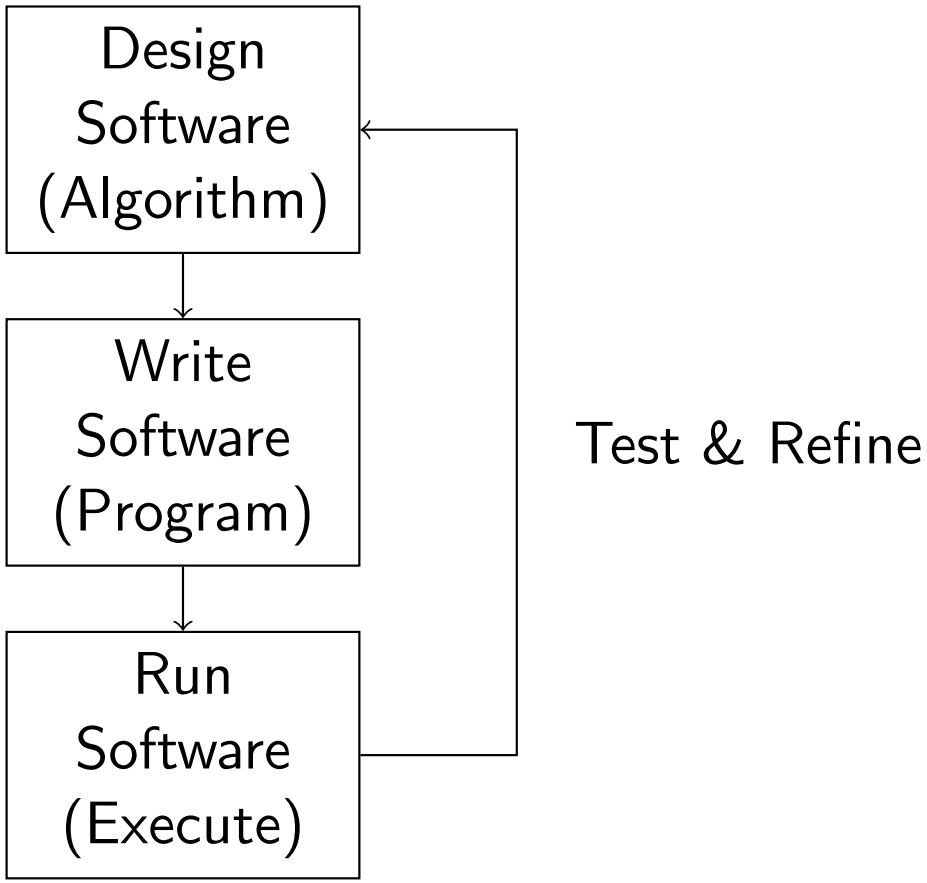
The Two Components of Computing

Computing consists of two parts:

1. **Software** or program that is being used to solve a problem
2. **Hardware** or computer on which the software is being run

Both components are essential for successful computing - neither can function without the other.

The Computing Process: Summary



Central Processing Unit (CPU)

- The CPU is the hardware component that performs a program's statements
- Contains circuitry needed for arithmetic, logical, and other program operations
- This circuitry is sometimes called the CPU's **core**
- Prior to 2005: Most computers had single-core CPUs
- Single cores perform program statements sequentially (one after another)

Evolution from Sequential to Multicore

- **Sequential Computing**: Single core performs statements one after another
- Programs written for single-core CPUs were called **sequential programs**
- **2005**: CPU manufacturers began building multicore CPUs
- Multicore CPUs can perform multiple statements simultaneously
- Evolution: dual-core $\rightarrow$ quad-core $\rightarrow$ today's 64- and 128-core CPUs

Multiprocessors

- **Multiprocessor**: A computer that can perform more than one statement at a time
- Today's multicore CPUs are one type of multiprocessor
- Multiprocessors have existed since the 1960s (mainframe computers with multiple CPUs)
- 1970s-80s: Cray Research built the first supercomputers
- 1990s: NASA built Beowulf clusters using networked personal computers

Three Categories of Multiprocessors

1. **Shared Memory Multiprocessor**
2. **Distributed Memory Multiprocessor**
3. **Heterogeneous Multiprocessor**

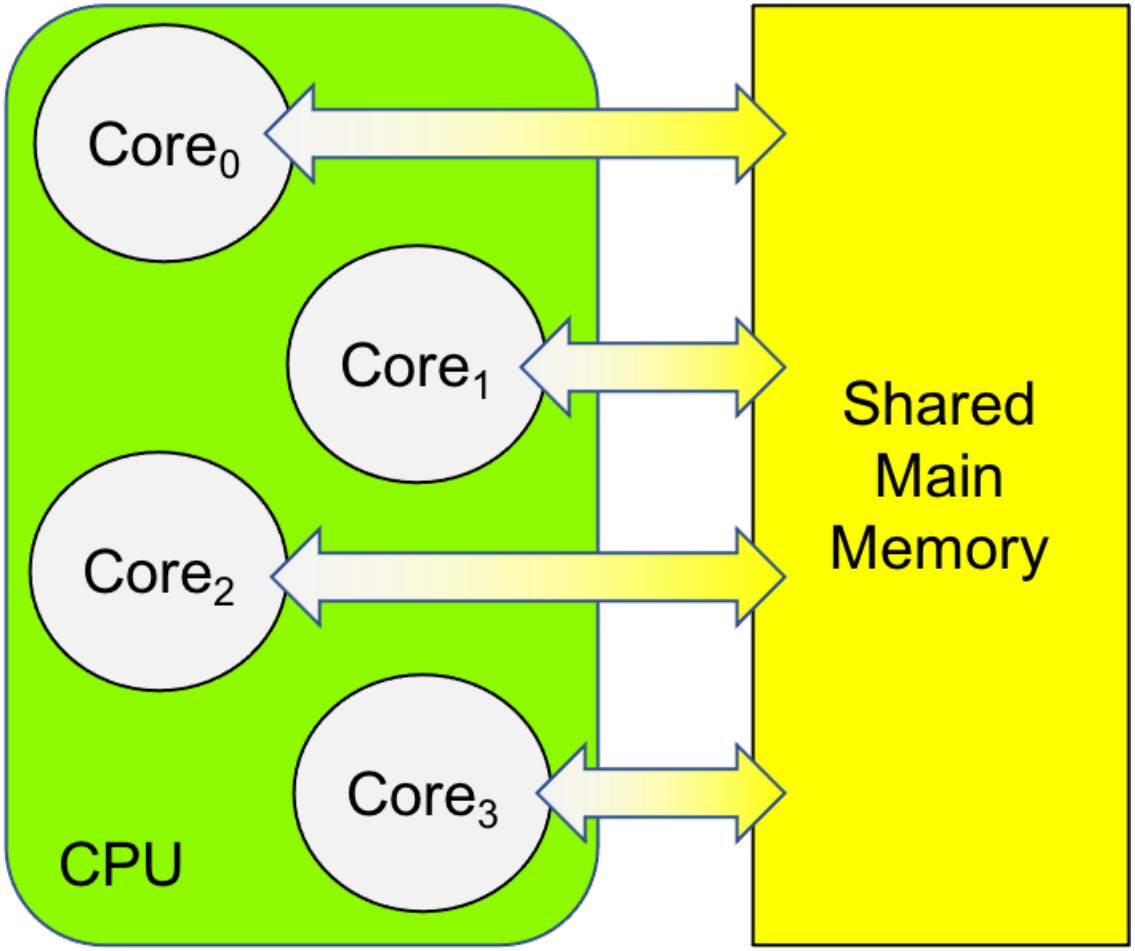
Each category has distinct characteristics in terms of memory organization and processing capabilities.

Shared Memory Multiprocessors

- Multiple cores share the same main memory
- Most common type in today's personal computers:
  - Desktops
  - Laptops
  - Tablets
  - Smartphones
  - Single board computers (e.g., Raspberry Pi)
- All cores can access the same memory space

Shared Memory Multiprocessors

Diagram Reference: Shared Memory Multiprocessor

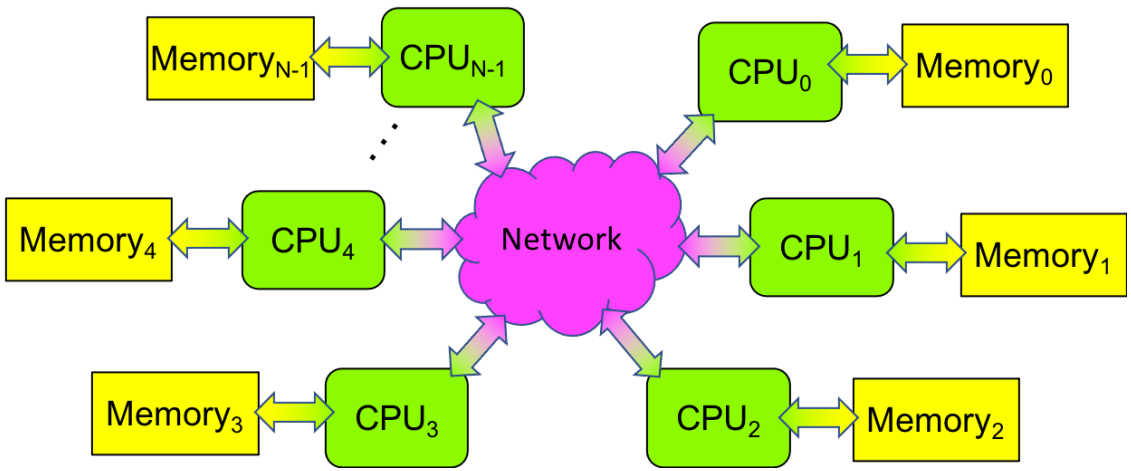


Distributed Memory Multiprocessors

- Multiple CPUs, each with its own local memory
- No memory shared between CPUs
- CPUs connected via network for communication
- Example: Original Beowulf clusters
- Each CPU operates independently with its own memory space

Distributed Memory Multiprocessors

Diagram Reference: Distributed Memory Multiprocessor

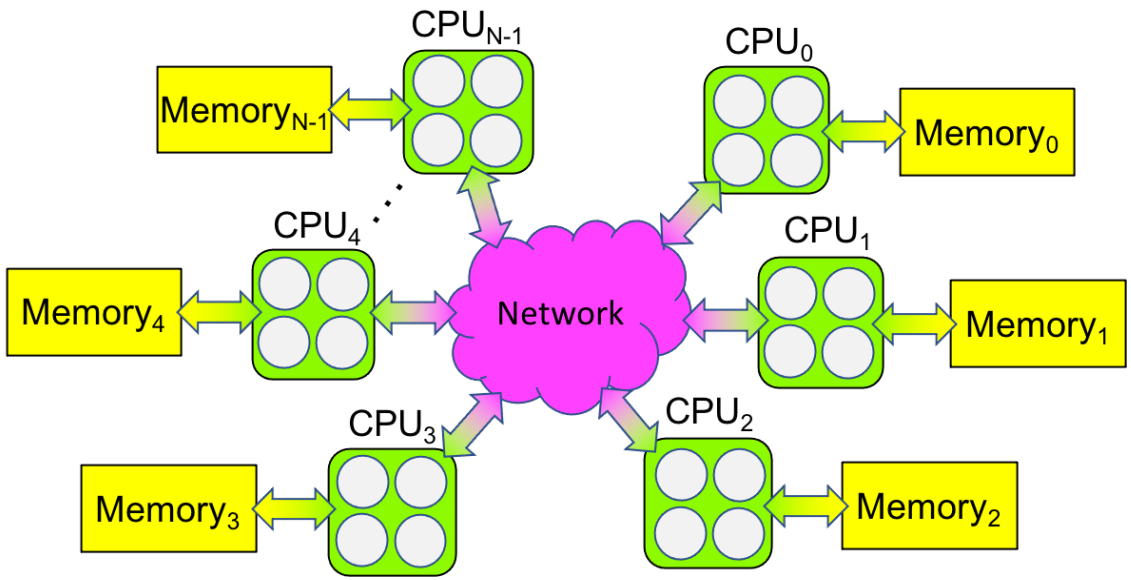


Heterogeneous Multiprocessors

- Distributed memory multiprocessor with additional features:
  - CPUs are shared memory multiprocessors (multicore)
  - Use Graphics Processing Units (GPUs) for acceleration
- Many of today's supercomputers are heterogeneous multiprocessors
- Essentially Beowulf clusters enhanced with:
  - Multicore CPUs
  - Hardware accelerators (GPUs)

Heterogeneous Multiprocessors

Diagram Reference: Heterogeneous Multiprocessor



Types of Heterogeneous Multiprocessors

- **Within-node heterogeneous multiprocessor**:
  - Each node supports different processing capabilities
  - Example: CPUs and GPUs in the same system
  - Most common type
- **Across-node heterogeneous multiprocessor**:
  - Different nodes provide different processing capabilities
  - Example: CPUs by different manufacturers (ARM and Intel)
  - Uncommon but possible

Impact on Program Design and Implementation

- Computing involves three steps:
  1. Designing a program
  2. Implementing a program
  3. Running a program on a computer
- **Key Point**: The type of computer affects how you design and implement programs
- Different multiprocessor architectures require different programming approaches
- Understanding hardware is crucial for effective program development

Summary

- CPU evolution: single-core $\rightarrow$ multicore (2005 onwards)
- Three types of multiprocessors:
  - Shared memory (common in personal computers)
  - Distributed memory (original clusters)
  - Heterogeneous (modern supercomputers)
- Hardware architecture influences program design
- Modern computing leverages parallel processing capabilities
- Understanding hardware is essential for effective programming

Introduction to Parallel Performance

Key Questions in Parallel Performance

Once we have written a parallel program for a given multiprocessor platform, we need to answer:

- How much faster is our parallel program than a sequential program that solves the same problem?
- How efficiently does our program use its hardware?

Key terminology:

- **Processing Element (PE)**: Generic term for parallel entities used by a parallel program
- **N**: Problem size (how "big" is the problem?)
- **P**: Number of PEs used to solve the problem

The Pizza Problem Setup

Penelope's Parallel Pizzeria sells pizzas of different sizes:

- Personal pizzas: 4 slices
- Medium pizzas: 8 slices
- Large pizzas: 12 slices
- X-large pizzas: 16 slices

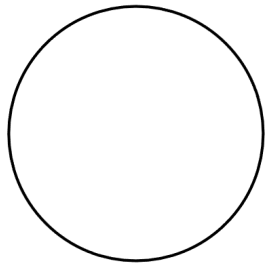
Problem: Consume a pizza as quickly as possible

- N = number of slices in the pizza
- P = number of PEs (pizza-eaters)
- Each person takes 2 minutes to eat a slice

Simple Parallel Pizza-Eating Algorithm

Simple Parallel Pizza-Eating Algorithm
(performed by each person)

1. Define N: How many slices of pizza? N: ☐
2. Define P: How many people? P: ☐
3. Compute slices per person (N/P). slicesPP: ☐
4. Get your own personal, unique id number (0..P-1). id: ☐
5. Assign each slice of pizza a unique number (0..N-1).
6. Compute your starting slice number (id * slicesPP). start: ☐
7. Compute your stopping slice number (start + slicesPP). stop: ☐
8. For (s = start; s < stop; ++s):
   Eat slice s. s (slice #s eaten): ☐



Simple Parallel Pizza-Eating Algorithm

The algorithm steps:

1. Read N and P
2. Get id (unique identifier for each PE)
3. Compute slicesPP = N / P
4. Compute start = id $\times$ slicesPP
5. Compute stop = start + slicesPP
6. For s = start to stop-1: Eat slice s

Scenario Analysis: Small Problem (N=4)

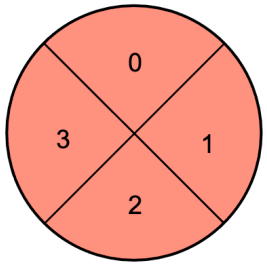
Personal Pizza (4 slices):

- P=1: Time $= 1 + 4 \times 2 = 9$ minutes (sequential)
- P=2: Time $= 1 + 2 \times 2 = 5$ minutes
- P=4: Time $= 1 + 1 \times 2 = 3$ minutes
- P=8: Time $= 1 + 0.5 \times 2 = 2$ minutes

Scenario Analysis: Small Problem (N=4)

Simple Parallel Pizza-Eating Algorithm
(performed by each person)

1. Define N: How many slices of pizza? N: 4
2. Define P: How many people? P: 1
3. Compute slices per person (N/P). slicesPP: 4
4. Get your own personal, unique id number (0..P-1). id: 0
5. Assign each slice of pizza a unique number (0..N-1).
6. Compute your starting slice number (id * slicesPP). start: 0
7. Compute your stopping slice number (start + slicesPP). stop: 4
8. For (s = start; s < stop; ++s):
   Eat slice s. s (slice #s eaten): 0-3



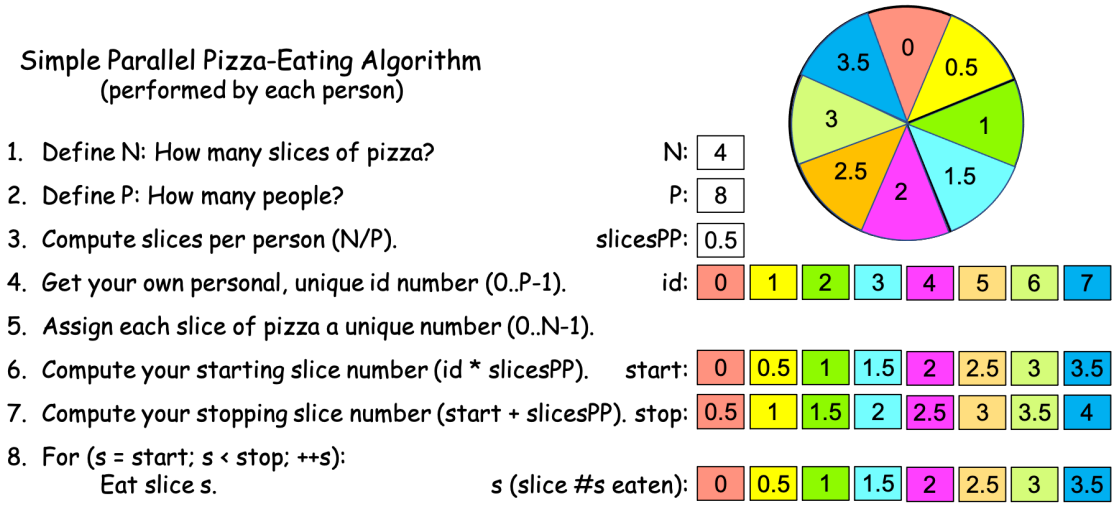
Notice: Diminishing returns as P increases beyond P=4

Scenario Analysis: Larger Problem (N=16)

X-Large Pizza (16 slices):

- P=1: Time $= 1 + 16 \times 2 = 33$ minutes
- P=2: Time $= 1 + 8 \times 2 = 17$ minutes
- P=4: Time $= 1 + 4 \times 2 = 9$ minutes
- P=8: Time $= 1 + 2 \times 2 = 5$ minutes

Scenario Analysis: Larger Problem (N=16)



Key insight: Larger problem sizes allow for better scalability with more PEs

Speedup Definition and Calculation

Speedup has a precise meaning in parallel computing:

$$Speedup_P = \frac{Time_1}{Time_P}$$

Where:

- $Time_1$ = time to solve problem with P=1 (sequential)
- $Time_P$ = time to solve problem with P PEs

Examples from Pizza Scenarios:

- N=4, P=2: $Speedup_2 = 9/5 = 1.8$ (ideal: 2.0)
- N=16, P=2: $Speedup_2 = 33/17 = 1.94$ (closer to ideal)

Computational Efficiency

Computational Efficiency measures how well we utilize our PEs:

$$Efficiency_P = \frac{Speedup_P}{P}$$

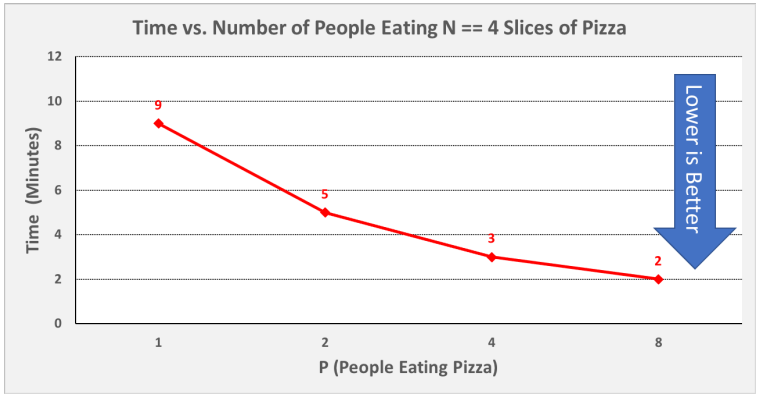
In a perfect world, $Efficiency_P = 1.0$ (100%)

Acceptability threshold: $Efficiency_P > 0.6$ (60%)

Examples:

- N=4, P=8: $Efficiency_8 = 4.5/8 = 0.56$ (unacceptable)
- N=16, P=8: $Efficiency_8 = 6.6/8 = 0.83$ (acceptable)

Efficiency Comparison



Key observations:

- Efficiency tends to decrease as P increases for fixed N
- Larger problem sizes (N) yield better efficiency for any given P
- The larger the problem size, the higher the efficiency

Scalability Concepts

Scalability: How well a parallel computation's speedup increases as the number of PEs increases

Embarrassingly/Perfectly Parallel: Computations where:

- $Speedup_P \approx P$
- $Efficiency_P \approx 1$
- The computation scales perfectly

Most real problems are not perfectly parallel due to:

- Sequential setup steps (like steps 1-7 in pizza algorithm)
- Coordination and synchronization overhead

Amdahl's Law

Gene Amdahl's 1967 formula predicts parallel performance limits:

$$Speedup_P = \frac{1}{seqPct + \frac{parPct}{P}}$$

Where:

- **parPct**: Percentage of runtime that benefits from parallelization
- **seqPct**: Percentage of runtime that must be performed sequentially
- $seqPct + parPct = 1.0$

Key insight: As $P \to \infty$, $Speedup_P \to \frac{1}{seqPct}$

Amdahl's Law Examples

Scenario 1 (N=4):

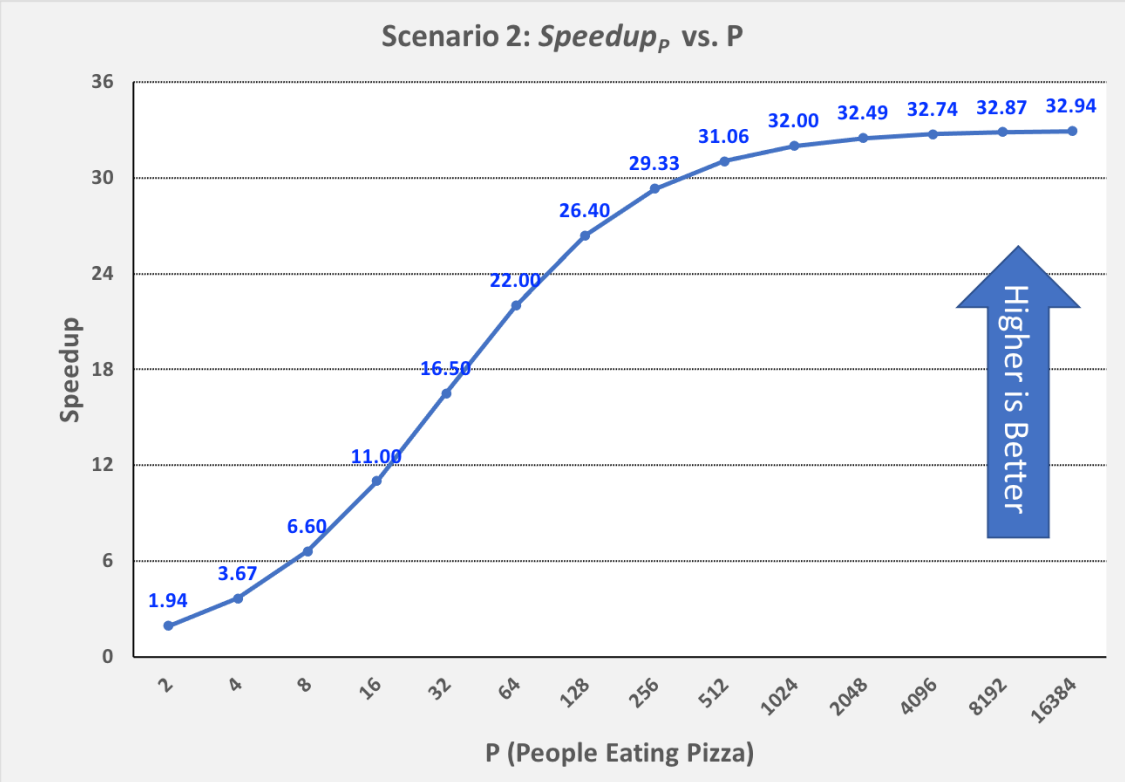
- Sequential part: 1 minute, Parallel part: 8 minutes
- $seqPct = 1/9 = 0.11$ (11%)
- Maximum possible speedup: $1/0.11 = 9$

Scenario 2 (N=16):

- Sequential part: 1 minute, Parallel part: 32 minutes
- $seqPct = 1/33 = 0.03$ (3%)
- Maximum possible speedup: $1/0.03 = 33$

Amdahl's Law Examples

Amdahl's Law Examples



The Gustafson-Barsis Law (1988)

Answer to Amdahl's pessimistic outlook:

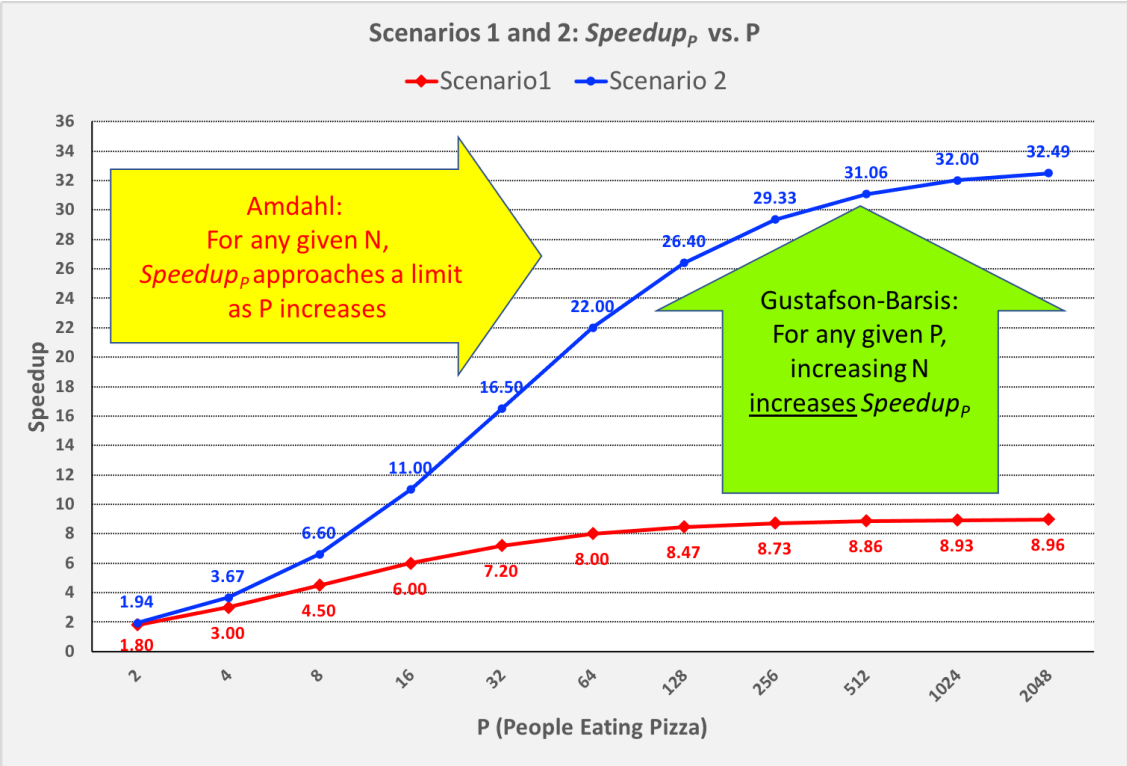
$$Speedup_P = P + seqPct \times (1 - P)$$

Key insight: If we keep P fixed but increase N:

- Sequential portion stays the same
- Higher percentage of time spent in parallel portion
- As $N \to \infty$, $seqPct \to 0$ and $Speedup_P \to P$ (ideal!)

Rescue from Amdahl's Law: Provided sequential time remains constant as problem size increases

Comparing Amdahl vs Gustafson-Barsis



Amdahl's Law: For fixed N, as $P \to \infty$, approach asymptotic limit

Gustafson-Barsis Law: For fixed P, as $N \to \infty$, $Speedup_P \to P$

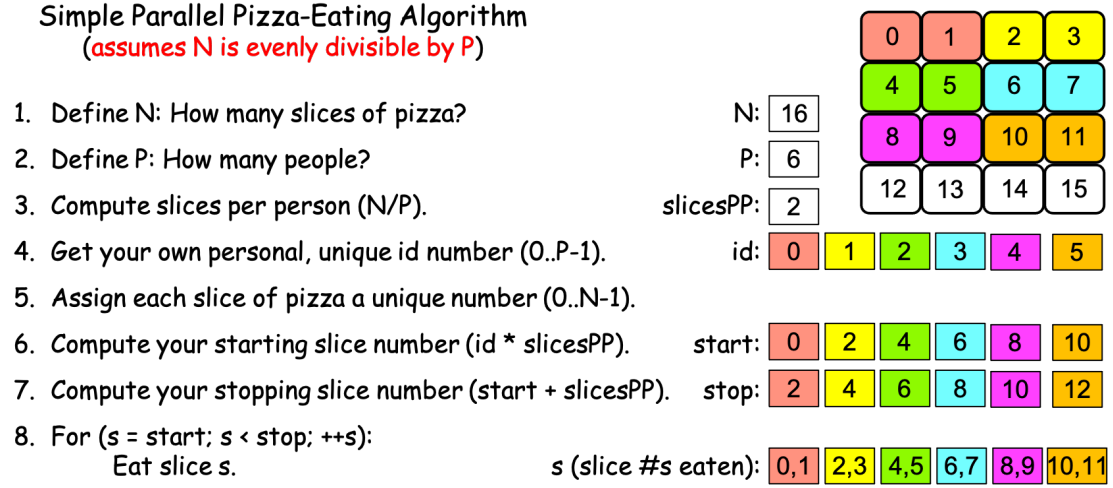
The key is that increasing problem size can overcome the limitations imposed by sequential portions of the algorithm.

Problem with Simple Algorithm

The simple pizza-eating algorithm has a flaw:
What happens when N is not evenly divisible by P?

Example: N=16, P=6

- Each PE gets $\lfloor 16/6 \rfloor = 2$ slices
- Total slices consumed: $6 \times 2 = 12$
- Remaining slices: $16 - 12 = 4$ (uneaten!)



Precondition: Simple algorithm only works if N is evenly divisible by P

Strategy 1: Consecutive Slices Parallel Loop

"Consecutive Slices" Parallel Pizza-Eating Algorithm
(does not assume N is evenly divisible by P)

1. Define N: How many slices of pizza?
2. Define P: How many people?
3. Compute slicesPP1 = ceiling(float(N)/P).
4. Compute slicesPP2 = slicesPP1 – 1.
5. Compute remnants = N % P (the remainder of N/P).
6. Assign each slice of pizza a unique number (0..N-1).
7. Get your own personal, unique id number (0..P-1).
8. Compute start = id * slicesPP1; stop = start + slicesPP1.
9. If (remnants > 0) AND (id >= remnants), recompute:
   start = remnants * slicesPP1 + (id – remnants) * slicesPP2;
   stop = start + slicesPP2.
10. For (s = start; s < stop; ++s):
    Eat slice s.

N:	16
P:	6
slicesPP1:	3
slicesPP2:	2
remnants:	4

id:	0	1	2	3	4	5
start:	0	3	6	9	12	14
stop:	3	6	9	12	14	16

s (slice #s eaten):	0-2	3-5	6-8	9-11	12-13	14-15

Modified algorithm to handle remnants:

1. Compute $slicesPP1 = \lceil N/P \rceil$ (ceiling function)
2. Compute $slicesPP2 = slicesPP1 - 1$
3. Compute $remnants = N \bmod P$
4. First R PEs get $slicesPP1$ slices each
5. Remaining PEs get $slicesPP2$ slices each

Distributes remnant slices among first few PEs

Consecutive Slices: Example

N=16, P=6:

- $slicesPP1 = \lceil 16/6 \rceil = 3$
- $slicesPP2 = 3 - 1 = 2$
- $remnants = 16 \bmod 6 = 4$

Result:

- PEs 0,1,2,3: eat 3 slices each (total: 12 slices)
- PEs 4,5: eat 2 slices each (total: 4 slices)
- Total consumed: 16 slices

Strategy 2: Every P^{th} Slice Parallel Loop

Much simpler approach:

1. Read N and P
2. Get id
3. For(s = id; $s < N$; $s+ = P$): Eat slice s

Each PE eats every P^{th} slice, starting from its id

- PE 0: slices 0, P, 2P, 3P, ...
- PE 1: slices 1, P+1, 2P+1, 3P+1, ...
- PE 2: slices 2, P+2, 2P+2, 3P+2, ...

Every P^{th} Slice: Example

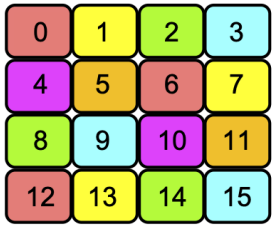
N=16, P=6:

- PE 0: eats slices 0, 6, 12 (3 slices)
- PE 1: eats slices 1, 7, 13 (3 slices)
- PE 2: eats slices 2, 8, 14 (3 slices)
- PE 3: eats slices 3, 9, 15 (3 slices)
- PE 4: eats slices 4, 10 (2 slices)
- PE 5: eats slices 5, 11 (2 slices)

'Every P^{th} Slice' Parallel Pizza-Eating Algorithm

1. Define N: How many slices of pizza?
2. Define P: How many people?
3. Assign each slice of pizza a unique number (0..N-1).
4. Get your own personal, unique id number (0..P-1).
5. For (s = id; s < N; s += P):
   Eat slice s.

N: 16
P: 6



id:	0	1	2	3	4	5
s (slice #s eaten):	0,6,12	1,7,13	2,8,14	3,9,15	4,10	5,11

Automatically handles non-divisible cases!

Comparing the Two Parallel Loop Strategies

Consecutive Slices

+ Good memory locality
+ Cache-aware
- More complex algorithm
- Poor load balancing for non-uniform processing times

Every P^{th} Slice

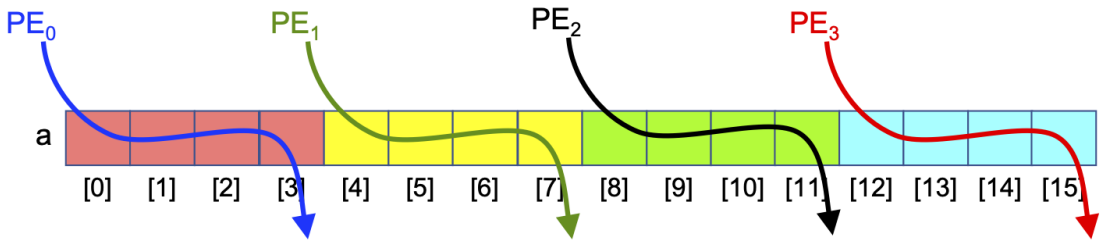
+ Simple algorithm
+ Better load balancing
- Poor memory locality
- Cache-oblivious

Choice depends on:

- Whether data is stored in arrays (consecutive better)
- Whether processing times are uniform (consecutive better)
- Whether processing times vary significantly (every P^{th} better)

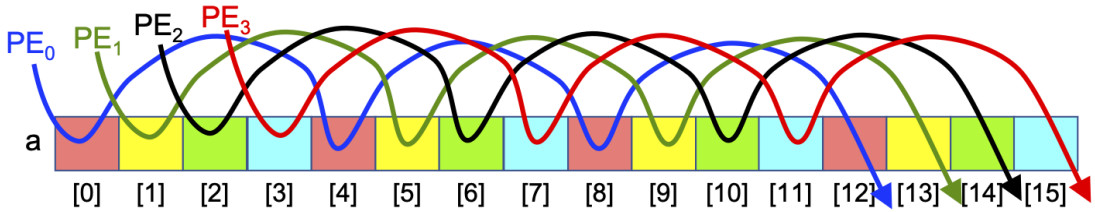
Memory Locality Comparison

Consecutive Slices - Good locality:



Each PE accesses adjacent memory locations

Every P^{th} **Slice** - Poor locality:

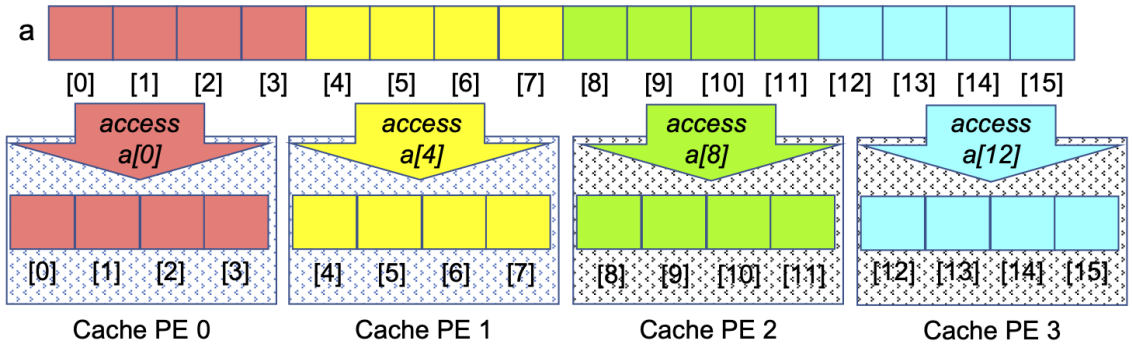


Each PE jumps around in memory, skipping P locations each time

Cache Awareness

Modern CPUs have high-speed local cache memory

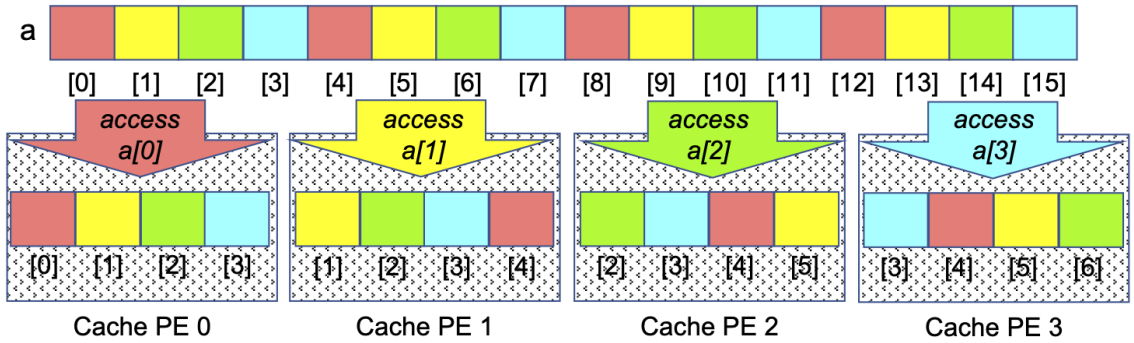
Consecutive Slices - Cache-aware:



- When PE accesses first value, hardware caches next several values
- Subsequent accesses hit cache (fast!)

Cache Awareness (continue)

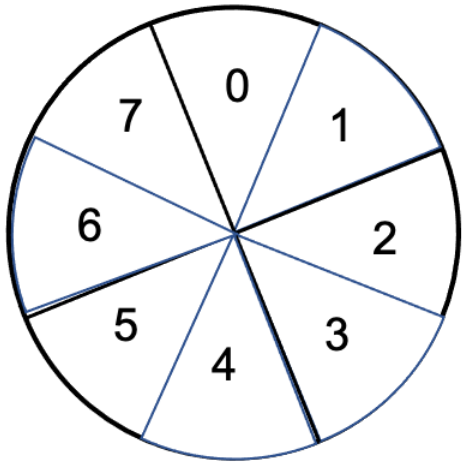
Every P^{th} Slice - Cache-oblivious:



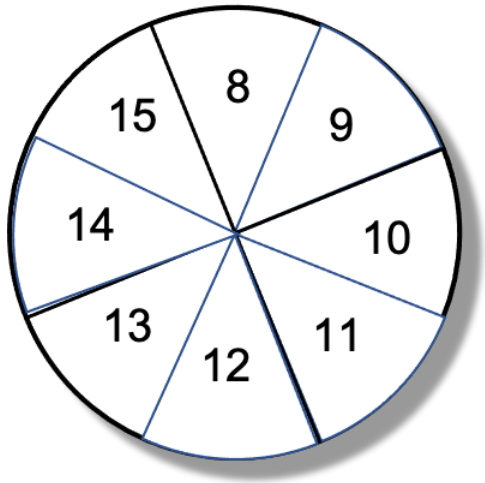
- Each access likely to miss cache
- Must fetch from main memory (slow!)

Load Balancing Example

Consider two different pizzas with non-uniform processing times:



A thin-crust plain pizza

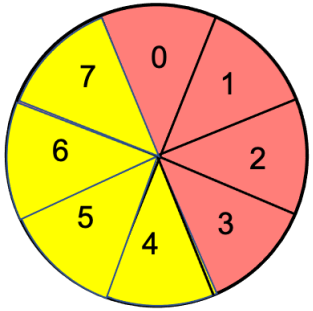


A double-crust triple-toppings deep-dish pizza

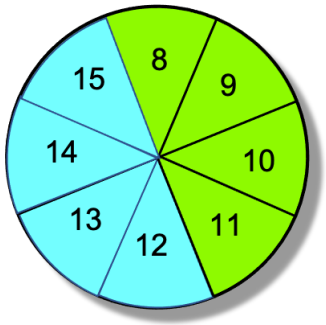
- Pizza 1: Thin crust, plain (fast to eat)
- Pizza 2: Deep dish, triple toppings (slow to eat)

Load Balancing Example (continue)

- **Consecutive Slices**: Some PEs get all easy/hard slices

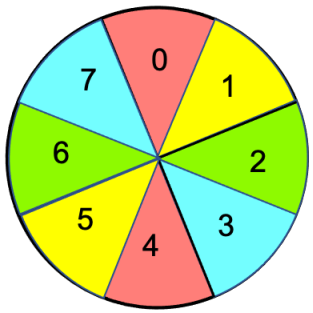


A thin-crust plain pizza

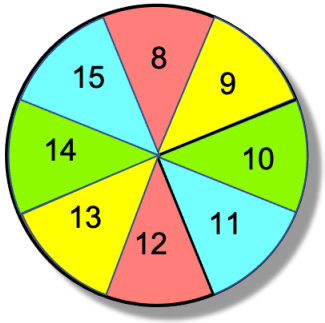


A double-crust triple-toppings deep-dish pizza

- **Every P^{th} Slice**: Each PE gets mix of easy and hard slices



A thin-crust plain pizza



A double-crust triple-toppings deep-dish pizza

Leader-Worker Strategy

Used when tasks cannot be easily numbered or pre-assigned

- One PE (usually id==0) designated as **leader**
- All other PEs are **workers**
- Leader distributes tasks dynamically
- Workers repeatedly: get task $\rightarrow$ perform task $\rightarrow$ repeat

Algorithm structure:

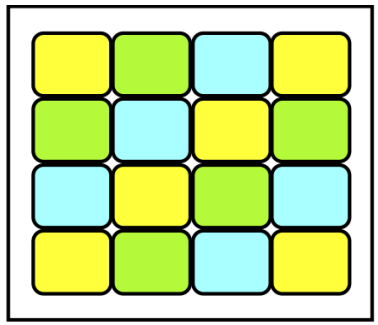
- If id == 0: Act as leader (distribute tasks, handle I/O)
- Else: Act as worker (get and perform tasks)

Leader-Worker Example

Leader-Worker Parallel Pizza-Eating Algorithm

1. Define N: How many slices of pizza? N: 16
2. Define P: How many people? P: 4
3. Define LEADER (0..P-1, usually 0). LEADER: 0
4. Get your own personal, unique id number (0..P-1). id: 0 1 2 3
5. If id == LEADER:
   - While the pizza-box is not empty:
     a. Get a slice of pizza from the box.
     b. Give that slice to a WORKER.
   - Announce: "The pizza is all gone!"

   Otherwise (non-LEADER == WORKER):
   - While there is pizza remaining:
     a. Get a slice of pizza from the LEADER.
     b. Eat that slice.



N=16, P=4:

- PE 0 (leader): Hands out pizza slices
- PEs 1,2,3 (workers): Get slice from leader $\rightarrow$ eat slice $\rightarrow$ repeat

Leader-Worker Example (continue)

Advantages:

- Dynamic load balancing
- No need to number tasks
- Handles variable processing times well

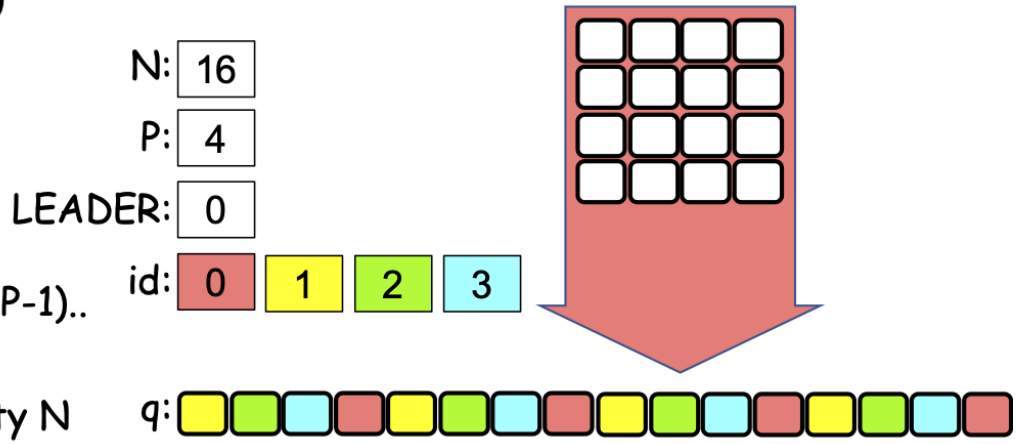
Disadvantages:

- Leader can become bottleneck
- Communication overhead

Leader-Worker + Barrier

Shared Queue Parallel Pizza-Eating Algorithm
(with Barrier and Leader-Worker)

1. Define N: How many slices of pizza?
2. Define P: How many people?
3. Define LEADER (a value from 0..P-1).
4. Get your own personal, unique id number (0..P-1)..
5. If id == LEADER:
   - Create an empty, shared queue q, capacity N
   - While the pizza box is not empty:
     a. Get a slice of pizza from the box
     b. Append that slice to the queue q.
6. BARRIER (wait here until all PEs arrive).
7. While q is not empty:
   a. Try to remove a slice of pizza from q
   b. If successful, eat that slice.



The Reduction Problem

Often each PE produces a partial solution that must be combined

Example: Determine total weight of cheese on 16-slice pizza

- Each PE removes cheese from their slices and weighs it
- Each PE has partial weight (partial solution)
- Need to combine into total weight (total solution)

Sample partial results:

PE	0	1	2	3	4	5	6	7
Weight	44	40	43	43	40	41	44	41

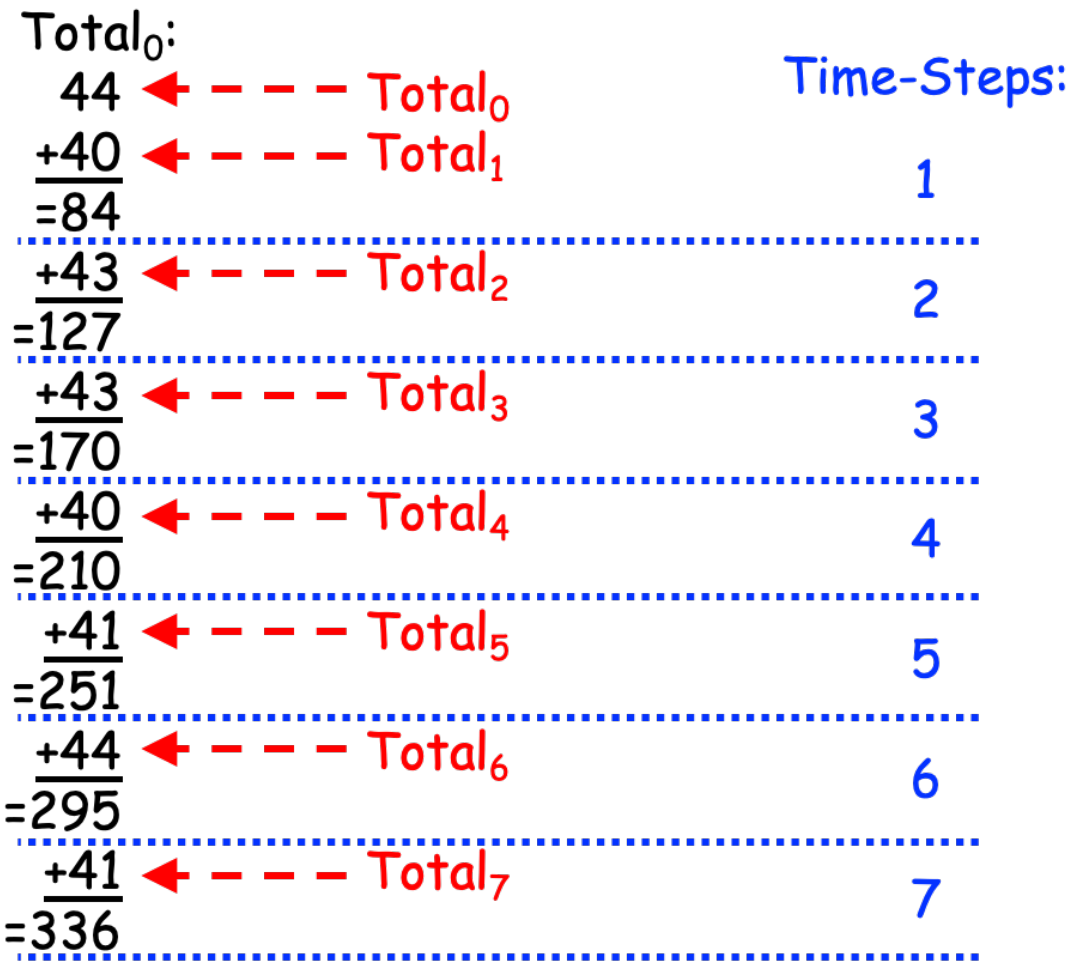
Linear Reduction Strategy

Simple approach: Leader does all the work

1. PE 0 (leader) starts with its partial result
2. For i = 1 to P-1:
   - Ask PE i for its partial result
   - Add that result to running total

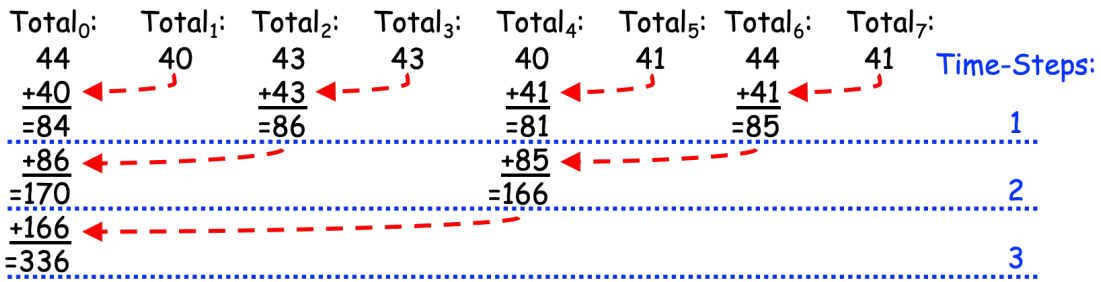
Time complexity: P-1 steps (linear time)
Problem: Most workers sit idle; leader does all work

Linear Reduction Strategy (continue)



Parallel Reduction Strategy

Better approach: Distribute the work among PEs



Tree-like structure:

- Round 1: PEs pair up, half become inactive
- Round 2: Remaining PEs pair up, half become inactive
- Continue until only leader remains

Time complexity: $\log_2(P)$ steps (logarithmic time)
Advantage: Much faster as P increases

- P=8: 3 steps vs 7 steps
- P=1024: 10 steps vs 1023 steps!

Parallel Reduction Example

P=8 PEs, 3 rounds:

Round 1 ($i = 1$, $2^i = 2$):

- Even PEs (0,2,4,6) get values from odd PEs (1,3,5,7)
- PE 0: $44 + 40 = 84$, PE 2: $43 + 43 = 86$, etc.
- Odd PEs quit

Round 2 ($i = 2$, $2^i = 4$):

- PEs divisible by 4 (0,4) get values from PEs 2,6
- PE 0: $84 + 86 = 170$, PE 4: $40 + 41 + 44 + 41 = 166$

Round 3 ($i = 3$, $2^i = 8$):

- PE 0 gets value from PE 4: $170 + 166 = 336$

Single Program, Multiple Data (SPMD)

The Parallel Loop, Leader-Worker, and Reduction strategies are all examples of the **SPMD approach**

Key concept: Write a single program that:

- Runs on multiple PEs simultaneously
- Each PE processes different data items
- Uses PE's id and P to determine behavior

Required capabilities:

1. Determine P (number of PEs available)
2. Determine id (unique identifier for each PE, 0 to P-1)

Usage of id:

- Parallel Loops: id determines which iterations to perform
- Leader-Worker: id determines leader vs worker role
- Reduction: id determines pairing and roles

Decomposition Approaches

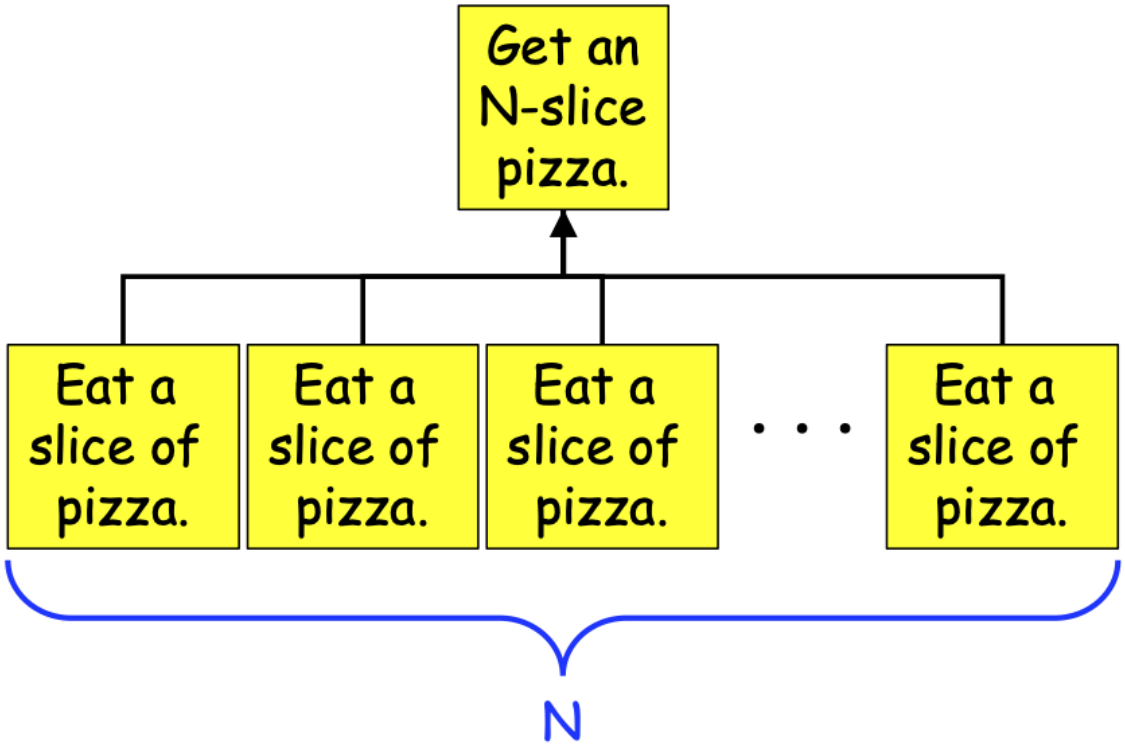
Decomposition: Dividing up work to solve a problem in parallel

Two main approaches:

1. **Data Decomposition**:
   - Divide large dataset into pieces
   - Each PE processes subset of data
   - Use Parallel Loops + Reduction if needed
2. **Task Decomposition**:
   - Identify functional tasks to be performed
   - Build dependency graph
   - Map independent tasks to PEs

Task Decomposition Example

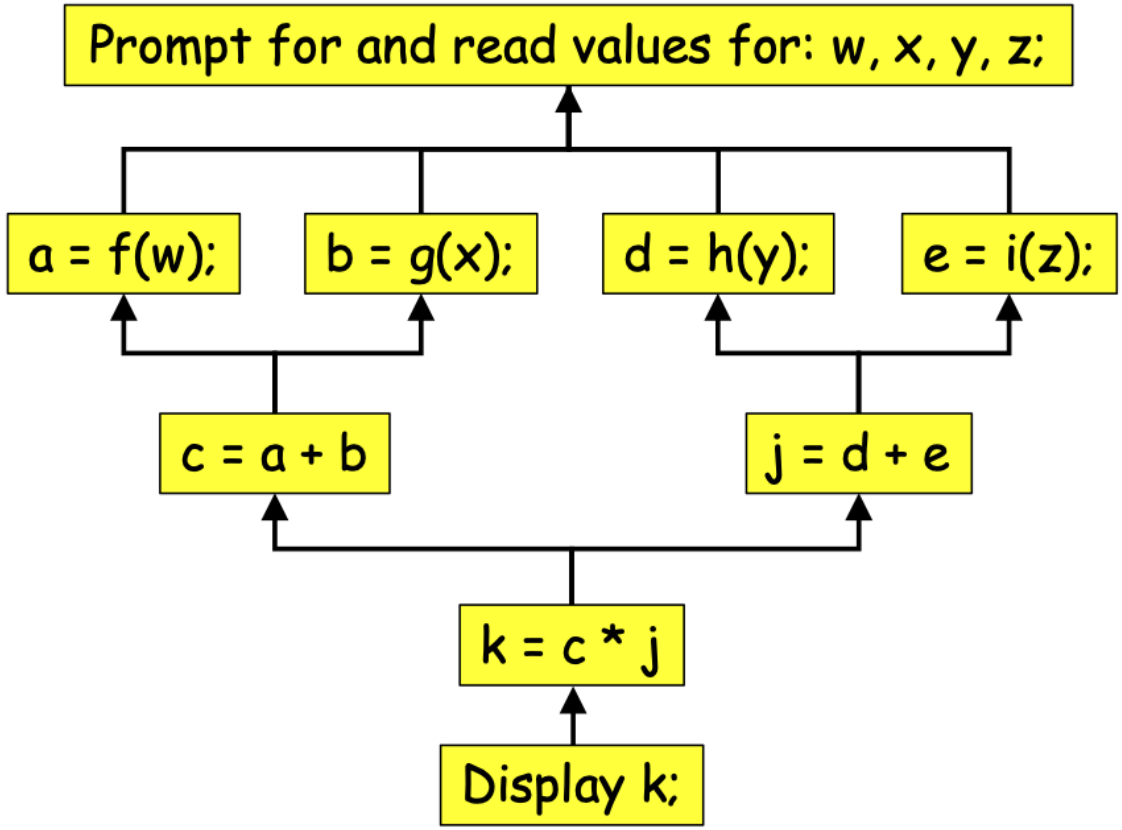
Pizza eating as task decomposition:



- N+1 tasks: 1 "Get pizza" + N "Eat slice" tasks
- "Eat slice" tasks depend on "Get pizza" task
- "Eat slice" tasks independent of each other → can parallelize

Complex Task Dependencies

Program with multiple functional dependencies:



- Tasks: Read inputs, compute a,b,d,e, compute c,j, compute k, display
- Maximum parallelism: 4 PEs (computing a,b,d,e simultaneously)
- Scalability limited by T_{max} (max independent tasks)

Scalability Comparison

Task Decomposition:

- Scalability limited by T_{max} (maximum independent tasks)
- Maximum speedup T_{max}, regardless of P
- Cannot benefit from more PEs beyond T_{max}

Data Decomposition:

- Scalability limited by P and N
- Each PE processes N/P data items
- Can increase speedup by increasing N (Gustafson-Barsis)
- As $N \to \infty$, $Speedup_P \to P$

Focus of this book: Data decomposition, especially for B̈ig Dataëra

Summary

Key Performance Concepts:

- Speedup = $Time_1 / Time_P$
- Efficiency = $Speedup_P / P$
- Amdahl's Law: Sequential portions limit speedup
- Gustafson-Barsis: Larger problems overcome limitations

Parallel Programming Strategies:

- Parallel Loops: Consecutive chunks vs Every Pth element
- Leader-Worker: Dynamic task distribution
- Reduction: Combine partial results (linear vs parallel)
- SPMD: Single program, multiple data approach

Design Choice: Data decomposition preferred for scalability in Big Data applications