TEXAS STATE UNIVERSITY

The rising STAR of Texas

Parallel Performance – Basic Concepts

Module developed Spring 2013
By Wuxu Peng

This module created with support form NSF under grant # DUE 1141022

Improving Computer Performance

- What performance translates into:
  - Time taken to do computation
    - Improving performance → reducing time taken
- What key benefits improving performance can bring:
  - Can solve "now-computationally-attainable" problems in *less time*
    - Save time (and time is valuable, with some equating time to money)
  - May *enable solving* "now-computationally-unattainable" problems
    - E.g.: Higher resolution numerical weather prediction → better/safer living(?)
- 2 facets of performance: (*time/unit_of_work*) vs (*units_of_work/time*)
  - Usually related, sometimes inversely: *raw speed vs throughput*
  - Depending on application, one may trump the another in importance
    - Freeways: good for moving around suburbs and between metropolitan areas
    - City streets: good for moving into and around urban core
- What key factors determine performance:

$$Total\ Time\ to\ Do\ Computation = T\downarrow T = N\downarrow I \times CPI \times T\downarrow C$$

  - $N\downarrow I$ = # of instructions to execute (to complete computation)

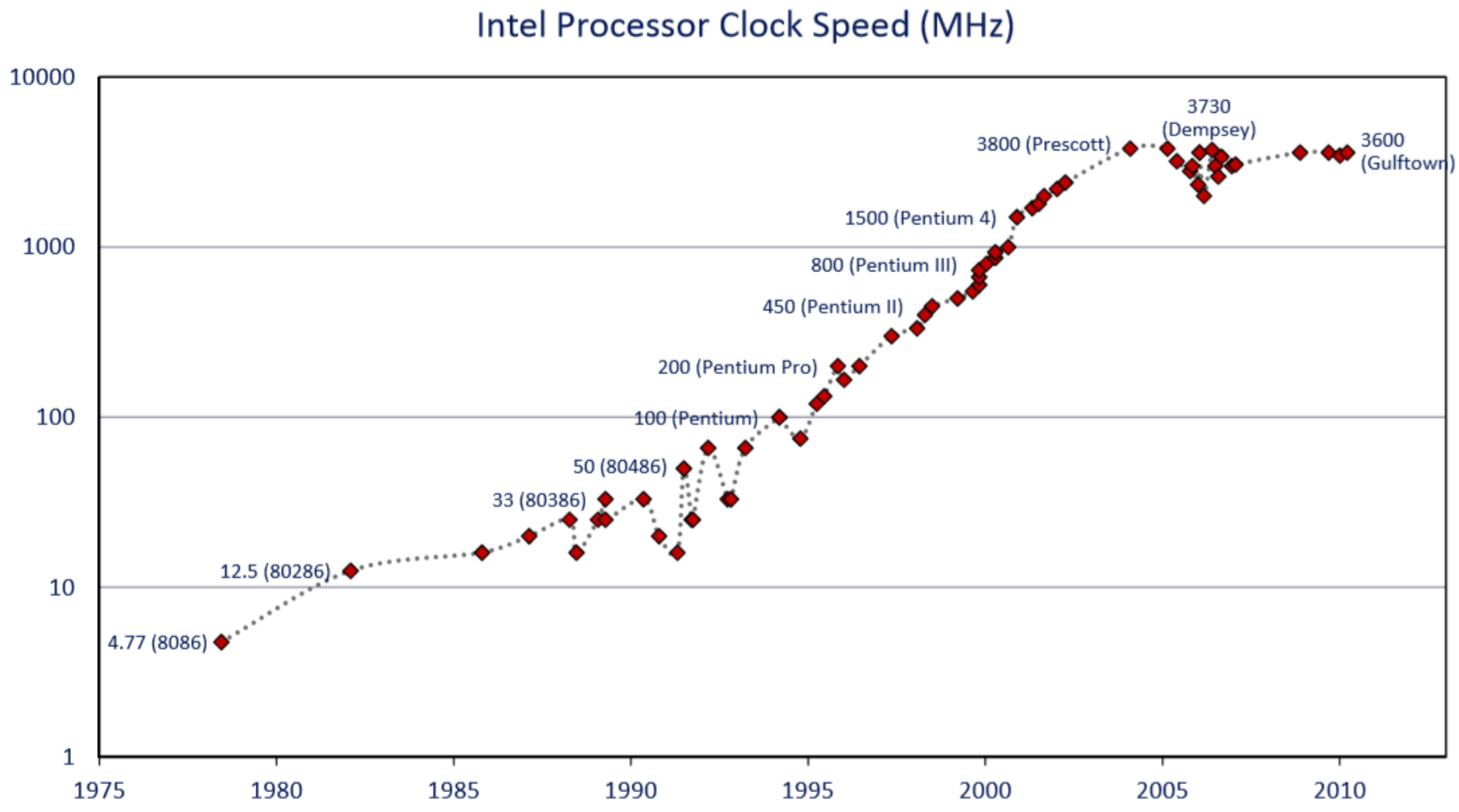
Running into a Wall with the Clock

- What key factors determine performance:

  Total Time to Do Computation = $T\downarrow I = N\downarrow I \; x CPI x T\downarrow C$

- Lowering $T\downarrow C$ (increasing clock rate):
  - Matter of engineering
    - Use more advanced material, better production techniques, etc.
  - Reflected in:
    - Systems of progressively higher clock rates over the years
  - Hitting a wall:
    - How fast can data be moved through hardware (to feed processors)
      - Absolute limit → speed of light (but "getting hopelessly hot" long before that)
    - How small can components be
      - Absolute limit → size of molecule/atom (if technology ever ventured near that)
    - How long can pressing on in this direction remain economically justifiable
      - Increasingly more expensive to make (single) processor run faster
      - Tell-tale trend → multicore computers

Clock Speeds Peaked in ca. 2005



Parallelism Unifies "Non-clock" Routes

- What key factors determine performance:

  Total Time to Do Computation = $T\downarrow I = N\downarrow I$ xCPIx $T\downarrow C$

- Lowering $N\downarrow I$ and CPI:
  - Various ways at various levels → classified by some as follows:
    - Bit level
    - Instruction level
    - Data level
    - Task level
  - *Common thread* running through these ways:
    - Parallelism → *Bingo! Voila!* ***Parallel Computing!***

What is Parallel Computing

- Traditional *serial (sequential)* computing:
  - Uses single computer with single CPU to solve problem
  - Problem solved using finite # of hardware-executable operations
  - Operations executed one after another and one at a time
- *Parallel* computing:
  - Uses multiple computing resources to solve problem *concurrently*
    - Multiple computing resources may be
      - Single computer with multiple computing components

      (component may be entire processor, or nearly so → multicore computer)
      - Multiple computers connected by some interconnecting link (e.g., network)
      - Combination of both
  - Problem broken into parts that can be solved *in parallel*
    - If this can't be done to a useful extent → little use for parallel computing
      - Parallel computing challenge → develop/implement *parallelizing ideas*
  - Each part solved using finite # of hardware-executable operations
  - Operations for each part executed *simultaneously*
    - Using the multiple computing resources

Parallelism Unifies "Non-clock" Routes

- Lowering $N \downarrow I$ and CPI:
  - Various ways at various levels → classified by some as follows:
    - Bit level
    - Instruction level
    - Data level
    - Task level
- Bit level parallelism:
  - Mainly concerned with design of ALU (and its supporting casts)
  - May-not-be-immediately-clear example →↑ computer *word size*
    - (word size → amount of data processor can manipulate per cycle)
    - ↓ # of instructions needed to execute operations on *word-size*+ operands
      - E.g.: 32-bit-int addition → 2 instructions for 16-bit ALU, but 1 for 32-bit ALU
    - Microprocessors historically:
      - 4-bit → 8-bit → 16-bit → 32-bit (steadily), (= 2 decade lull), 64-bit (=2003-4)

Parallelism Unifies "Non-clock" Routes

- Lowering $N \downarrow I$ and CPI:
  - Various ways at various levels → classified by some as follows:
    - Bit level
    - Instruction level
    - Data level
    - ~~Task level~~
- Instruction level parallelism:
  - Reorder and group instructions to ↑ parallel execution opportunity
    - Improves *units_of_work/time* (not *time/unit_of_work*) performance
  - *Pipelining* → most common way to exploit such parallelism
    - Most modern processors have multi-stage *instruction pipelines*
      - Enables execution of *sequence of instructions* to be *overlapped*
    - Instructions are *reordered* to maximize pipelining efficacy
  - *Superscalar* processors up the ante
    - Can issue multiple instructions at a time
    - Instructions are *grouped* for simultaneous dispatch
      - ("data dependency" among instructions → a major roadblock)

Parallelism Unifies "Non-clock" Routes

- Lowering $N \downarrow I$ and CPI:
  - Various ways at various levels → classified by some as follows
    - Bit level
    - Instruction level
    - Data level
    - Task level
- Data level parallelism:
  - Break down program code into parts that are processed in parallel
    - Most expediently is parallelism inherent in program *loops*
      - Often involves applying *same calculations* to parts of big data structure
    - May (less expediently) also be just *independent sections* of program code
  - E.g. → computing matrix product **C** = **A** x **B**:
    - Divide **C** into quadrants
    - Have 4 processors compute the quadrants in parallel (1 quadrant each)
    - Product can be obtained in close to 4 times as fast

Parallelism Unifies "Non-clock" Routes

- Lowering $N \downarrow I$ and CPI:
  - Various ways at various levels → classified by some as follows:
    - Bit level
    - Instruction level
    - Data level
    - Task level
- Task level parallelism:
  - Apply *different calculations* in parallel to same or different data
    - (as opposed to *same calculations* seen in data level parallelism)
  - E.g. → lab aiming to handle more (typically diverse) jobs at a time
    - Can install more systems so more jobs can be running at any one time
  - Concerned more with *units_of_work/time* than *time/unit_of_work*
    - Preceding e.g. → more with *jobs per day* than *time to complete each job*
  - Pertains more to performance at community (collective) level

Parallel Computing has its Limit

- *Parallel* computing:
  - …
  - Problem broken into parts that can be solved *in parallel*
    - If this can't be done to a useful extent → little use for parallel computing
      - Parallel computing challenge → develop/implement parallelizing ideas
  - …
- **Amdahl's** law:
  - Gives *commonsense ceiling* on achievable performance gain
    - Speedup a better way can give → limited by the usability of the better way
  - Quantitatively:

$$1/Speedup{\downarrow}MaxAchievable = \%Usable/Speedup{\downarrow}Bette$$

Memory Aid

Parallels formula for 2 resistors in parallel:

$$1/R{\downarrow}Total = 1/R{\downarrow}1 + 1/R{\downarrow}2$$

$$Speedup{\downarrow}MaxAchievable = Speedup{\downarrow}BetterWay / \%Usable$$

Note: Speedup is in terms of "# of times as fast". %Unusable = (1 - %Usable)

N.B.

- Trends (manifested in ever faster networks, distributed systems, and multi-processor computers) of the past many years point to "*future of computing lies in parallelism*".
  - Current supercomputers are all parallel computers.
  - *Parallel computers* entered mainstream computing with the advent of *multicore computers.*
- Without *efficient parallel algorithms*, a parallel computer is like Austin Powers loses his mojo!
- It's *harder to write parallel programs* than sequential ones:
  - There must be *even distribution of work* over processors.
  - Amount of *inter-processor communication* must be curbed.
  - If not programmed in *portable fashion,* may run fast on certain architectures but surprisingly slow on others.
- How'd it be like if there're to be no *parallel/concurrent* activities?
  - (Less mind-boggling warm-up: How'd it be like if there's no gravity?)

Coaxing Mr. Quick & Mr. Parallel to Work Wonders

- Trends (manifested in ever faster networks, distributed systems, and multi-processor computers) of the past many years point to "*future of computing lies in parallelism*"
  - …
- Without *efficient parallel algorithms*, a parallel computer is like Austin Powers loses his mojo!

- How safe is quicksort's mojo in the world of parallel computing?
  - *Divide-and-conquer* nature → feels right at home
  - *Individual in-place partitioning* → doesn't seem to sit well with Amdahl
    - (where parallelizing challenge lies?)
- Trails on parallelizing quicksort → rather well-trodden
  - Only selective sampling of these for our study tour
    - A rather naïve algorithm for *shared-memory* processors
    - Algorithm 1 for *distributed-memory* processors
    - Algorithm 2 for *distributed-memory* processors

A Rather Naïve Algorithm for Shared-Memory Processors

Quick take on *shared-memory* architecture:

- *Multiple processors access* common pool of memory via 1 shared bus
- Higher risk of running into bus contention
- Multiple processes (one per processor) execute in parallel
  - *Items* to be sorted → stored in *global array*
  - *Global stack* stores *indices* of sub-arrays (unsorted in general)
- Each idling process checks whether stack is nonempty and if so…
  - Pops indices of a sub-array
  - Done (returns) if size of sub-array is 1 (already sorted), otherwise.. … partitions sub-array into 2 and …
    - Pushes indices of 1 partitioned sub-array (unsorted in general) onto stack
    - Apply *same treatment* to the other sub-array (unsorted in general)

(Note: Each process locks/unlocks stack before/after pushing or popping.)

A Rather Naïve Algorithm for Shared-Memory Processors

- Can achieve only rather mediocre speedup:
  - Analytical estimates:
    - 1023 data items, 8 processors: less than 4
    - 65535 data items, 16 processors: less than 6
  - Observed speedups on an actual machine: worse off
- Most important reason:
  - Low amount of parallelism early in the algorithm's execution
    - Partitioning of original unsorted array → significant *sequential* component
    - Limits speedup achievable (Amdahl's law), regardless of # of processors
  - Cold start → low or no parallelism early, high parallelism near the end
- Another factor:
  - Shared global stack causes contention among processors

Definition of "Sorted" for Distributed-Memory Processors

Quick take on *distributed-memory* architecture:

- Each processor has access to …
  - *Local* memory (with the processor itself) → fast access
  - *Remote* memory (with other processors) → slower access

- Let there be
  - *p* processors labeled P_0 , P_1 , …, P_{i-1}
  - Items to be sorted are distributed over the *p* processors
- Condition for being sorted (*non-decreasing* order presumed):
  - 1 Every processor's list of items is sorted
  - 2 For every pair of processors P_i and P_{i+1} :
    - Last (largest) item on P_i's list …

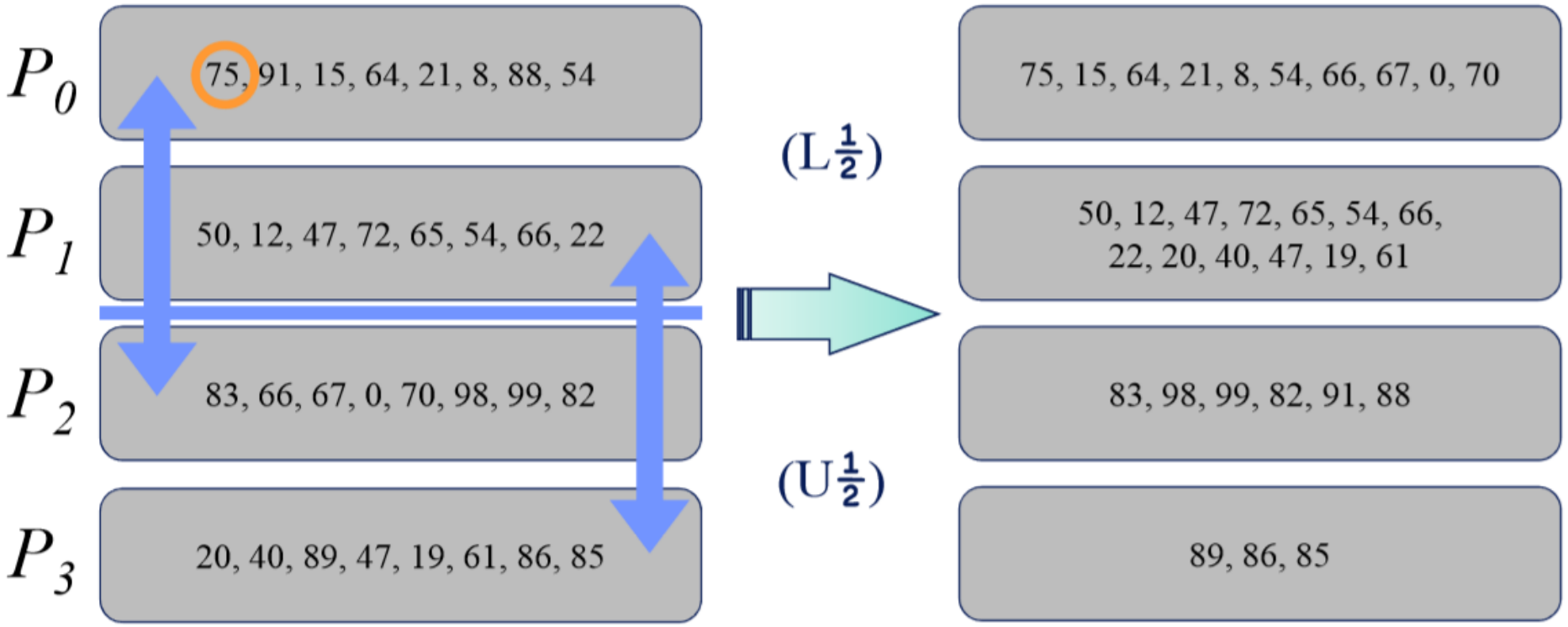
    … is less than or equal to …

    … first (smallest) item on P_{i+1} 's list

(Note 1: i = 0, 1, 2, …, p – 2)

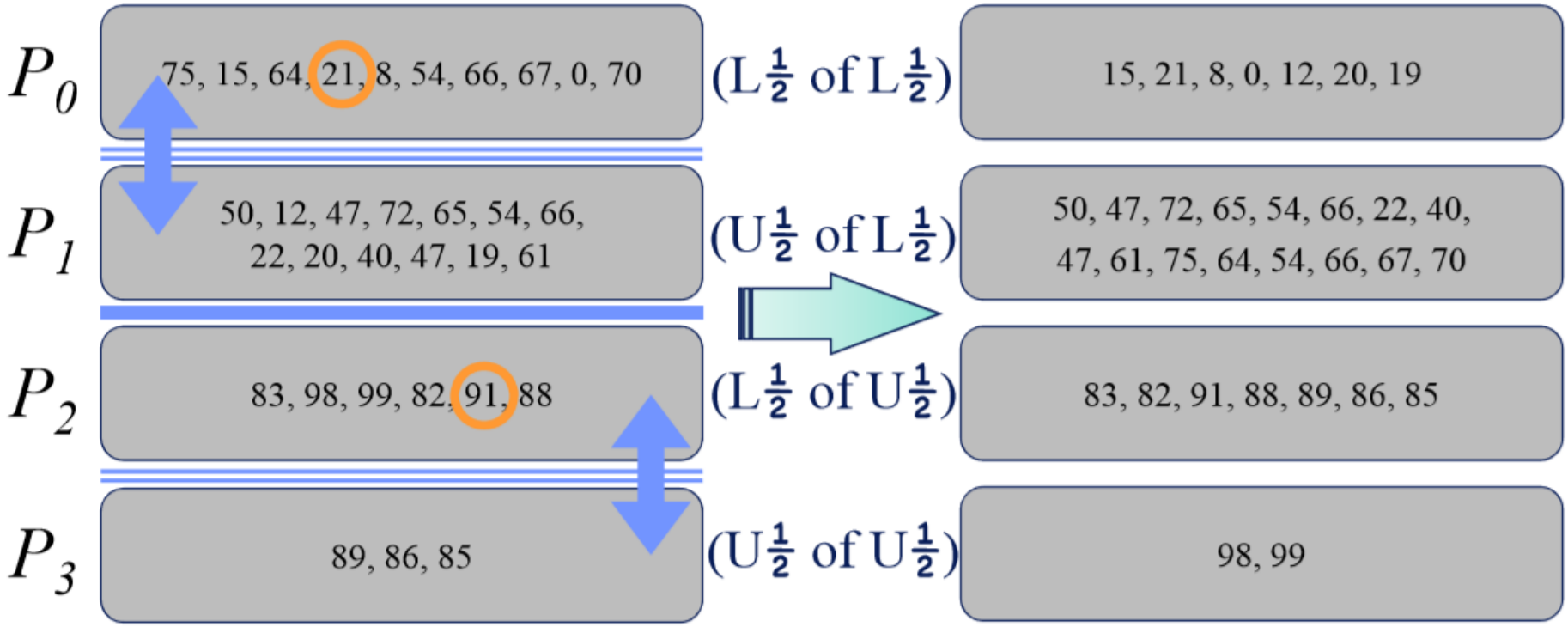
(Note 2: sorted items need not be evenly distributed over the *p* processors)

Algorithm 1 for Distributed-Memory Processors



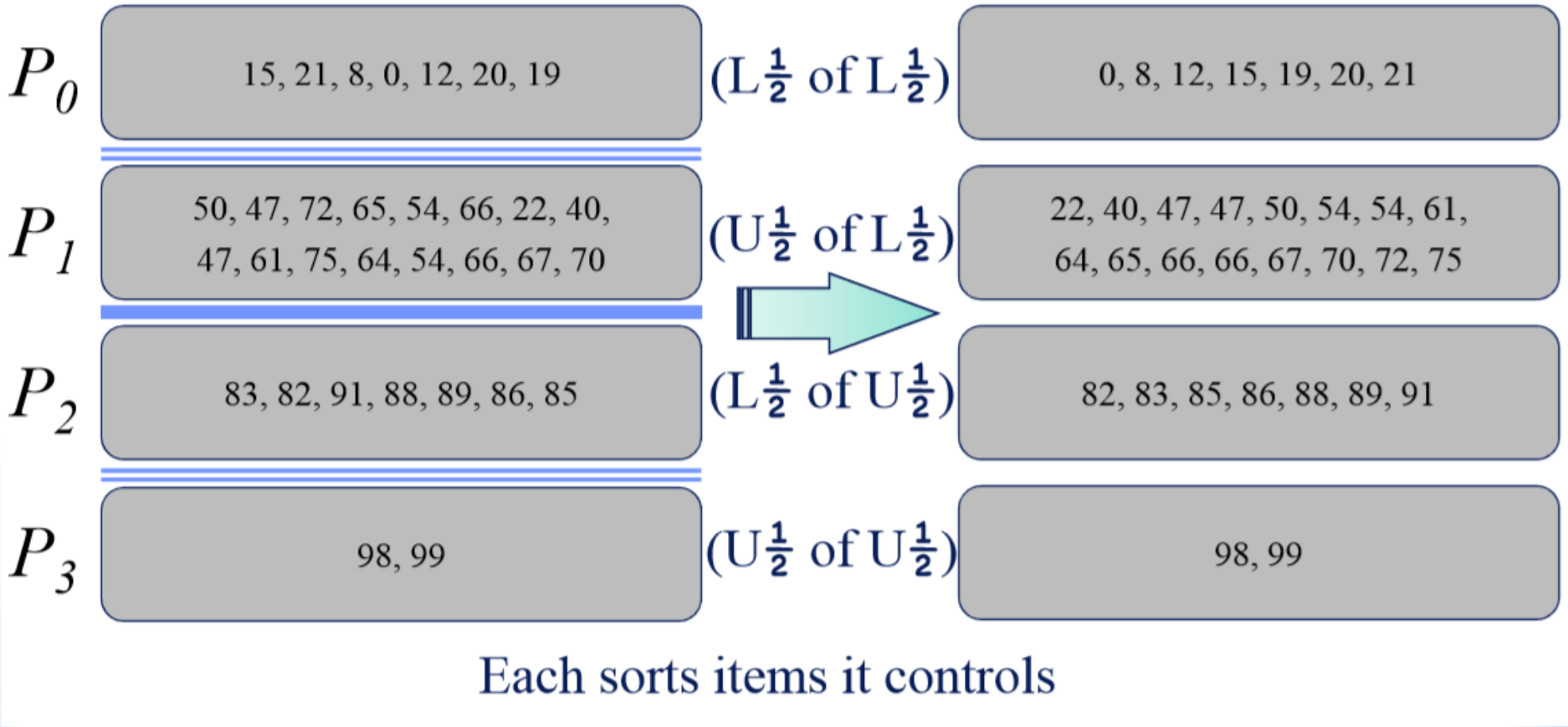
P_0 chooses and broadcasts pivot
Partners exchange "$L\frac{1}{2}$", "$U\frac{1}{2}$" items

Algorithm 1 for Distributed-Memory Processors



P_0 and P_2 choose and broadcast pivots
Partners exchange "$L\frac{1}{2}$", "$U\frac{1}{2}$" items

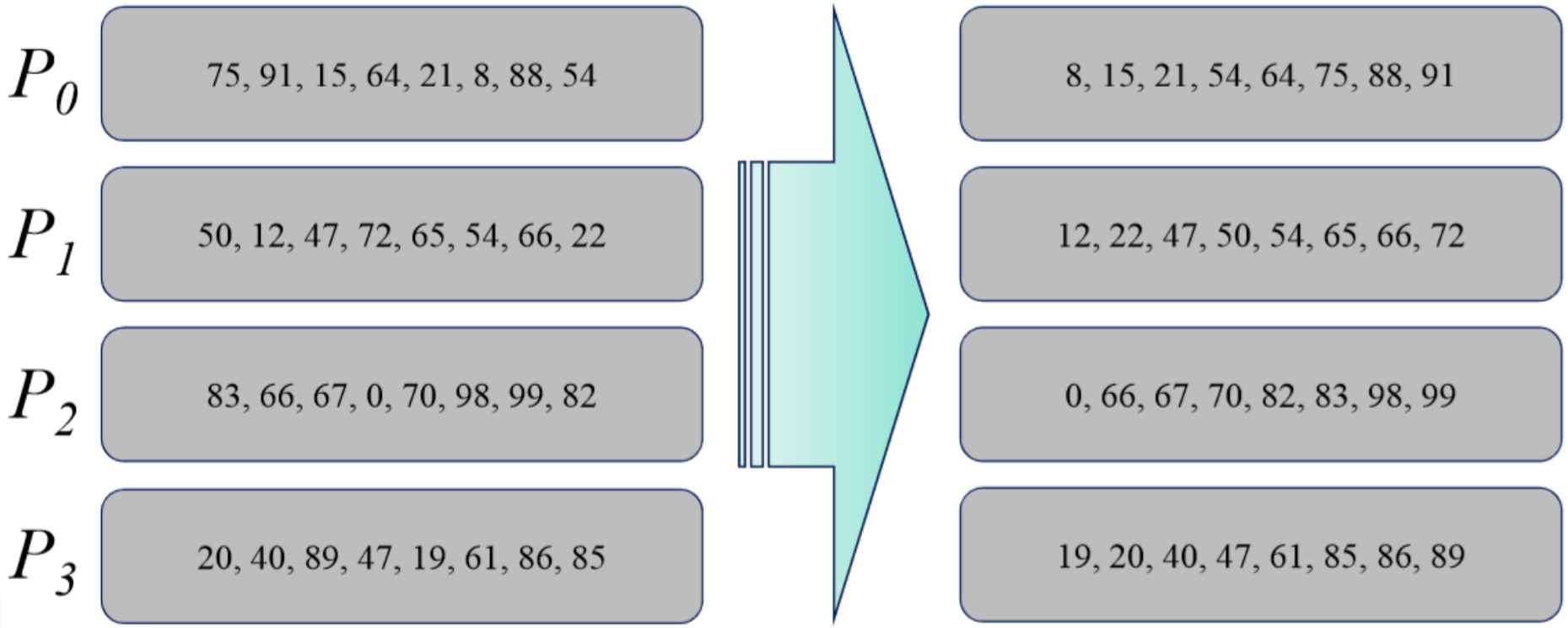
Algorithm 1 for Distributed-Memory Processors



Algorithm 1 for Distributed-Memory Processors

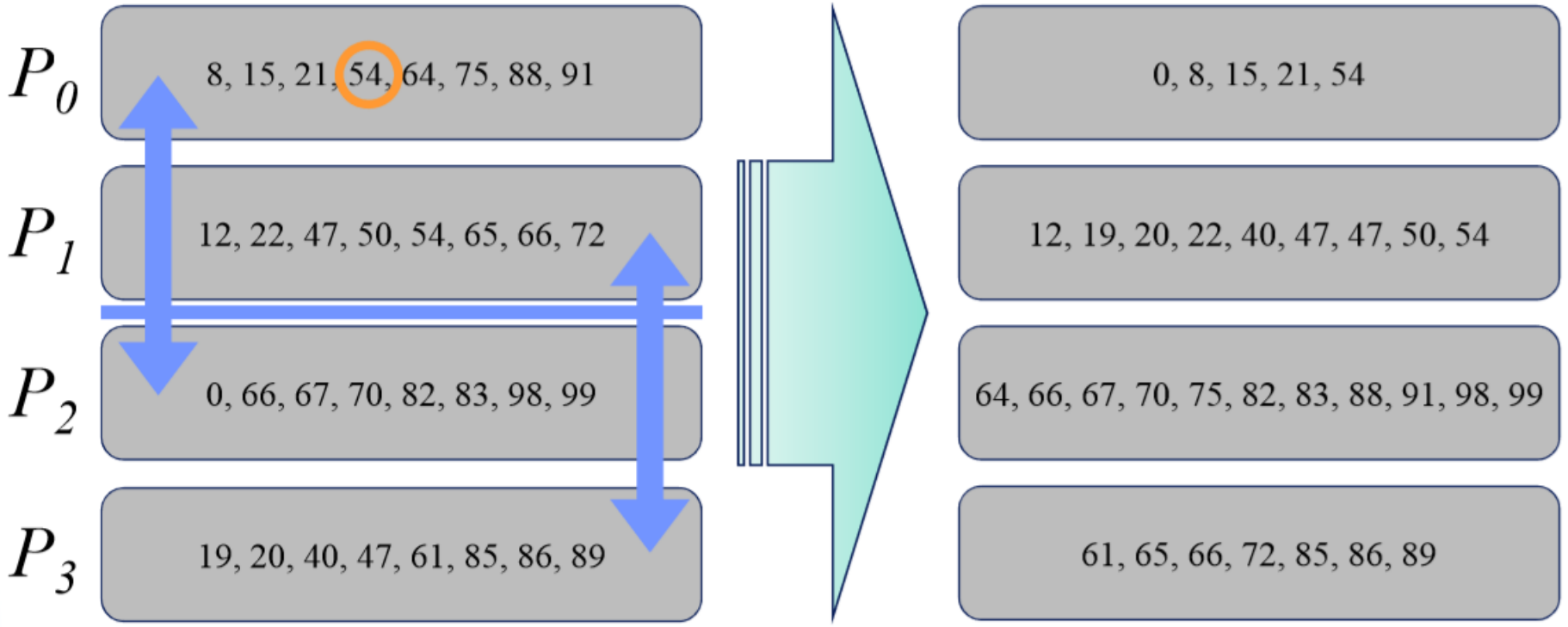
- Observations:
  - Execution time dictated by when last processor completes
  - Likely to do poor job at job balancing
    - Number of elements each processor must sort
  - Low likelihood for pivot to be true median (or near it)
    - (recall: good quicksort performance obtained if median used as pivot)
- Improvements?
  - Better job balancing?
    - (each processor sort an evenly distributed # of items?)
  - Better pivot?
    - (better likelihood to be true median or something closer to true median?)

Algorithm 2 for Distributed-Memory Processors



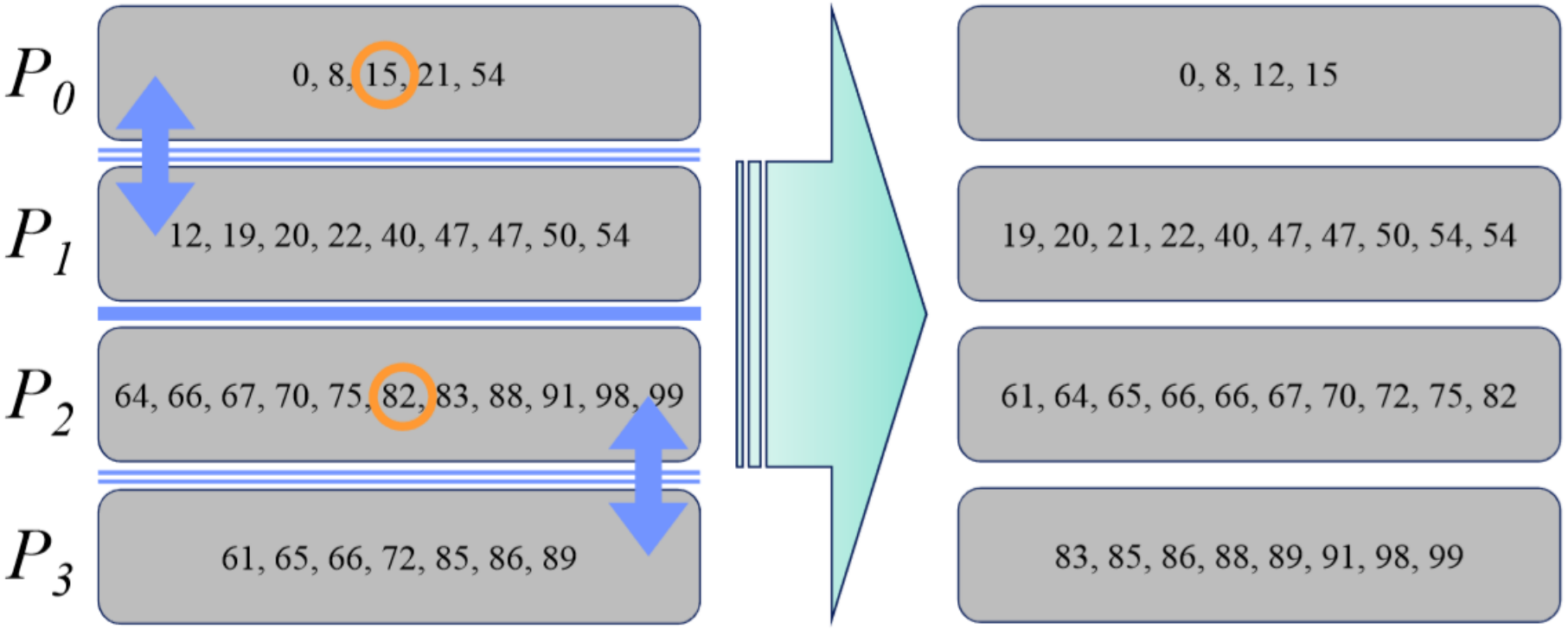
Each sorts items it controls

Algorithm 2 for Distributed-Memory Processors



P_0 broadcasts its median
Partners exchange "low", "high" lists
Each merges kept and received items

Algorithm 2 for Distributed-Memory Processors



P_0 and P_2 broadcast medians
Partners exchange "low", "high" lists
Each merges kept and received items

Algorithm 2 for Distributed-Memory Processors

- Observations:
  - Starts where **Algorithm 1** ends
    - Each processor sorts items it controls
    - When done → "sortedness" condition 1 is met
  - Processes must still exchange items
    - To meet "sortedness" condition 2
  - Process can use median of its sorted items as pivot
    - This is much more likely to be close to true median
- Even better pivot?
  - Median of medians?
    - (empirically found to be not attractive)
    - (incurs extra communication costs → offsets any computational gains)

Acknowledgments

- A major part of this note is derived from the material of others
  - Material in one form or another
  - Especially material that has been made available online
- In the spirit of parallelism, gratitude is hereby conveyed *in parallel*
  - To each and every one whose material has been exploited
    - In one way or another (for educational purpose)
- A closing example of “parallelism exploitation” it may well serve!