

Parallel and distributed computing

Conf. Dr. ing. Anca Hangan
Department of Computers
Anca.Hangan@cs.utcluj.ro

Content

- Major topics: parallel computing, distributed computing
- Parallel computing
 - Platforms, models
 - Parallel algorithms design
 - Improving the performance of parallel programs
 - Workload-driven evaluation models for parallel programs
 - Cache coherence
 - MapReduce, Hadoop, Grid

Content

- Distributed computing:
 - Time: clock synchronization, order of events
 - Mutual exclusion
 - Leader election
- Invited lectures (March 17th): Prof. Zsolt Istvan, TU Darmstadt

Bibliography

- Parallel Computing
 - (Grama) Introduction to Parallel Computing, A. Grama, A. Gupta, G. Karpypis, V. Kumar, 2003
 - (Culler) Parallel Computer Architecture: A Hardware / Software Approach, D.E. Culler, J.P. Singh, A. Gupta, Morgan Kaufman Publishers, 1999
- Distributed Computing
 - (Kshemkalayani) Distributed Computing: Principles, Algorithms, and Systems, A. D. Kshemkalayani, M. Singhal, Cambridge University Press, 2008
 - (Coulouris) Distributed Systems. Concepts and Design, G. Coulouris, J. Dollimore, T. Kindberg, Addison-Wesley, Fifth Edition, 2012

Rules and grading

- Enroll in Moodle course, Teams
- Lab attendance: 100%
- Grading – earn points!
 - Lab assignments + tests: 4op
 - Exam: 5op
 - Course quizzes, attendance: 10p-15p
- To pass the exam: at least 50p (of which at least 25p from exam)

Definitions

- **Parallel computer:** “collection of processing elements that communicate and cooperate to solve large problems fast” (Gottlieb et al 1989)
- **Parallel computing:** “a type of computation in which many calculations or the execution of processes are carried out simultaneously” (Gottlieb et al 1989)

Definitions

■ Distributed system

- “one in which several autonomous processors and data stores supporting processes and/or databases interact in order to cooperate to achieve an overall goal. The processes coordinate their activities and exchange information by means of information transfer over a **communication network**.” (Sloman and Kramer 1987)
- “a collection of independent computers that appear to the user of the system as a single computer” (Tanenbaum 1995)
- “one in which hardware or software components located at **networked computers** communicate and coordinate their actions only by passing messages” (Coulouris et al 2001)

Definitions

- **Concurrent** system: “one where a computation can advance without waiting for all other computations to complete”
(Silberschatz)
- **Concurrent** computations are executed during overlapping time periods

Concurrent, Parallel, Distributed

- The terms overlap, there is no clear distinction between them
- Concurrent tasks can be executed
 - on a single processor that supports multi-tasking (no parallelism!)
 - on a parallel computer
 - on computers connected through a network

Parallel computing

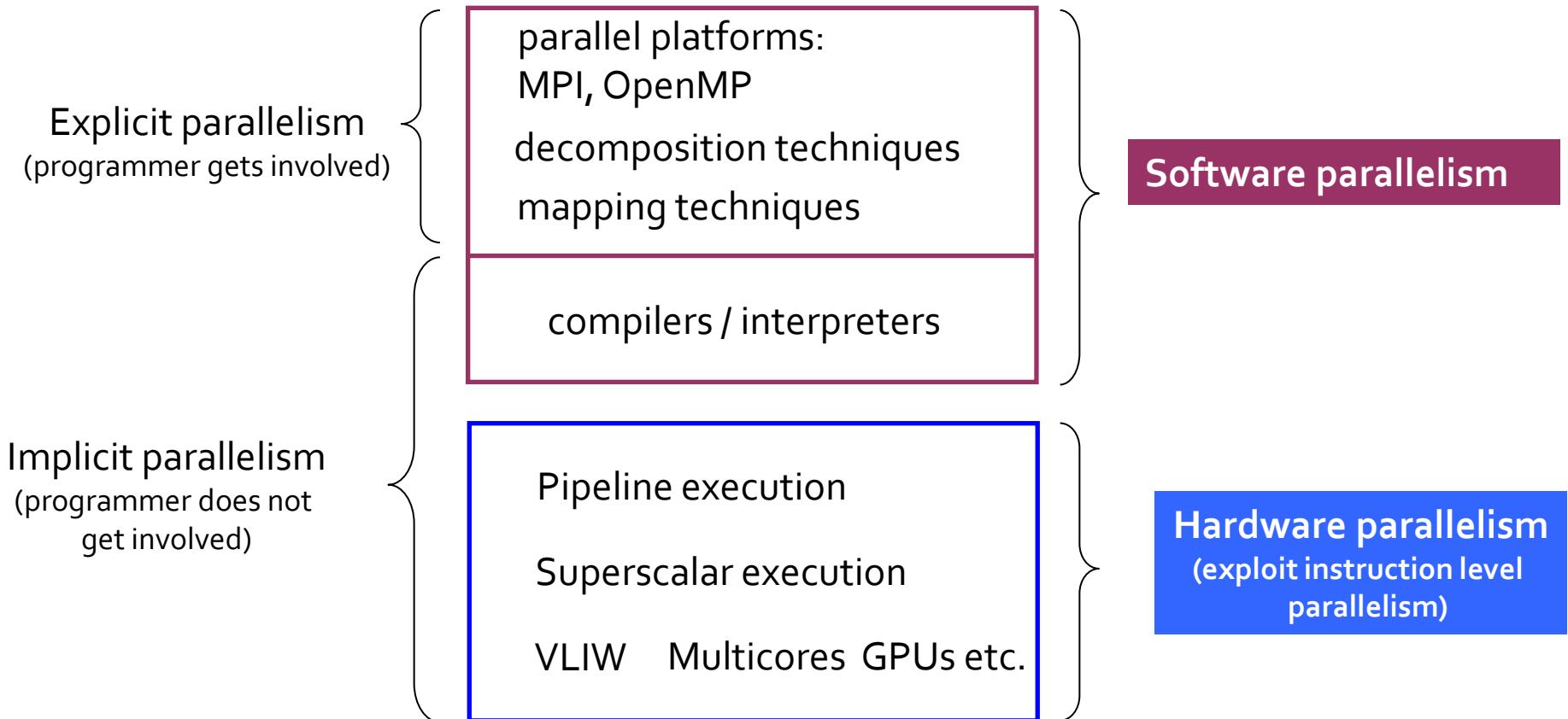
Why parallel computing?

- Solve problems faster
 - Increase processor frequency (Moore's law)
 - Pipeline and superscalar processor architectures
 - Multicores / Multiprocessors
- Off-the-shelf parallel computers at a reasonable price
-> good reason to write parallel programs; rewrite sequential programs
- Solve really BIG problems in a reasonable time
 - Lots of data
 - Lots of computations
- Get a more accurate result in a reasonable time

Applications of parallel computing

- **Applications in engineering and design:** mathematical modeling, geometric modeling, ...
- **Scientific applications:** simulations, weather modeling, DNA sequencing, 3D plasma modeling, ...
- **Commercial applications:** transactions, data analysis, ...
- **Applications in computer systems:** image/signal processing, cryptography, security, ...

Levels & types of parallelism



Levels & types of parallelism

- Level of parallelism
 - Program / process level parallelism
 - Thread level parallelism
 - Instruction level parallelism
- Types of parallelism
 - Data parallelism
 - Control (functional) parallelism

Critical components of parallel computing

- From the programmer's perspective:
 - how to express parallel tasks?
 - how to specify the interaction between parallel tasks?

How to express parallel tasks?

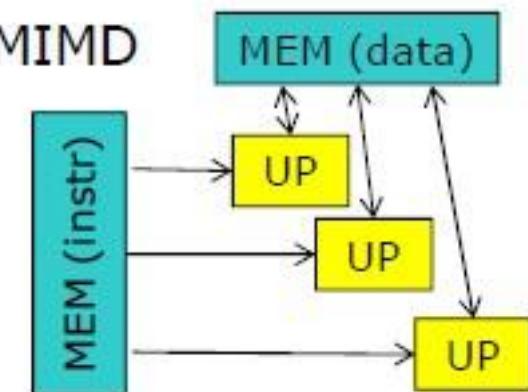
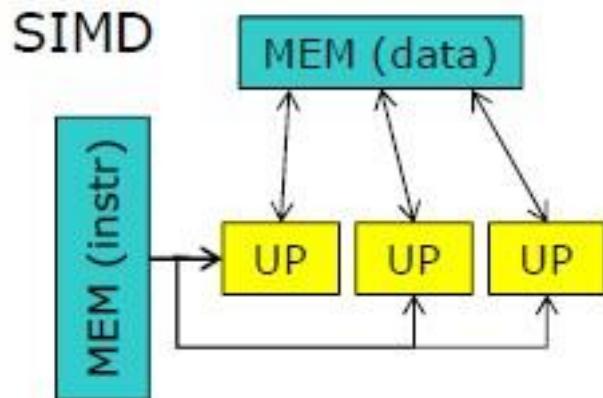
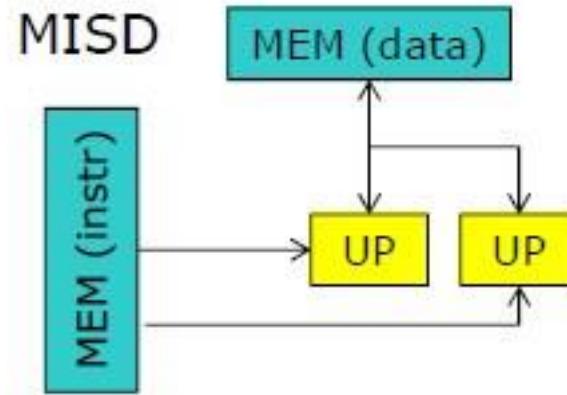
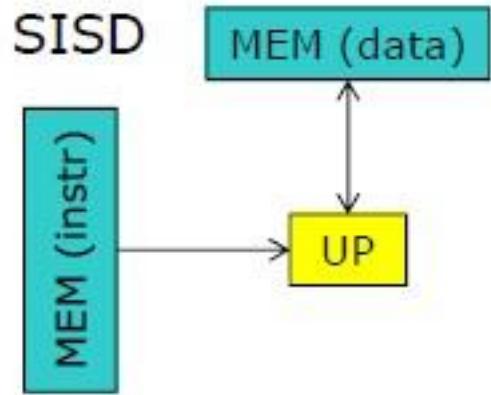
- each program can be viewed as a parallel task
- ...
- each instruction can be viewed as a parallel task

```
1  for (int i = 0; i < 1000; i++)  
2      c[i] = a[i] + b[i];
```

How to specify the interaction between parallel tasks?

- access a shared data space
- exchange messages

Parallel architectures classification – Flynn



Distributed computing

Why distributed computing?

- **Inherently distributed computations** e.g. money transfer
- **Resource sharing:** peripherals, data/databases, processing, ...
- **Access remote resources:** sensitive data that cannot be replicated , special (expensive) resources,...
- **Enhanced reliability:** many distributed resources are not likely to crash at the same time
- **Increased performance/cost ratio:** replace special, expensive, parallel machines with distributed computers (ex: Grids).

Examples of distributed systems

- Computer networks are everywhere and are support for a variety of services!
- Web search: billions of web pages, billions of searches per month
- Massively multiplayer online games: hundreds of thousands of users, fast response times
- Financial trading: real-time access to information sources, event detection

Models

- Architectural models
 - how are responsibilities distributed between system components
 - how these components are placed
- Interaction models
 - how to handle the time
 - time limits on process execution or message delivery?

Architectural models

- Client server
 - client calls a service on a server (by sending a request message to the server)
 - server does the work and sends the result back to the client
 - server can act as a client for other servers
 - issue: centralization (point of failure, bottleneck)
- Peer-to-peer
 - all processes are equal
 - every computer holds resources that are commonly shared
 - no bottleneck for processing and communication
 - issue: high complexity

Architectural models

- Service-oriented
 - Integrates distributed, separately maintained and deployed software components
 - Service = functionality that can be accessed remotely
 - Service consumers are not aware of the services' inner workings

Interaction models

- Synchronous
 - lower and upper bounds on the execution time of processes
 - messages are received within a known bounded time
 - drift rates between local clocks have a known bound
 - global physical time (with a certain precision)
 - predictable behavior in terms of timing (proper for hard real-time apps)
 - timeouts can be used to detect failures
 - difficult and costly to implement
- Asynchronous
 - **no** lower and upper bounds on the execution time of processes
 - messages are **not** received within a known bounded time
 - drift rates between local clocks do **not** have a known bound
 - **no** global physical time (logical time is needed)
 - unpredictable in terms of timing
 - timeout **cannot** be used for failures detections
 - widespread in practice

Finally...

Parallel vs Distributed

- Goal
 - parallel: solve a large problems faster, get more accurate results
 - distributed: cooperate to solve a problem (resource sharing, remote access, improve reliability)
- Processors types
 - parallel: identical, non-identical processing units
 - distributed: heterogeneous
- Memory
 - parallel: shared
 - distributed: not shared
- Geographical distribution
 - parallel: close to each other
 - distributed: far from each other

Parallel Computing Basics

Computer Architecture + Programming Models

Parallel computer architectures

Computer architecture evolution

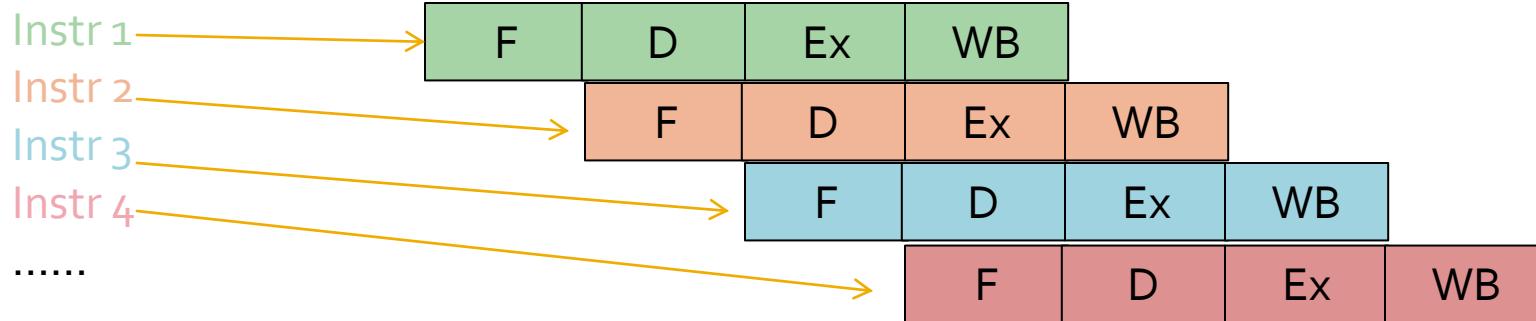
- Until 2004, performance gain through:
 - Frequency scaling (Moore's Law)
 - Instruction level parallelism (ILP)
- Since: 2004
 - Multicore CPUs
 - SIMD execution in GPUs

Processor: pre multicore

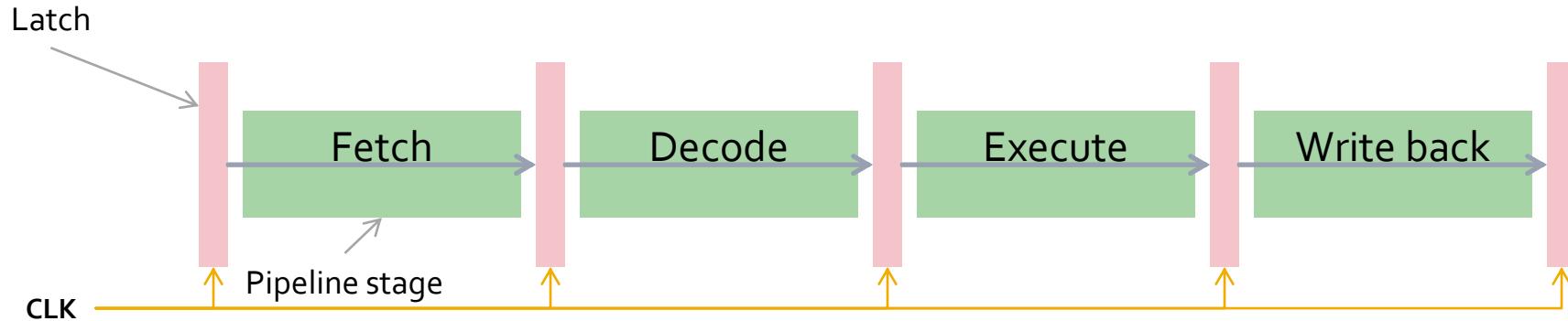
- Big L1 cache
- Memory pre-fetch
- Branch predictor
- Out-of-order execution
- Sophisticated processor logic – requires more transistors

Pipeline (ILP)

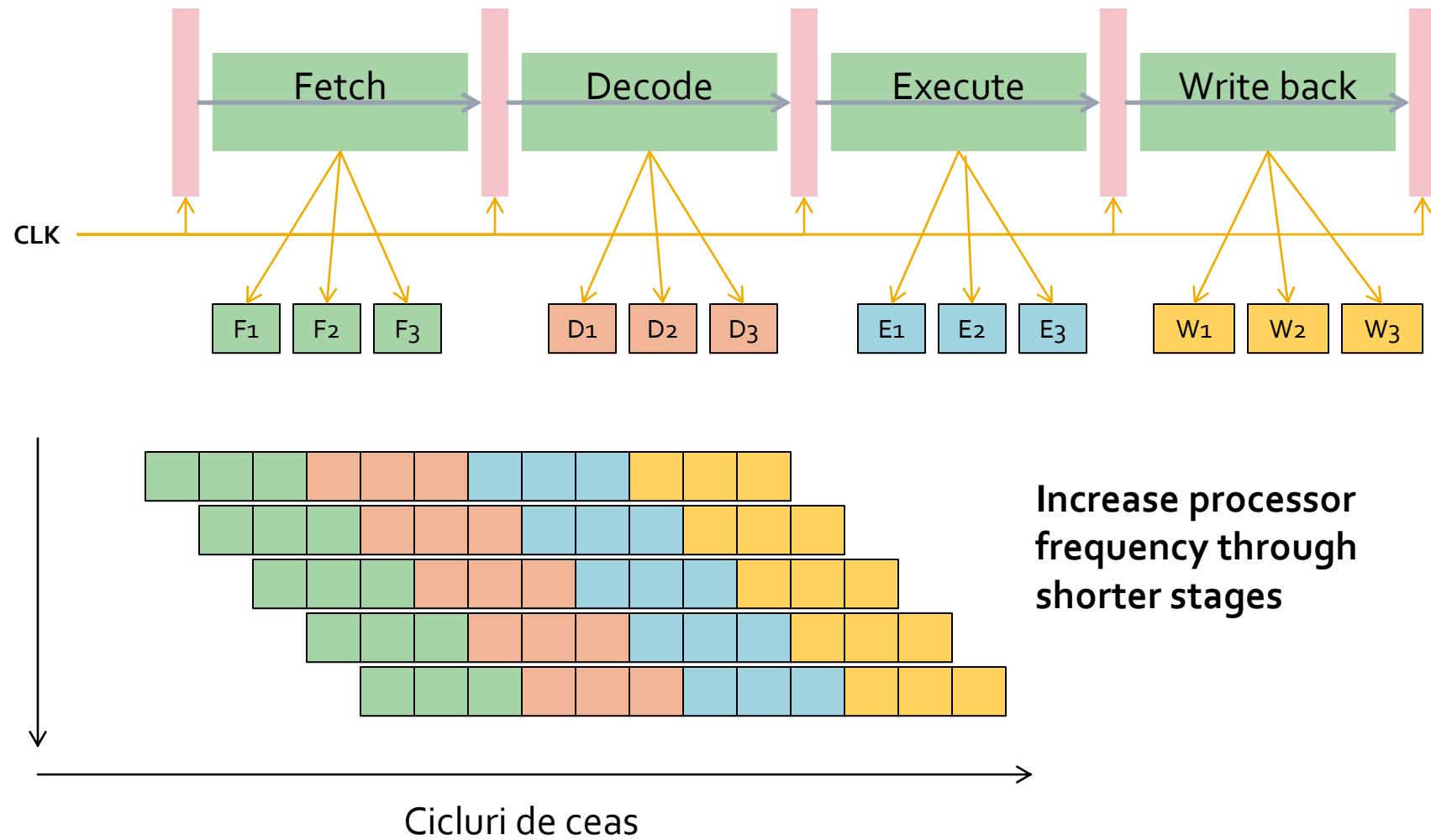
Program:



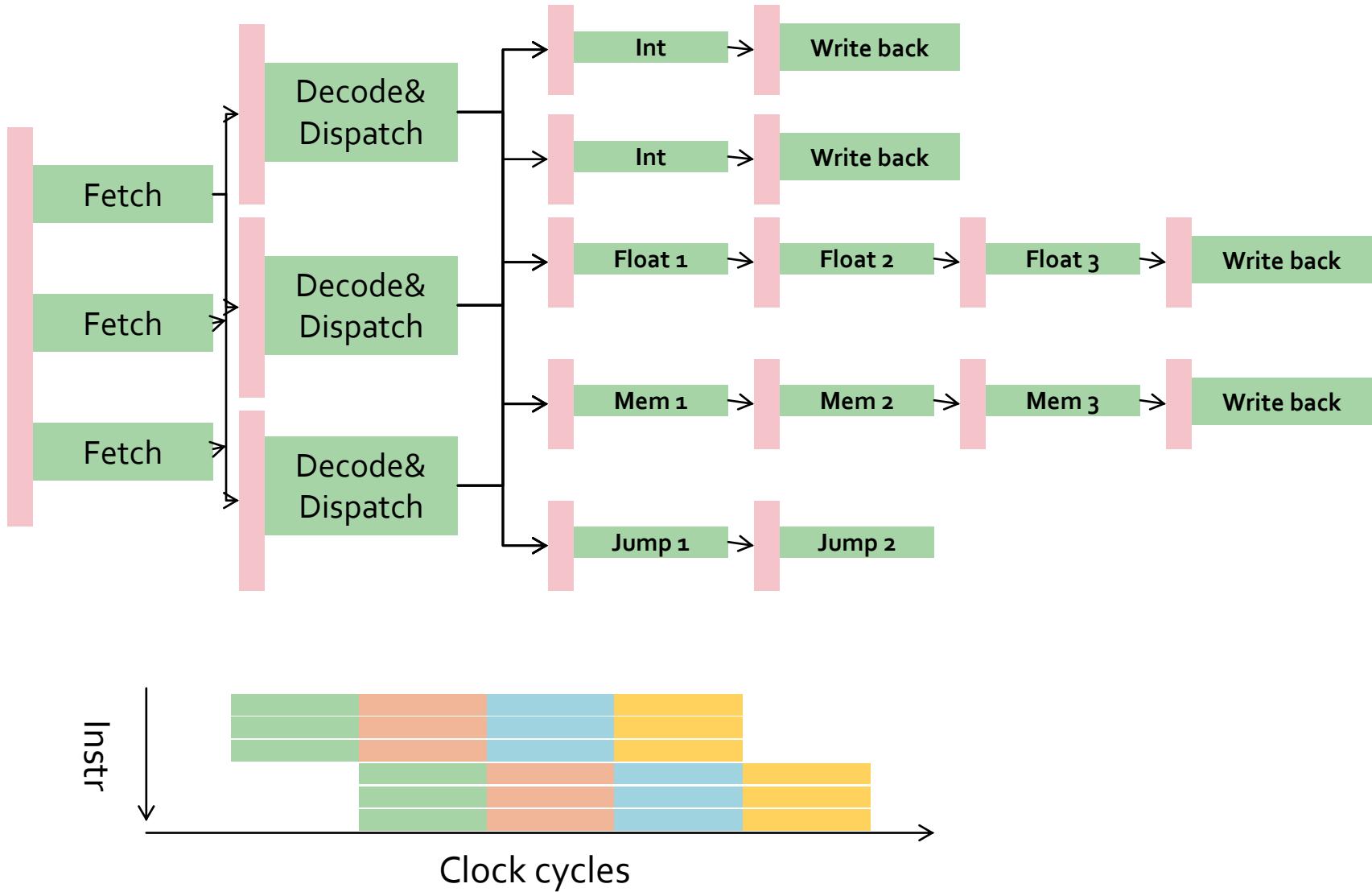
CPI=1? Hazard situations!



Superpipeline



Superscalar (ILP)



Parallel computers trends

- Multicore processors:
 - identical processing cores, integrated multicore GPU in the same chip (ex: Intel Skylake)
 - non-identical processing cores: (ex: Intel Alder Lake)
- GPUs: multiple processing blocks (ex: NVIDIA Maxwell GTX 980 GPU)
- Mobile parallel processing: multiple non-identical processing cores + multicore GPU (ex: Exynos 8890 -4 Cortex A53 cores, 4 Exynos M1 cores, 12 core Mali GPU)
- Parallelism and HW specialization: FPGAs
- Supercomputers: clusters of multi-core CPUs + GPUs

Multicores

- Use more transistors to add more cores to a processors
- Use less of that sophisticated processor logic => simpler cores
- Even if cores are slower, there is a potential speedup (ex: 2 x 25% slower cores -> speedup $2 * 0.75 = 1.5$)
- Programs
 - Issue: if the program doesn't express parallelism, there is NO speedup
 - Solution: use threads!

SIMD execution

- Add ALUs to increase compute capability
- SIMD computing: single instruction, multiple data (all ALUs execute the same instructions in parallel)
- Programs
 - Special instructions for SIMD processing (ex: SSE, AVX instructions)
 - Special language constructions (ex: parallel for) transformed by compiler in low level SIMD instructions
 - Issues: efficient only if the same instruction sequence applies to all elements operated upon simultaneously (**instruction stream coherence**)

Hardware-supported multi-threading

- Core manages execution contexts for multiple threads
 - Runs instructions from runnable threads
 - Makes decision about which thread to run each clock
- Interleaved multi-threading
 - each clock, the core chooses a thread, and runs an instruction from the thread on the ALUs
- Simultaneous multi-threading
 - Each clock, core chooses instructions from multiple threads to run on ALUs
 - Extension of superscalar architecture
 - Ex: Intel Hyperthreading
- Issues:
 - additional storage for thread context
 - Increased run time for threads

Forms of parallel execution in modern processors

- Multicores
 - Multiple execution cores
 - Thread level parallelism
- SIMD
 - Multiple ALUs that execute the same instruction stream
 - Efficient design for data-parallel workloads
 - Dependencies have to be known and specified beforehand
- Superscalar
 - Exploit ILP within an instruction stream
 - Parallelism automatically and dynamically discovered by the hardware during execution (not programmer visible)
- Hardware-supported multi-threading
 - Context of threads managed in HW
 - Execute instructions from several threads at the same time

Heterogeneous Parallelism

- Scheduling problem: map processes on matching processing components
- Programming: design algorithms that decompose into components that match well with available resources
- HW design: establish a good mixture of resources for better performance

Parallel programming models

Speedup

- Main motivation of parallel processing:
achieve speedup

speedup (using P processors)=

$$\frac{\text{Execution time (1 processor)}}{\text{Execution time (P processors)}}$$

Cumpute in parallel

- Sum of numbers in an array
- $\text{Nums}=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$
- Master-slave model & message passing

- $P=5$ processors (students)
- $P=3$ processors
- $P=1$ processor

- Analyze execution times

Issues of parallel programming

- Observations
 - Communication time limits speedup
 - Imbalance in work assignment limits speedup
- Goals for designing parallel programs
 - Decomposing work into pieces that can safely be performed in parallel
 - Assigning work to processors
 - Managing communication/synchronization between the processors so that it does not limit speedup

Programming model

- Provides way of thinking about the structure of programs
- Parallel programs:
 - Abstractions for describing concurrent, parallel, or independent computation: threads, specific language constructions
 - Abstractions for describing communication
 - Shared address space
 - Message passing
 - Data parallel

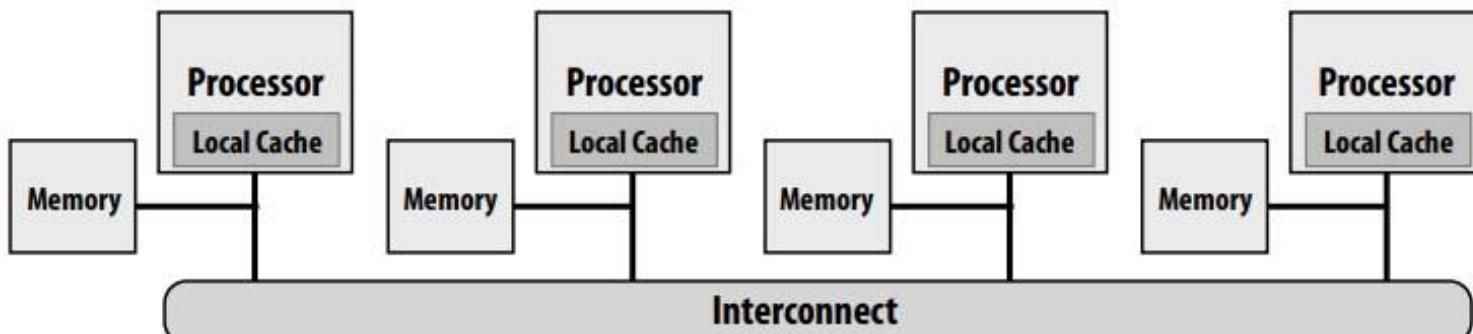
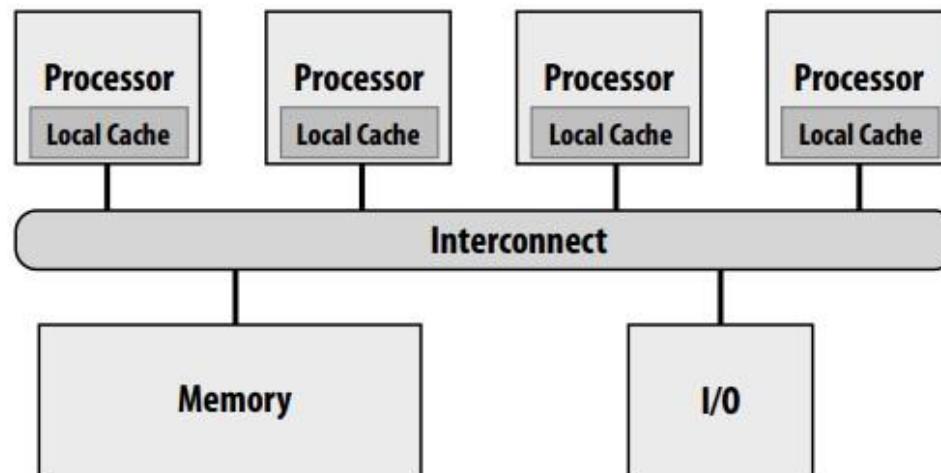
Shared address space model

- Threads communicate reading/writing to shared variables
- Ensure mutual exclusion via use of locks
- Natural extension of sequential programming

- HW implementation:
 - UMA - uniform memory access time
 - NUMA- non-uniform memory access time
 - COMA – cache only memory architecture

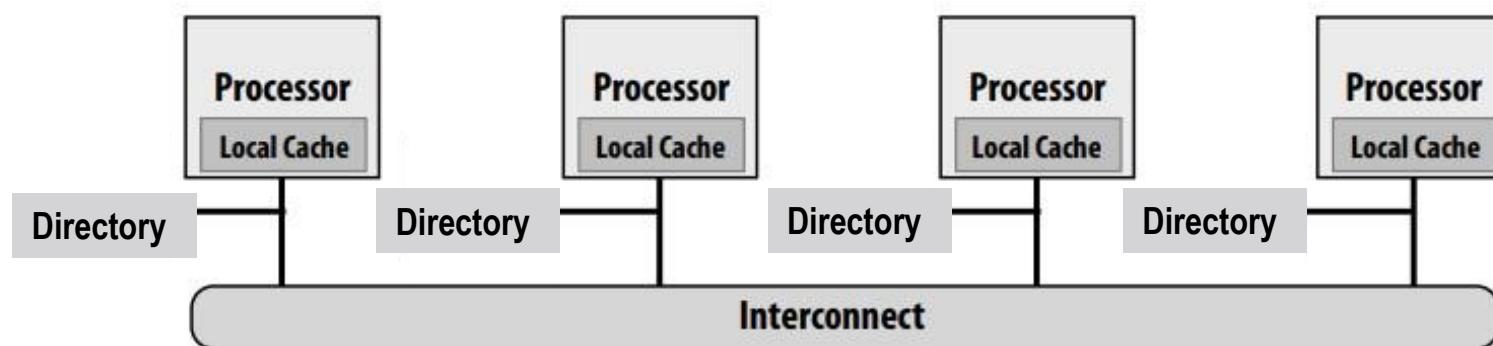
UMA, NUMA

UMA



NUMA

COMA

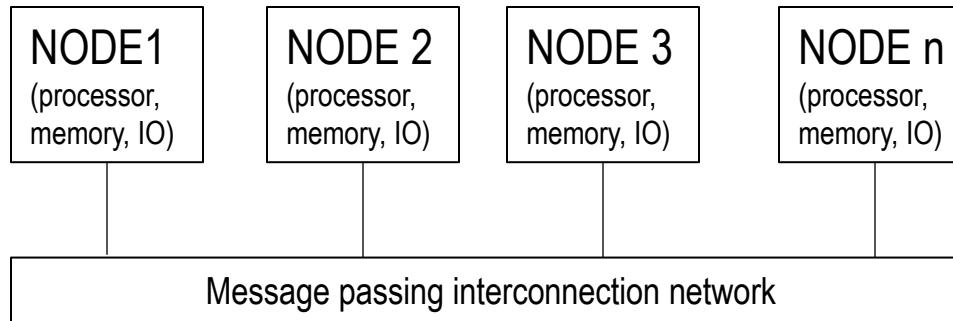


Message passing model

- Threads operate within their own private address spaces
- Threads communicate by sending/receiving messages
 - send: recipient, buffer to be transmitted and message ID
 - receive: sender, buffer to store data and message ID
- HW implementation: computing nodes able to communicate through messages

No-Remote Memory Access

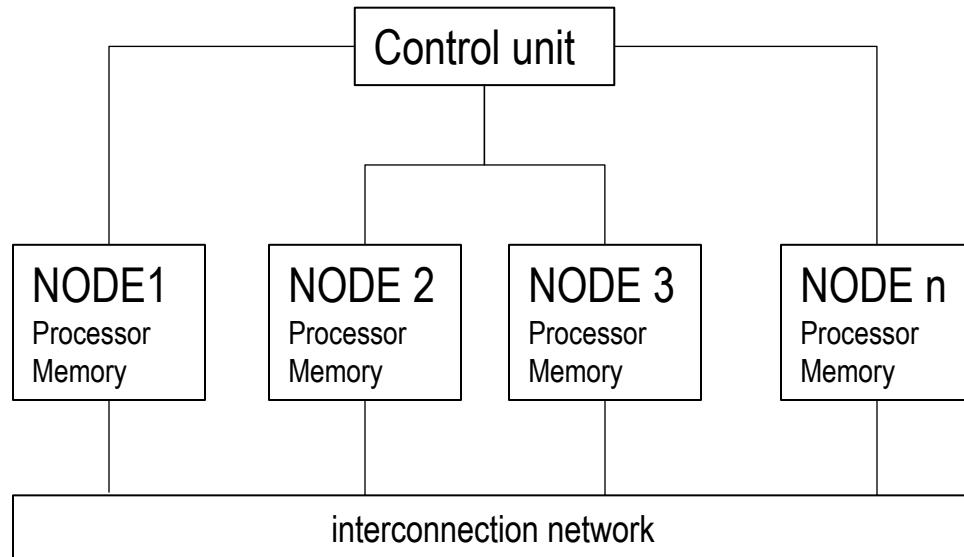
NORMA



Data parallel model

- Programs perform same operation on different data elements in a collection (ex: MAP - same function on each element of an array)
- Assumes shared address space
- Limits communication between iterations
- Stream programming: streams and kernels
- Communication: scatter/gather instructions

SIMD computers



To summarize

- Describe concurrent, parallel computations through:
 - Threads
 - Specific language constructions
- Programming models
 - Threads use shared variables
 - Threads/processes communicate through messages
 - Structure computation as a map over a collection of data, limit communication

Additional bibliography

- CMU Lectures:

[http://15418.courses.cs.cmu.edu/fall2017/lecture
/basicarch](http://15418.courses.cs.cmu.edu/fall2017/lecture/basicarch)

[http://15418.courses.cs.cmu.edu/fall2017content
/lectures/03_progmodels/03_progmodels_slides.
pdf](http://15418.courses.cs.cmu.edu/fall2017/content/lectures/03_progmodels/03_progmodels_slides.pdf)

- [http://www.lighterra.com/papers/modernmicrop
rocessors/](http://www.lighterra.com/papers/modernmicroprocessors/)

Parallel algorithm design

Parallel execution in modern processors

- Multicores
 - Multiple execution cores
 - Thread level parallelism
- SIMD
 - Multiple ALUs that execute the same instruction stream
 - Efficient design for data-parallel workloads
- Superscalar
 - Exploit ILP within an instruction stream
 - Parallelism automatically and dynamically discovered by the hardware during execution
- Hardware-supported multi-threading
 - Context of threads managed in HW
 - Execute instructions from several threads at the same time

Parallel programming models

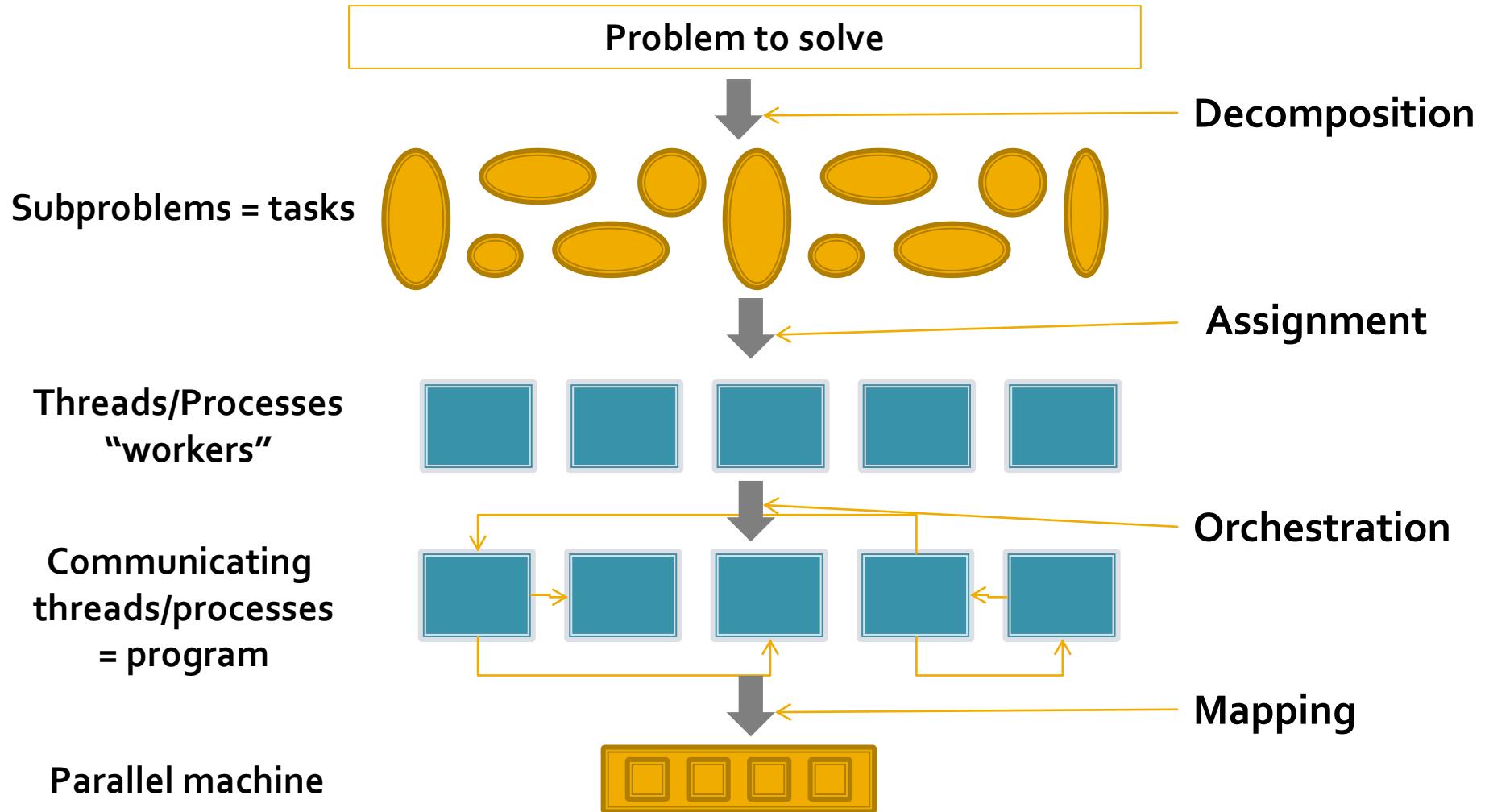
- Describe concurrent, parallel computations through:
 - Threads
 - Specific language constructions
- Programming models
 - **Shared address space:** threads use shared variables
 - **Message passing:** threads/processes communicate through messages
 - **Data parallel:** structure computation as a map over a collection of data, limit communication

Goals for designing parallel programs

- Decomposing work into pieces that can be performed in parallel
- Assigning work to processors
- Managing communication/synchronization between the processors
- Achieve speedup:

$$\text{speedup} = \frac{\text{Execution time (1 processor)}}{\text{Execution time (P processors)}}$$

Creating a parallel program



Concepts

- Task = piece of work done by the program
 - Smallest unit of concurrency
 - Granularity (amount of work): fine-grained; coarse-grained
- Thread/Process = abstract entity that performs tasks (“worker”)
 - Are assigned one or more tasks
 - Communicate and synchronize with one another
- Processor = executes threads/processes
 - Number of processes \neq number of processors

Parallelization process

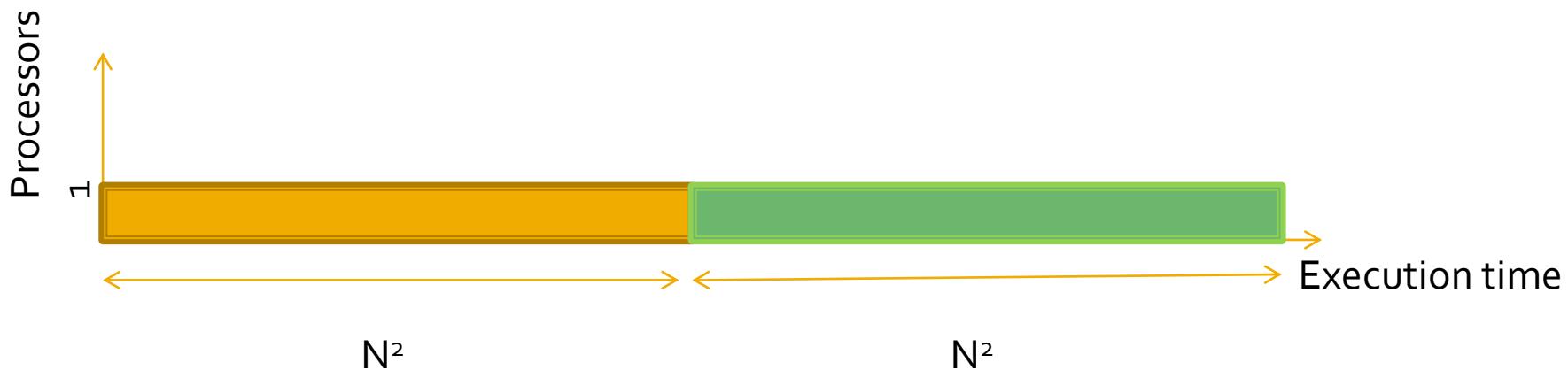
1. Decomposition
2. Assignment
3. Orchestration
4. Mapping

Decomposition

- Break up a computation into a collection of tasks
- When?
 - Statically
 - Tasks can be created dynamically
- How many?
 - At least enough tasks to occupy all execution units of the physical machine
 - Not too many => create large overhead
- Are there any dependencies?

Simple example

- 2 computations on a $N \times N$ matrix of numbers
 - Add 1 to all elements
 - Sum all the elements
- On one processor

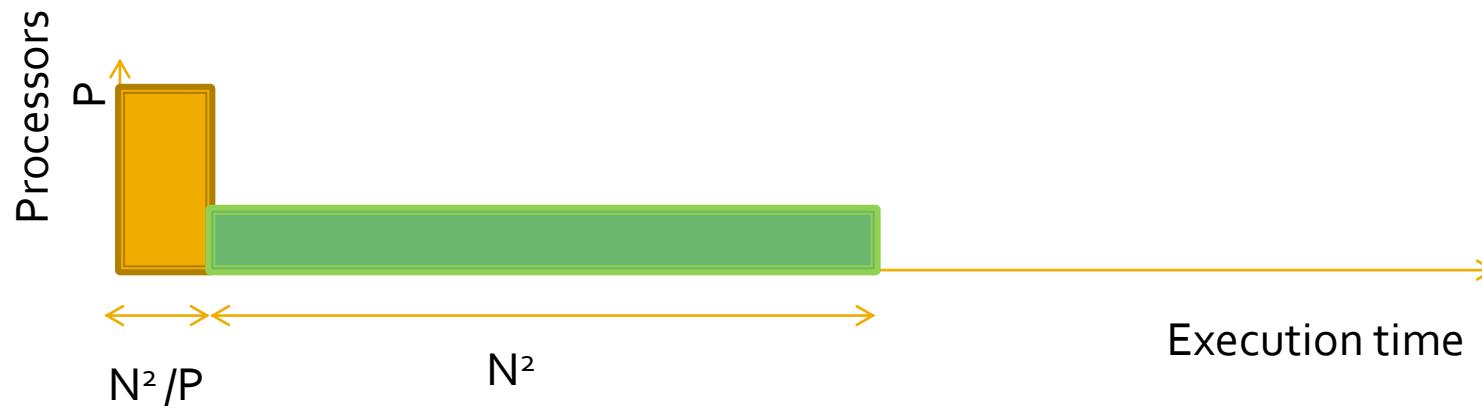


Simple example

- P processors
- Solution 1:
 - Assign N^2/P numbers to P processors
 - Step 1: all P processors perform 1st computation
 - Step 2: each processor adds each of its assigned numbers to a global sum
- Which is the problem?
- Compute the speedup!

Simple example

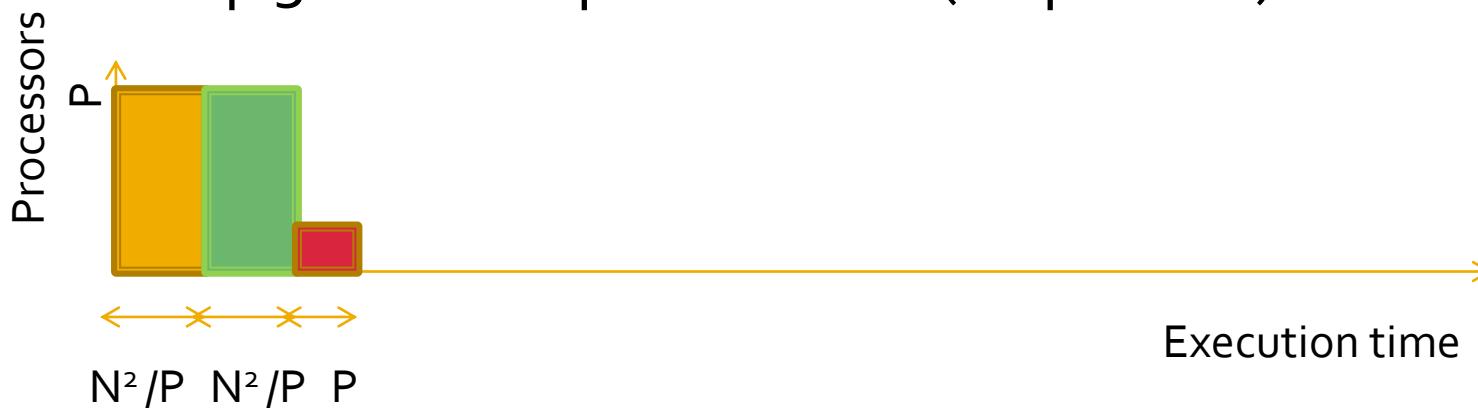
- Solution 1



- Parallel execution time: $N^2/P+N^2$

Simple example

- Solution 2:
 - Step2: Each processor computes the sum of assigned numbers locally (in parallel)
 - Step 3: Add the partial sums (sequential)



- Parallel execution time: $N^2 / P + N^2 / P + P$
- Compute the speedup!
- Recall Amdahl's law?

Assignment

- Tasks (“work to do”) are distributed to threads (“workers”)
- Goals:
 - Balance the workload among processes (load balancing)
 - Reduce communication between threads
 - Reduced overheads introduced by managing the assignment
- When?
 - Static
 - Dynamic

Partitioning

- Partitioning = Decomposition + Assignment
- Usually independent of the underlying architecture
- Who?
 - Decomposition: often done by programmer
 - Assignment: often in the responsibility of languages/runtimes

Orchestration

- Involves:
 - Structuring communication
 - Adding synchronization to preserve dependencies if necessary
 - Organizing data structures in memory
 - Scheduling tasks (order of task execution)
- Dependent of the architecture and programming model
- Goals: reduce comm/sync costs, preserve locality of data, reduce overhead

Mapping

- Map threads to actual HW execution units
- Who?
 - Operating system
 - Compiler
 - Hardware
 - Programmer?
- Example: to reduce communication cost , place cooperating threads on the same processor

Decompose computation or data?

- Many problems need parallelization because they process a lot of data
- Parallelize programs = partitioning computation?
- Partitioning data: it is equally valid!

Goals of the parallelization process

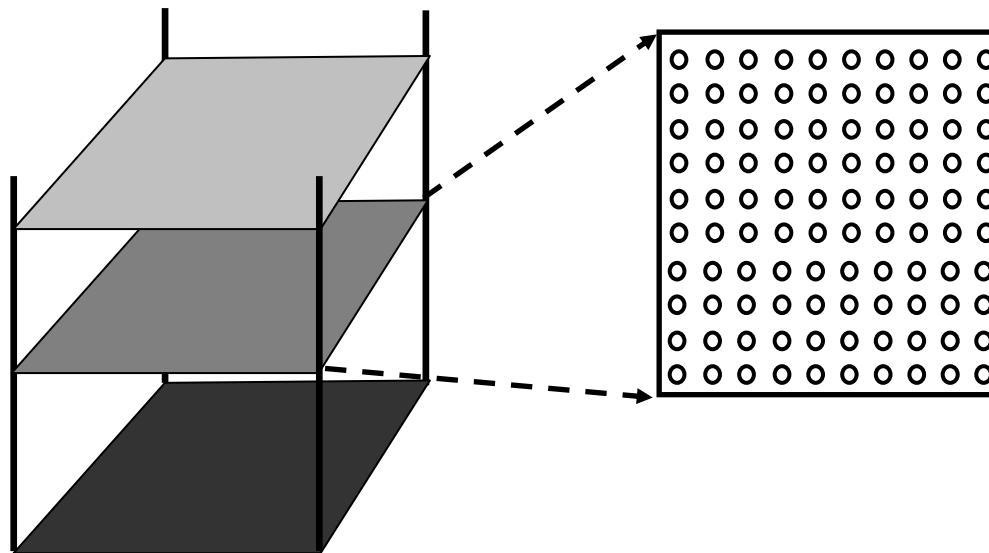
Step	Architecture-dependent?	Major performance goals
Decomposition	Mostly no	Expose enough concurrency but not too much
Assignment	Mostly no	Balance workload Reduce communication volume
Orchestration	Yes	Reduce non-inherent communication via data locality Reduce communication and synchronization cost as seen by the processor Reduce serialization at shared resources Schedule tasks to satisfy dependencies early
Mapping	Yes	Put related processes on the same processor if necessary Exploit locality in network topology

Example – Ocean currents simulation (from Culler et. al. - Ch 2)

- Simulate the motion of the water currents in the ocean
- Models of ocean behavior and water currents over time
- Problem is continuous in space and time → make problem discrete in space and time
 - How to make space be discrete?
 - How to make time be discrete?

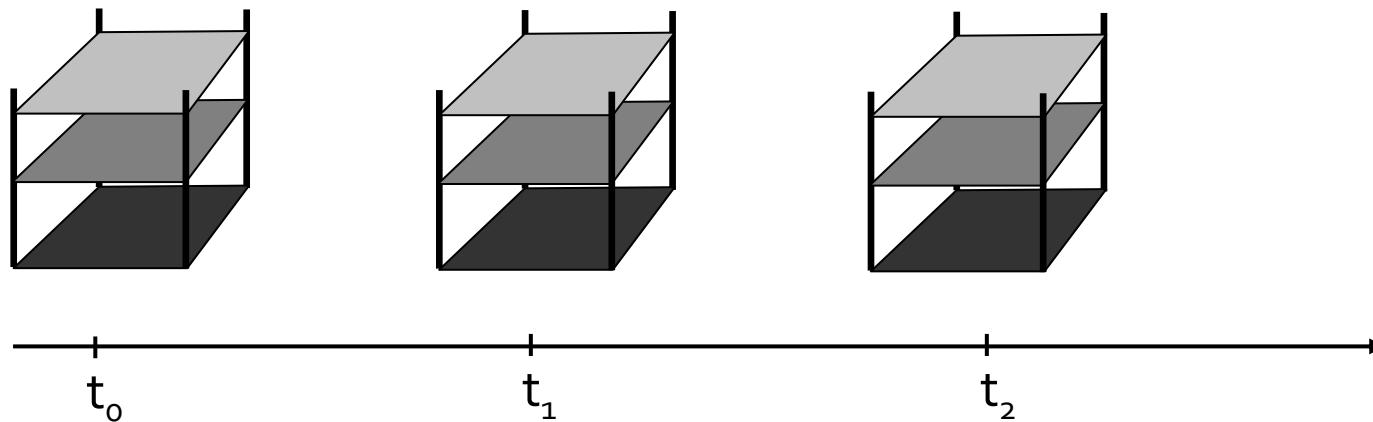
Ocean simulation

- How to make space be discrete?
 - model the ocean as a grid of points -> set of two-dimensional grids
 - every important variable (pressure, velocity, etc) has a value at each point



Ocean simulation

- How to make time be discrete?
 - time is made discrete into a series of finite time-steps
 - equations of motion are solved for each point in one time-step, the value of variables are updated, and the equations are solved again
 - every time-step consists of several operations (e.g. set up values for variables using the results from previous step, solve the equations, etc)



Ocean simulation

- The more grid points, the more accurate the simulation
 - Atlantic: 2 000 km x 2 000 km, using grids of 100 x 100 points implies 20 km between points
- The shorter interval between steps, the more accurate simulation
 - to simulate 5 years of ocean movement, updating every 8 hours implies 5 500 steps

Ocean simulation

- Simplified version of a part of Ocean application i.e. equation solver (called kernel)
- Illustrate the parallelization using:
 - shared memory
 - message passing
- Decomposition and assignment are the same for the two models

Kernel solver

- Grid: $(n+2) \times (n+2)$ - border rows and columns contain boundary values that do not change, the rest of $n \times n$ points are updated by the solver
- Computation for each point:
 - replace its value with a weighted average of itself and its four nearest-neighbor points
 - compute for each point the difference of updated value from its old value
 - if the average difference over all elements is smaller than a predefined value → the solution has converged → kernel stops, otherwise another sweep
- Update computation for a point sees the new values of the points above and to the left and the old values of the points below and to the right

```

1. int n;                                /*size of matrix: (n + 2-by-n + 2) elements*/
2. float **A, diff = 0;

3. main()
4. begin
5.     read(n);                         /*read input parameter: matrix size*/
6.     A ← malloc (a 2-d array of size n + 2 by n + 2 doubles);
7.     initialize(A);                  /*initialize the matrix A somehow*/
8.     Solve (A);                      /*call the routine to solve equation*/
9. end main

10. procedure Solve (A)                 /*solve the equation system*/
11.    float **A;
12.    begin
13.        int i, j, done = 0;
14.        float temp;
15.        while (!done) do             /*outermost loop over sweeps*/
16.            diff = 0;               /*initialize maximum difference to 0*/
17.            for i ← 1 to n do       /*sweep over nonborder points of grid*/
18.                for j ← 1 to n do
19.                    temp = A[i,j];   /*save old value of element*/
20.                    A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                                    A[i,j+1] + A[i+1,j]); /*compute average*/
22.                    diff += abs(A[i,j] - temp);
23.                end for
24.            end for
25.            if (diff/(n*n) < TOL) then done = 1;
26.        end while
27.    end procedure

```

Kernel solver – decomposition

- Straightforward: program structure-based approach
 - start from the individual loop or loop nests in the program
 - check if their iterations can be performed in parallel
 - check if enough concurrency is exposed
 - look for concurrency across loops

Kernel solver – decomposition

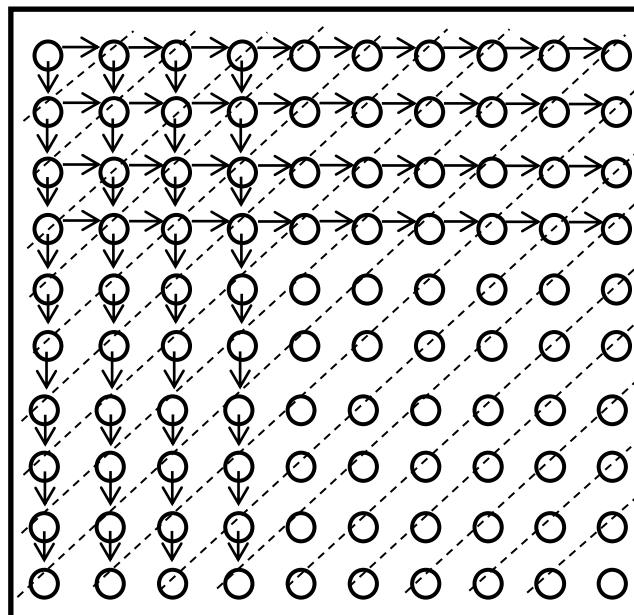
- Decompose work in tasks; each task updates a single grid point
- Ways to exploit exposed concurrency:
 - A. keep loop structure and insert point-to-point synchronization
 - B. change the loop structure
 - C. exploit knowledge of problem beyond the dependencies in the sequential program
 - D. ignore the dependencies among grid points within a sweep, global synchronization between sweeps

Kernel solver – decomposition

- A. Keep loop structure and insert point-to-point synchronization
 - by point-to-point synchronization it is ensured that the new value for a grid point has been produced in the current sweep before it is used by the points below or to its right
 - different loops and sweeps might be in progress at the same time on different elements
 - disadvantage: synchronization overhead

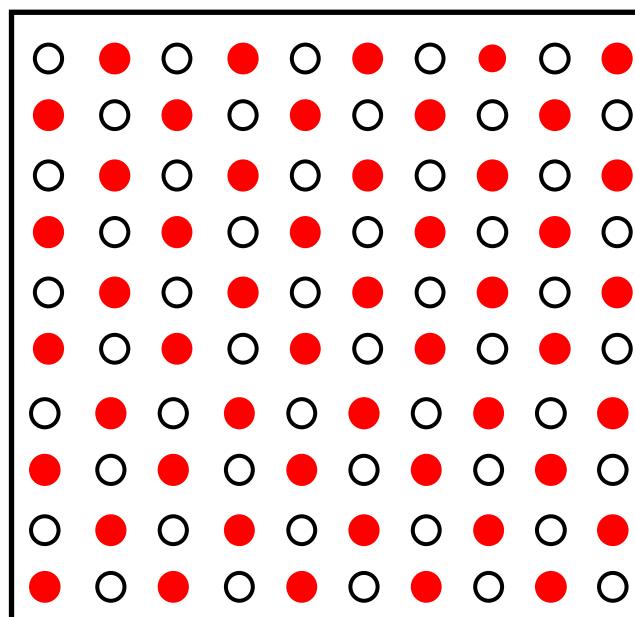
Kernel solver – decomposition

- B. Change the loop structure
 - check fundamental dependencies in the algorithm



Kernel solver – decomposition

- C. Exploit knowledge of problem beyond the dependencies in the sequential program



Kernel solver – decomposition

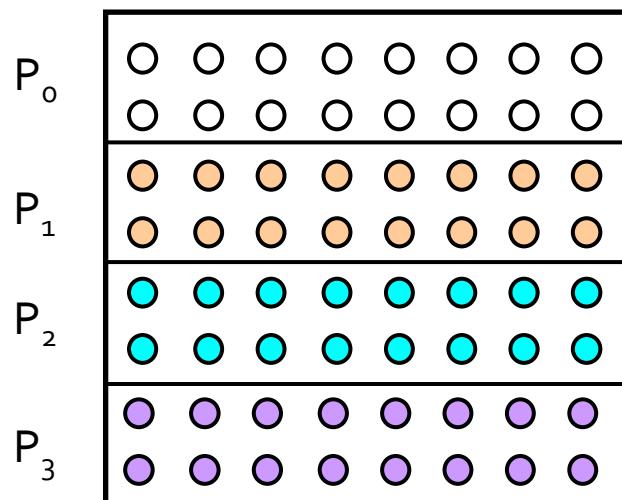
- D. Ignore the dependencies among grid points within a sweep, global synchronization between sweeps

```
15. while (!done) do                                /*a sequential loop*/
16.     diff = 0;
17.     for_all i ← 1 to n do                      /*a parallel loop nest*/
18.         for_all j ← 1 to n do
19.             temp = A[i,j];
20.             A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                         A[i,j+1] + A[i+1,j]);
22.             diff += abs(A[i,j] - temp);
23.         end for_all
24.     end for_all
25.     if (diff/(n*n) < TOL) then done = 1;
26. end while
```

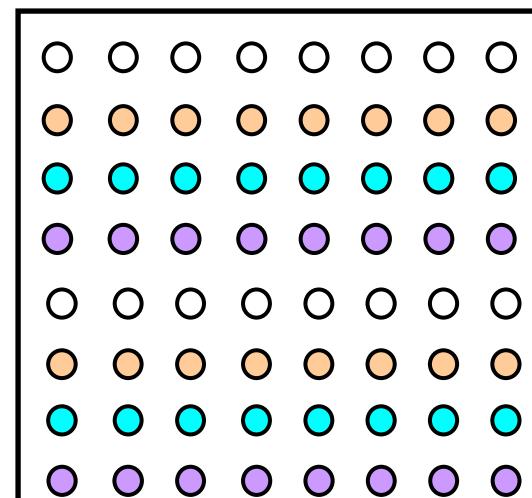
Kernel solver – assignment

- P processes (threads)
 - Each process takes N/P

Block



Interleaved



Kernel solver – orchestration (DP)

```
1. int n, nprocs;                                /* grid size (n+2-by-n+2) and number of processes*/
2. float **A, diff = 0;

3. main()
4. begin
5.   read(n); read(nprocs);;           /*read input grid size and number of processes*/
6.   A ← Q_MALLOC (a 2-d array of size n+2 by n+2 doubles);
7.   initialize(A);                  /* initialize the matrix A somehow */
8.   Solve (A);                     /* call the routine to solve equation*/
9. end main

10. procedure Solve(A)                         /* solve the equation system */
11.   float **A;
12.   begin
13.     int i, j, done = 0;
14.     float mydiff = 0, temp;
15.     DECOMP A[BLOCK,*];
16.     while (!done) do                      /*outermost loop over sweeps */
17.       mydiff = 0;                         /*initialize maximum difference to 0 */
18.       for_all i ← 1 to n do             /*sweep over non-border points of grid */
19.         for_all j ← 1 to n do
20.           temp = A[i,j];               /* save old value of element */
21.           A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
22.                               A[i,j+1] + A[i+1,j]);      /*compute average */
23.           mydiff += abs(A[i,j] - temp);
24.       end for_all
25.     end for_all
26.     REDUCE (mydiff, diff, ADD);
27.     if (diff/(n*n) < TOL) then done = 1;
28.   end while
29. end procedure
```

Kernel solver – orchestration (SM)

- matrix A is declared as a shared array
- processes refer to A by load and store primitives

Name	Syntax	Function
CREATE	CREATE(p, proc, args)	Create p processes that start executing at procedure proc with arguments.
G_MALLOC	G_MALLOC(size)	Allocate shared data of size bytes.
LOCK	LOCK(name)	Acquire mutually exclusive access.
UNLOCK	UNLOCK(name)	Release mutually exclusive access.
BARRIER	BARRIER(name, number)	Global synchronization among number processes: none gets past BARRIER until number have arrived.
WAIT_FOR_END	WAIT_FOR_END(number)	Wait for number processes to terminate.
wait for flag	while(!flag); or WAIT(flag)	Wait for flag to be set (spin or block); used for point-to-point event synchronization.
set flag	flag = 1; or SIGNAL(flag)	Set flag ; wakes up process that is spinning or blocked on flag , if any.

```

1. int n, nprocs;           /* matrix dimension and number of processors to be used */
2a. float **A, diff;       /* A is global (shared) array representing the grid */
   /* diff is global (shared) maximum difference in current sweep */
2b. LOCKDEC(diff_lock);   /* declaration of lock to enforce mutual exclusion */
2c. BARDEC (bar1);        /* barrier declaration for global synchronization between sweeps */

3. main()
4. begin
5.   read(n);   read(nprocs); /* read input matrix size and number of processes*/
6.   A ← G_MALLOC (a two-dimensional array of size n+2 by n+2 doubles);
7.   initialize(A);          /* initialize A in an unspecified way*/
8a. CREATE (nprocs-1, Solve, A);
8.   Solve(A);              /* main process becomes a worker too*/
8b. WAIT_FOR_END;          /* wait for all child processes created to terminate */
9. end main

10. procedure Solve(A)
11. float **A;             /* A is entire n+2-by-n+2 shared array, as in the sequential program */
12. begin
13.   int i,j, pid, done = 0;
14.   float temp, mydiff = 0;           /* private variables */
14a. int mymin ← 1 + (pid * n/nprocs); /* assume that n is exactly divisible by */
14b. int mymax ← mymin + n/nprocs - 1; /* nprocs for simplicity here */

15. while (!done) do           /* outer loop over all diagonal elements */
16.   mydiff = diff = 0;        /* set global diff to 0 (okay for all to do it) */
17.   for i ← mymin to mymax do /* for each of my rows */
18.     for j ← 1 to n do      /* for all elements in that row */
19.       temp = A[i,j];
20.       A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                           A[i,j+1] + A[i+1,j]);
22.       mydiff += abs(A[i,j] - temp);
23.     endfor
24.   endfor
25a. LOCK(diff_lock);        /* update global diff if necessary */
25b.   diff += mydiff;
25c. UNLOCK(diff_lock);
25d. BARRIER(bar1, nprocs);  /* ensure all have got here before checking if done*/

25e. if (diff/(n*n) < TOL) then done = 1; /* check convergence; all get same answer*/
25f. BARRIER(bar1, nprocs); /* see Exercise c */
26. endwhile
27. end procedure

```

Kernel solver – orchestration (MP)

- matrix A is represented by a collection of smaller data structures distributed among the processes

Name	Syntax	Function
CREATE	CREATE(procedure)	Create p process that starts at procedure .
SEND	SEND(src_addr, size, dest, tag)	Send size bytes starting at src_addr to the dest process, with tag identifier.
RECEIVE	RECEIVE(buffer_addr, size, src, tag)	Receive a message with the tag identifier from the src process, and put size bytes of it into buffer starting at buffer_addr .
SEND_PROBE	SEND_PROBE (tag, dest)	Check if message with identifier tag has been sent to process dest (only for asynchronous message passing).
RECV_PROBE	RECV_PROBE(tag, src)	Check if message with identifier tag has been received from process src (only for asynchronous message passing).
BARRIER	BARRIER(name, number)	Global synchronization among number processes: none gets past BARRIER until number have arrived.
WAIT_FOR_END	WAIT_FOR_END(number)	Wait for number processes to terminate.

```

1. int pid, n, nprocs;           /* process id, matrix dimension and number of processors to be used */
2. float **myA;
3. main()
4. begin
5.   read(n);  read(nprocs);          /* read input matrix size and number of processes*/
8a.   CREATE (nprocs-1 processes that start at procedure Solve);
8b.   Solve();                      /* main process becomes a worker too*/
8c.   WAIT_FOR_END;                /* wait for all child processes created to terminate */
9. end main

10. procedure Solve()
11. begin
13.   int i,j, pid, n' = n/nprocs, done = 0;
14.   float temp, tempdiff, mydiff = 0;          /* private variables */
6.   myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2);/* my assigned rows of A */
7.   initialize(myA);                         /* initialize my rows of A, in an unspecified way */

15. while (!done) do
16.   mydiff = 0;                            /* set local diff to 0 */
16a. if (pid != 0) then SEND(&myA[1,0],n*sizeof(float),pid-1,ROW);
16b. if (pid = nprocs-1) then SEND(&myA[n',0],n*sizeof(float),pid+1,ROW);
16c. if (pid != 0) then RECEIVE(&myA[0,0],n*sizeof(float),pid-1,ROW);
16d. if (pid != nprocs-1) then RECEIVE(&myA[n'+1,0],n*sizeof(float),pid+1,ROW);
      /* border rows of neighbors have now been copied into myA[0,*] and myA[n'+1,*] */
17.   for i ← 1 to n' do                  /* for each of my rows */
18.     for j ← 1 to n do                /* for all elements in that row */
19.       temp = myA[i,j];
20.       myA[i,j] ← 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
21.                           myA[i,j+1] + myA[i+1,j]);
22.       mydiff += abs(myA[i,j] - temp);
23.   endfor
24. endfor
      /* communicate local diff values and obtain determine if done; can be replaced by reduction and broadcast */
25a. if (pid != 0) then                 /* process 0 holds global total diff*/
25b.   SEND(mydiff,sizeof(float),0,DIFF);
25c.   RECEIVE(mydiff,sizeof(float),0,DONE);
25d. else
25e.   for i ← 1 to nprocs-1 do        /* for each of my rows */
25f.     RECEIVE(tempdiff,sizeof(float),*,DONE);
25g.     mydiff += tempdiff;          /* accumulate into total */
25h.   endfor
25i.   for i ← 1 to nprocs-1 do        /* for each of my rows */
25j.     SEND(done,sizeof(int),i,DONE);
25k.   endfor
25l. endif
26.   if (mydiff/(n*n) < TOL) then done = 1;
27. endwhile
28. end procedure

```

Improving the performance of parallel programs

Bibliography

- Culler et al – Chapter 3
- Grama et al – Chapter 3

Parallelization process

1. Decomposition
2. Assignment
3. Orchestration
4. Mapping

Performance of parallel programs

- Optimize the performance = refine choices for decomposition, assignment, orchestration and mapping
- Goals:
 - Balance workload
 - Reduce communication
 - Reduce overhead introduced to increase parallelism, manage assignment, reduce communication, etc.
- **Optimization goals are at odds with each other!**
- Techniques to achieve these goals

Load balancing

- Ensuring that every processor does the same amount of work
- Goals for processes:
 - Same amount of work
 - Execute at the same time
 - Minimize the waiting time spent for sync
 - Minimize the serialization of processes due to mutual exclusion or dependencies

Load balancing issues

- Identifying enough concurrency in decomposition, and overcoming Amdahl's Law.
- Deciding how to manage the concurrency (statically or dynamically).
- Determining the granularity at which to exploit the concurrency.
- Reducing serialization and synchronization cost.

How to identify concurrency?

Decomposition techniques

- Identify enough concurrency
- Parallelizing loops leads to data parallelism
(same calculation on elements of a large data structure)
- Function (task) parallelism – different calculations performed in parallel on different or on the same data
- Too much? -> restrict the available concurrency by determining the granularity of tasks

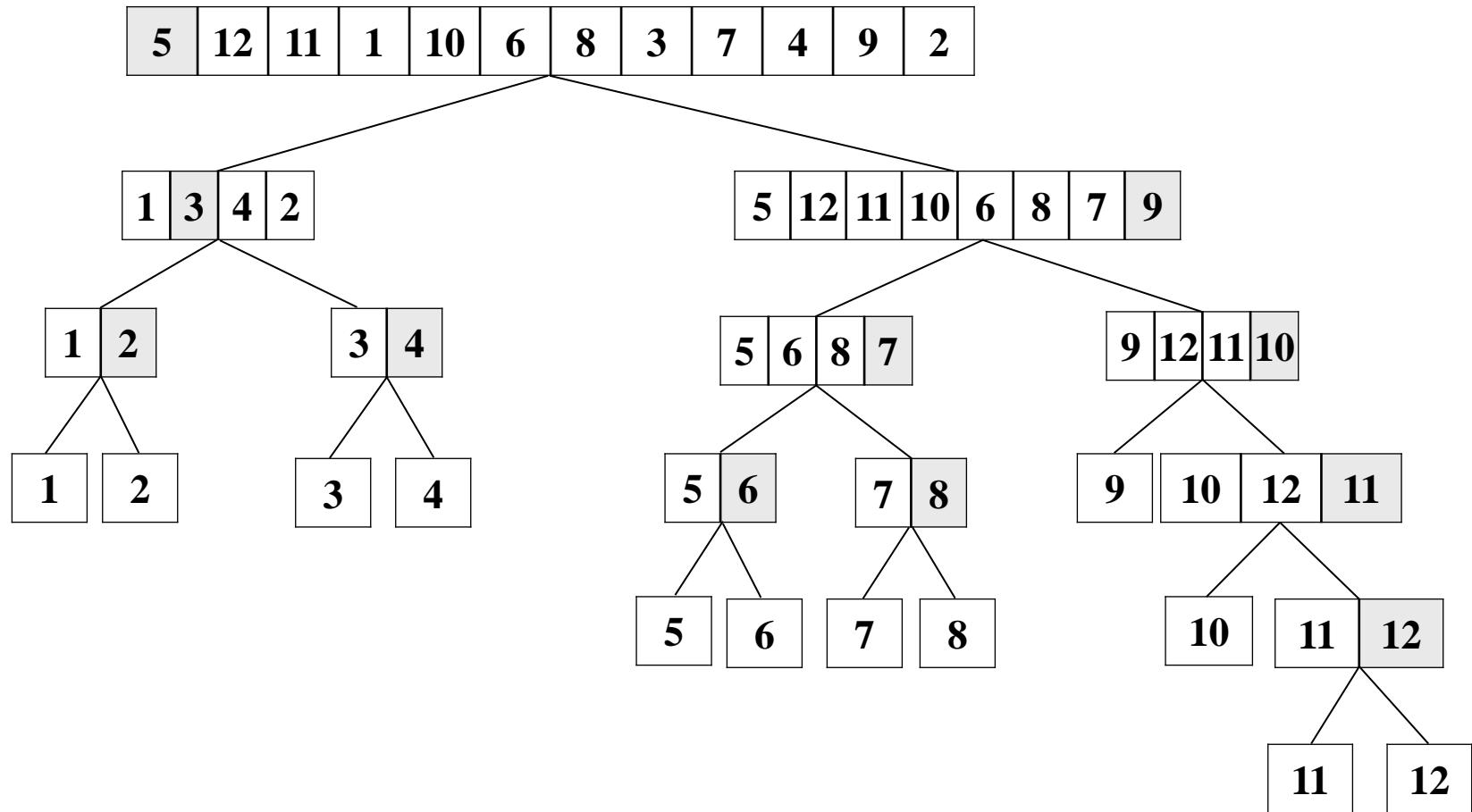
Decomposition techniques

- Allow to identify the concurrency in the problem
- Split the computation into a set of tasks
- Techniques
 - recursive decomposition
 - data decomposition
 - exploratory decomposition
 - speculative decomposition

Recursive decomposition

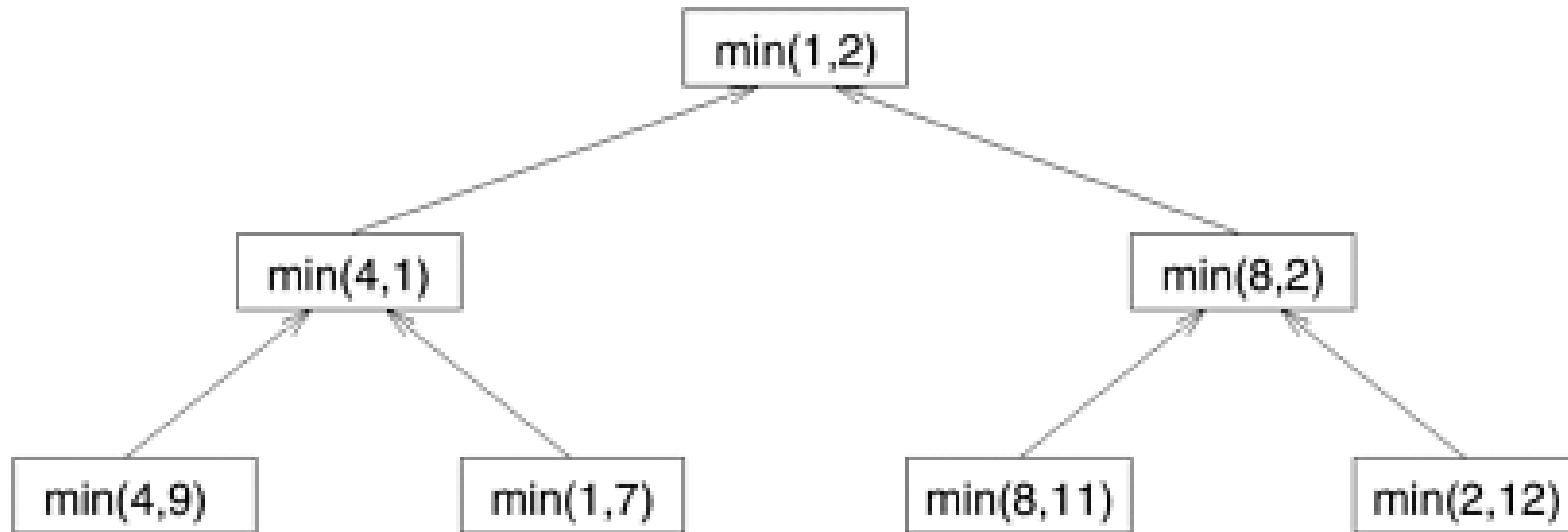
- Suitable for problems that can be solved using divide-and-conquer strategy
 - divide-and-conquer = problem divided into a set of independent subproblems, each subproblem is divided again ... at the end the partial results are combined
- Examples
 - quicksort – the sequential algorithm uses divide-and conquer
 - minimum – the sequential algorithm is modified to use divide-and-conquer

Quicksort



Minimum

- Restructure computation as a divide-and-conquer algorithm



Data decomposition

- Suitable for finding concurrency in algorithms that operate on large data structures
 - decomposition of computations is done into two steps
 - step 1: data is partitioned
 - step 2: data partitioning is used to induce a partitioning of the computations into tasks (operations performed by the tasks on the data partitions are similar or chosen from a small set of operations)
 - data partitioning can be done in various ways
 - partitioning output data
 - partitioning input data
 - partitioning both output and input data
 - partitioning intermediate data

Partitioning output data

- Partitioning of output data directly induces a decomposition of the problem into tasks
- Each task has to produce a part of the output
- Each partition of the output can be computed independently of others as a function of the input
- Example: matrix multiplication
 - partition output matrix C into four submatrices
 - each task computes one submatrix

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Task 1: $C_{1,1} = A_{1,1} B_{1,1} + A_{1,2} B_{2,1}$
Task 2: $C_{1,2} = A_{1,1} B_{1,2} + A_{1,2} B_{2,2}$
Task 3: $C_{2,1} = A_{2,1} B_{1,1} + A_{2,2} B_{2,1}$
Task 4: $C_{2,2} = A_{2,1} B_{1,2} + A_{2,2} B_{2,2}$

Partitioning output data

- a given data decomposition does not result into a unique task decomposition
- example – two task decomposition for the same data decomposition

Decomposition I	Decomposition II
Task 1: $C_{1,1} = A_{1,1} B_{1,1}$	Task 1: $C_{1,1} = A_{1,1} B_{1,1}$
Task 2: $C_{1,1} = C_{1,1} + A_{1,2} B_{2,1}$	Task 2: $C_{1,1} = C_{1,1} + A_{1,2} B_{2,1}$
Task 3: $C_{1,2} = A_{1,1} B_{1,2}$	Task 3: $C_{1,2} = A_{1,2} B_{2,2}$
Task 4: $C_{1,2} = C_{1,2} + A_{1,2} B_{2,2}$	Task 4: $C_{1,2} = C_{1,2} + A_{1,1} B_{1,2}$
Task 5: $C_{2,1} = A_{2,1} B_{1,1}$	Task 5: $C_{2,1} = A_{2,2} B_{2,1}$
Task 6: $C_{2,1} = C_{2,1} + A_{2,2} B_{2,1}$	Task 6: $C_{2,1} = C_{2,1} + A_{2,1} B_{1,1}$
Task 7: $C_{2,2} = A_{2,1} B_{1,2}$	Task 7: $C_{2,2} = A_{2,1} B_{1,2}$
Task 8: $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$	Task 8: $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$

Partitioning input data

- Partitioning of input data directly induces a decomposition of the problem into tasks
 - a task is created for each partition of input data
 - combine the results
- Example: computing the sum of N elements using p processors
 - partition the input into p subsets of (nearly) equal size
 - each task computes the sum for one subset
 - p partial sums are added to produce the final result

Partitioning intermediate data

- Suitable for algorithms organized in multiple steps (the output of one step is the input for next step)
- Sometimes leads to higher concurrency than partitioning input or output data
- Sometimes modifications of serial algorithms are needed as intermediate data are not explicitly generated in the serial algorithm

Partitioning intermediate data

Stage I

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,1} & D_{1,2,2} \\ D_{2,1,1} & D_{2,1,2} \\ D_{2,2,1} & D_{2,2,2} \end{pmatrix}$$

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,1} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,1} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Stage II

A decomposition induced by a partitioning of D

Task 01: $D_{1,1,1} = A_{1,1}B_{1,1}$

Task 02: $D_{2,1,1} = A_{1,2}B_{2,1}$

Task 03: $D_{1,1,2} = A_{1,1}B_{1,2}$

Task 04: $D_{2,1,2} = A_{1,2}B_{2,2}$

Task 05: $D_{1,2,1} = A_{2,1}B_{1,1}$

Task 06: $D_{2,2,1} = A_{2,2}B_{2,1}$

Task 07: $D_{1,2,2} = A_{2,1}B_{1,2}$

Task 08: $D_{2,2,2} = A_{2,2}B_{2,2}$

Task 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$

Task 10: $C_{1,2} = D_{1,1,2} + D_{2,1,2}$

Task 11: $C_{2,1} = D_{1,2,1} + D_{2,2,1}$

Task 12: $C_{2,2} = D_{1,2,2} + D_{2,2,2}$

Exploratory decomposition

- suitable for problems which search a space for the solutions
 - partition the search space into smaller parts
 - concurrently search each part
 - example: 15-puzzle problem

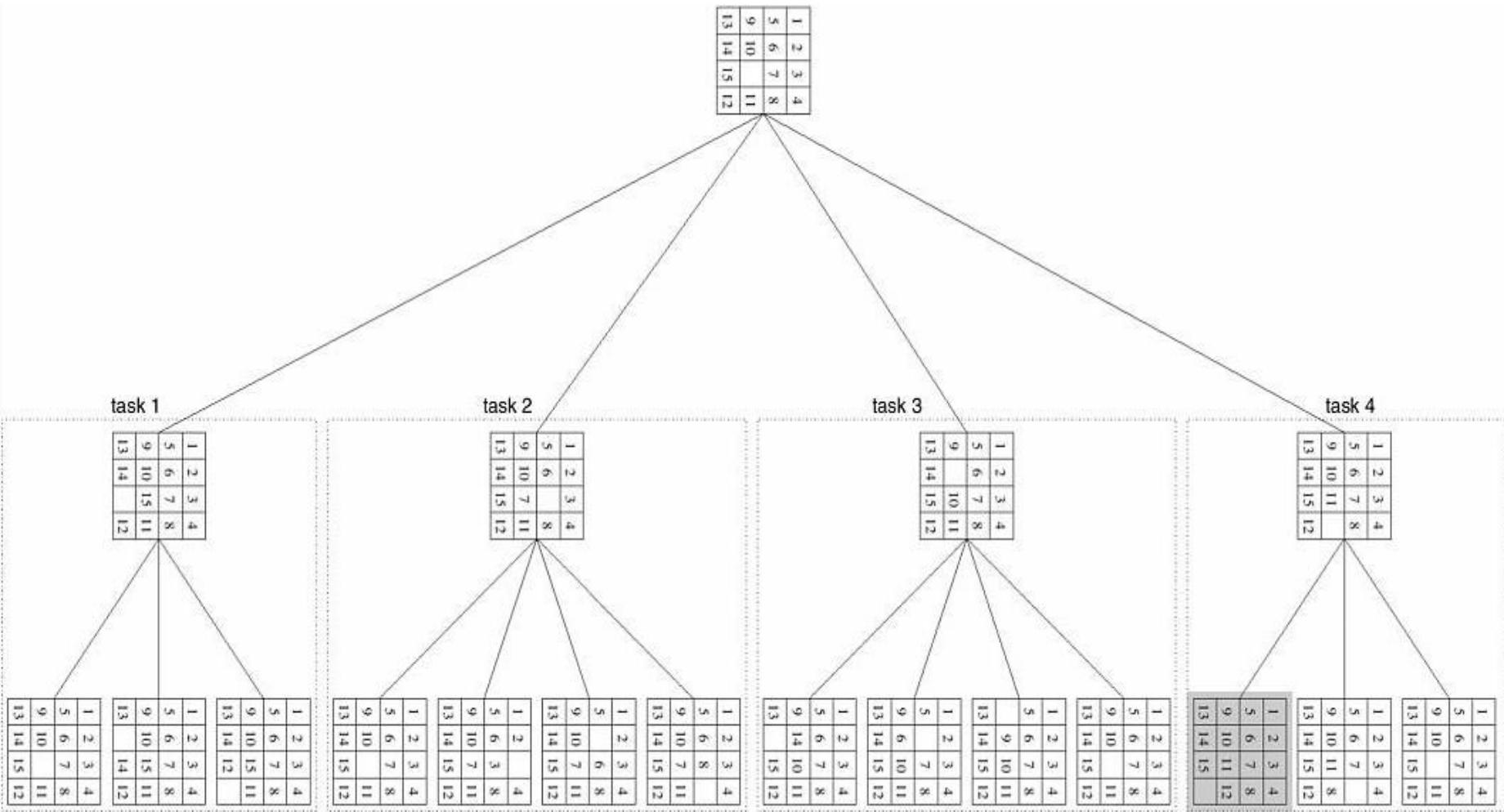
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	12

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	12

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	12

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

15 puzzle



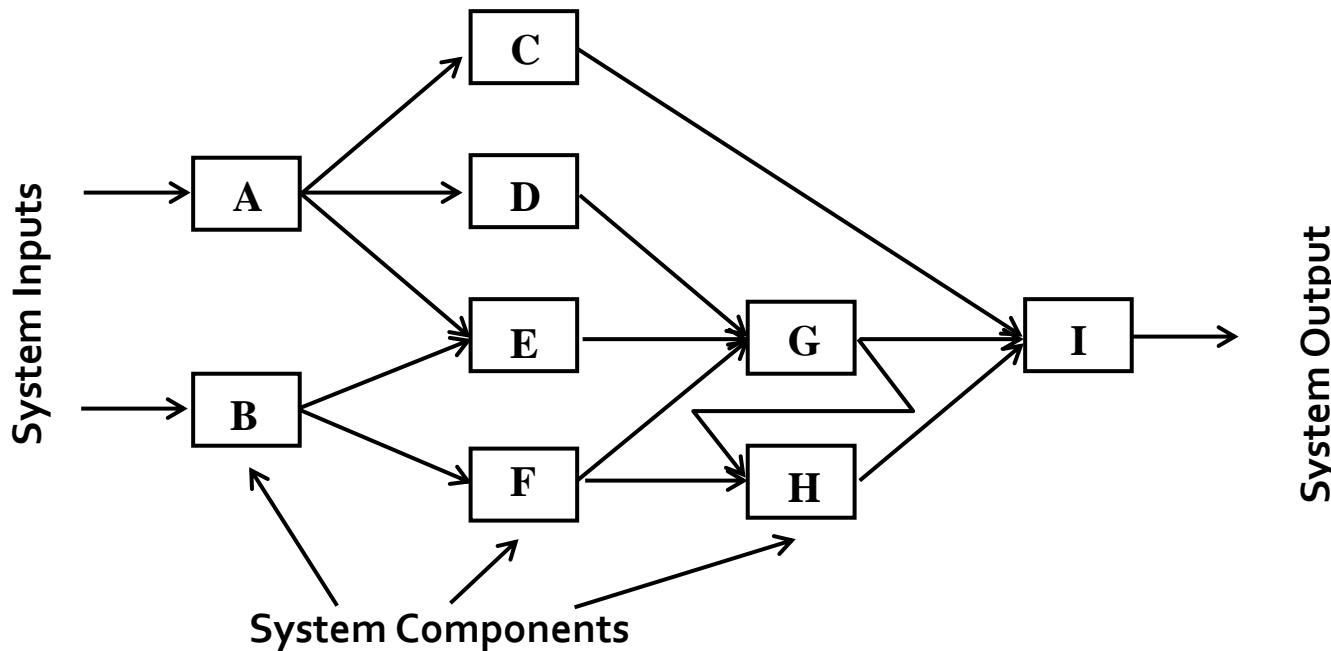
Exploratory decomposition

- Solution for 15-puzzle is based on tree-search
- Solving the problem in parallel
 - few levels of configurations are generated sequentially
 - each node is assigned to a task to expand it further → concurrent tasks
 - the task which finds the solution notifies the other tasks
- Is exploratory decomposition identical to data decomposition (search space = data)?
 - data decomposition: tasks induced by data decomposition are fully performed and each task perform useful computation towards the solution of the problem
 - exploratory decomposition: unfinished tasks can be terminated when the overall solution is found

Speculative decomposition

- Suitable for the situation in which a program may take one of many branches (B_1, \dots, B_n) depending on the output of another computation (C) that precedes it
 - C, B_1, \dots, B_n go in parallel
 - similar with evaluating branches of a switch statement in parallel before the input for switch is available
 - computations of some branches are discarded
 - minimize of wasted computation – consider the most likely / promising branch(es)

Simulator – event based

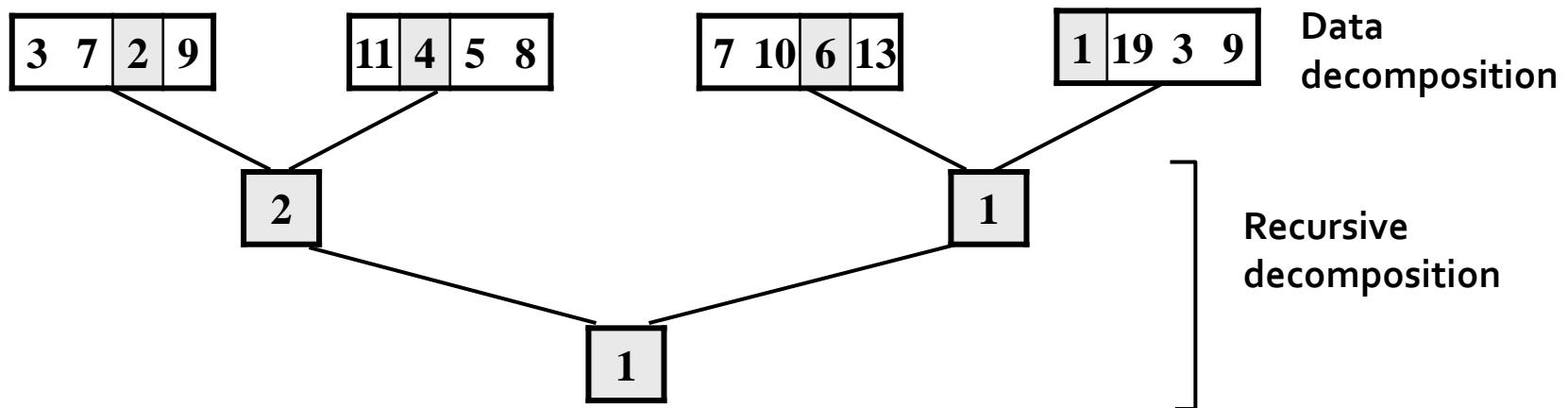


Speculative decomposition

- Is speculative decomposition identical to exploratory decomposition?
 - exploratory decomposition performs smaller, equal or greater amount of work compared to the serial algorithm (depending on the location of solution in the search space)
 - speculative decomposition performs equal or greater amount the work compared to the serial algorithm

Hybrid decomposition

- combine decomposition techniques
- computation is structured in multiple stages, apply different types of decomposition for different stage



Task characteristics

- Task generation
 - static: data partitioning, recursive decomposition (ex: minimum), exploratory decomp.
 - dynamic: recursive decomposition (ex: quicksort), exploratory decomp.
- Task size/granularity (time required to complete the task)
 - uniform
 - non-uniform
- Knowledge of task size
 - known
 - not known
- Size of data
 - input data
 - output data
 - amount of computation

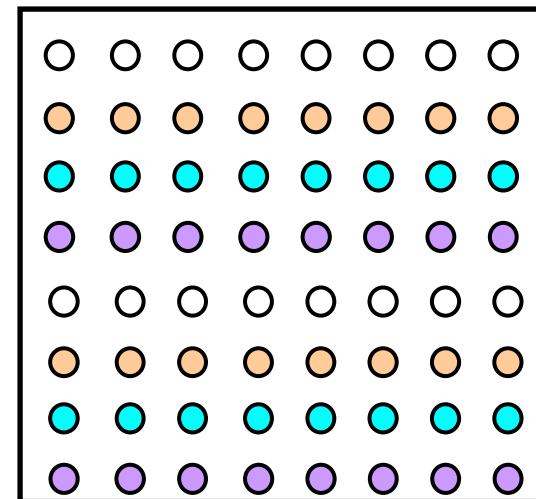
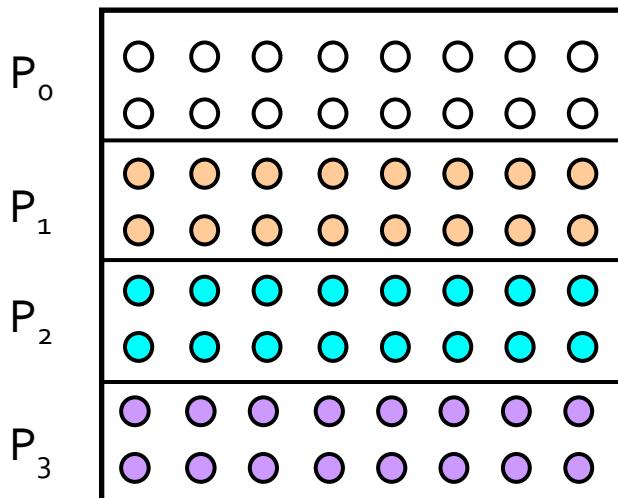
Inter-task interactions characteristics

- static vs dynamic
 - static: interactions are predetermined
 - dynamic: interactions are not known prior to the execution
- regular vs irregular
 - regular: structure can be exploited for efficient implementation
 - irregular: structure is unknown
- read-only vs read-write
- one-way vs two-way
 - one-way: one task initiates the communication and completes it without interrupting the other task
 - two-way: data needed by a task is explicitly supplied by another task

How to manage concurrency?

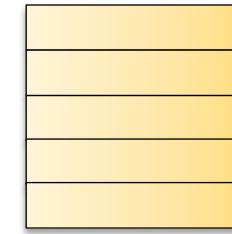
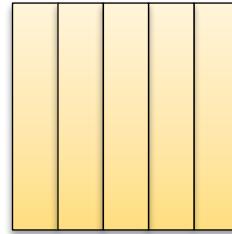
Static assignment

- Assignment of work to processes is pre-determined
- Static task generation => static assignment
- Solver example:

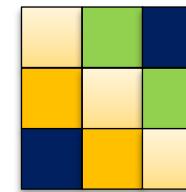
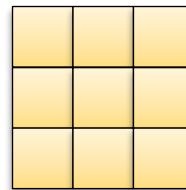


Block distributions

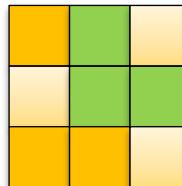
- 1D-block



- 2D-block



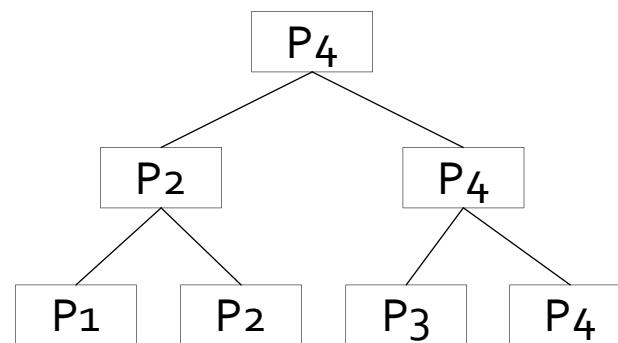
- Block-cyclic



- Randomized block

Task partitioning

- Task dependency graph
- Optimal assignment – NP-complete problem
- Find approximate solution



Static assignment

- Advantages: simple, zero runtime overhead
- Requirements:
 - The cost and amount of work is predictable
 - Processes have the same (known) execution time
 - Processes have different (known) execution time
 - When statistics about execution time are known
 - No interference from other applications

Semi-static assignment

- Cost of work is predictable for near-term future
- Application periodically profiles itself and re-computes assignment
- Good for environments that evolve slowly

Dynamic assignment

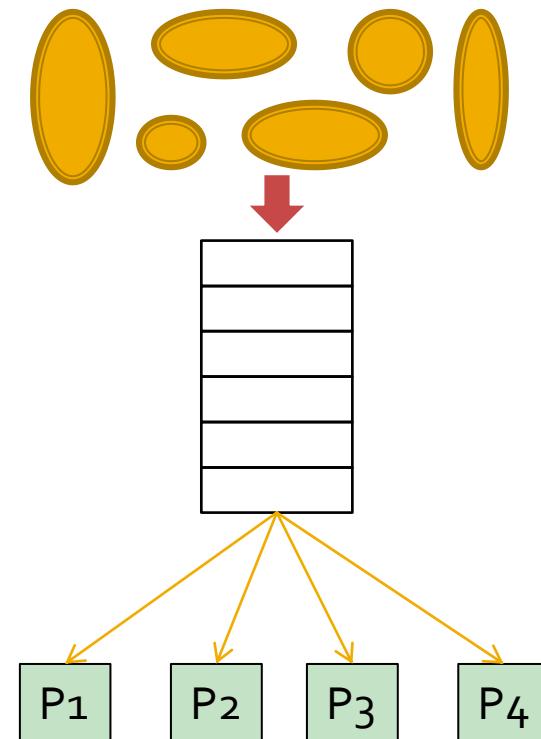
- Work distribution or the system environment is too unpredictable to periodically re-compute the assignment
- Program determines the assignment dynamically at runtime to ensure a well distributed load

Dynamic assignment

- Self-scheduling (parallel loop):
 - Shared loop counter
 - Processes increment the counter (atomically), execute the iteration, and repeat...
- Collection of work queues
 - Tasks are inserted into queues
 - Processes remove tasks and execute them

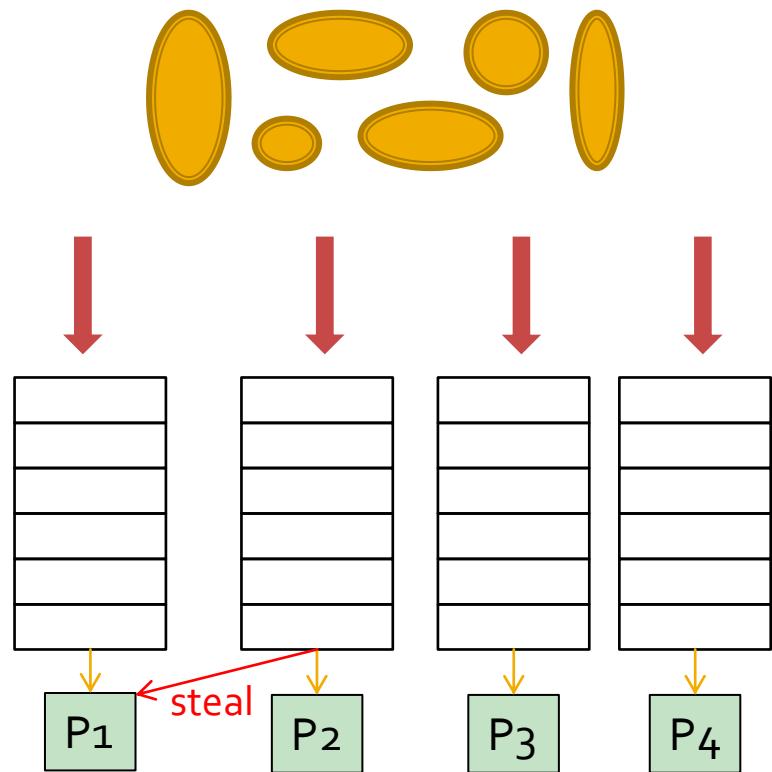
Dynamic assignment – shared queue

- Single centralized queue
- Simple
- All processes access the same queue:
 - Contention
 - Mutual exclusion
- Performance bottleneck:
 - Many small tasks
 - Large numbers of processes
- Load imbalance: large tasks



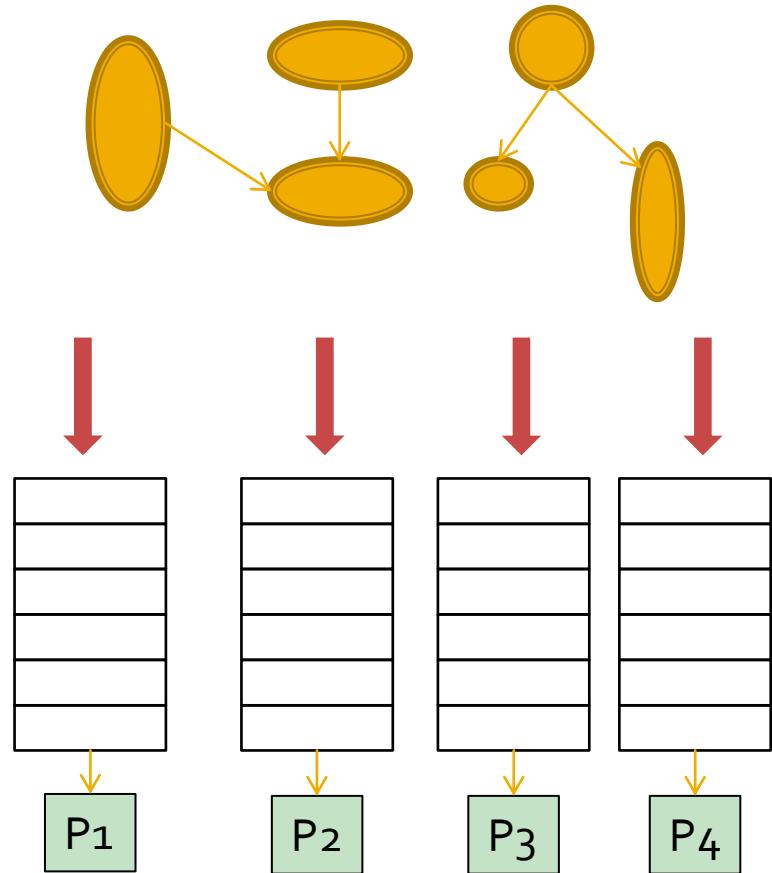
Dynamic assignment – distributed queue

- Avoids synchronization
- Expensive management
- Intelligent assignment of tasks to queues: can minimize communication
- Task stealing: generates contention
 - Minimize stealing
 - How much to steal
 - Where to steal from
 - Termination detection



How about task dependencies?

- Task is removed from queue and assigned to process after all dependencies are satisfied



Task granularity

Task granularity

- Load imbalance caused by
 - Task dependences
 - Granularity of the largest task
- Granularity = amount of work associated with a task, measured by the number of instructions (execution time)
- Small tasks:
 - Better load balancing
 - Larger management overhead
 - More contention
 - More communication

Task granularity

- Dynamic task queuing
 - Large management overhead with small tasks
 - Contention and communication increase with smaller tasks
- Static assignment
 - Smaller effect on task management overhead
 - Larger tasks can cause load imbalance

Questions

- Define the term “Partitioning” in the context of parallel programs design.
- Which are the issues of task stealing (distributed task queues)?

Improving the performance of parallel programs (part 2)

Bibliography

- Culler et al – Chapter 3
- Grama et al – Chapter 3

Optimization goals

- Balance workload
- Reduce communication
- Reduce overhead introduced to increase parallelism, manage assignment, etc.

Load balancing issues

- Identifying enough concurrency in decomposition, and overcoming Amdahl's Law.
- Deciding how to manage the concurrency (statically or dynamically).
- Determining the granularity at which to exploit the concurrency.
- **Reducing serialization and synchronization cost.**

Decomposition techniques

- Recursive
- Data
- Exploratory
- Speculative
- Hybrid

- Identify enough concurrency
- If too much, restrict by determining granularity of tasks

Management of concurrency

- Static assignment
 - Simple, predetermined assignment
 - Not much management overhead
 - Cost and amount of work should be predictable
- Semi-static assignment
 - Cost of work is predictable for near-term future
 - Periodically re-compute assignment
- Dynamic assignment
 - Collection of work queues
 - Contention
 - Management overhead
- **Task granularity influences contention and communication overhead**

Reduce serialization

- Serialization = wait time due to synchronization or mutual exclusion
- Synchronization
 - High wait time -> barriers
 - Lower wait time -> point-to-point or group
- Mutual exclusion
 - Separate locks for separate data items
 - Make critical sections smaller and less frequent
- Fine-grained synch and locks
 - Less wait time
 - Higher overhead (programming, execution time, not enough reuse...)

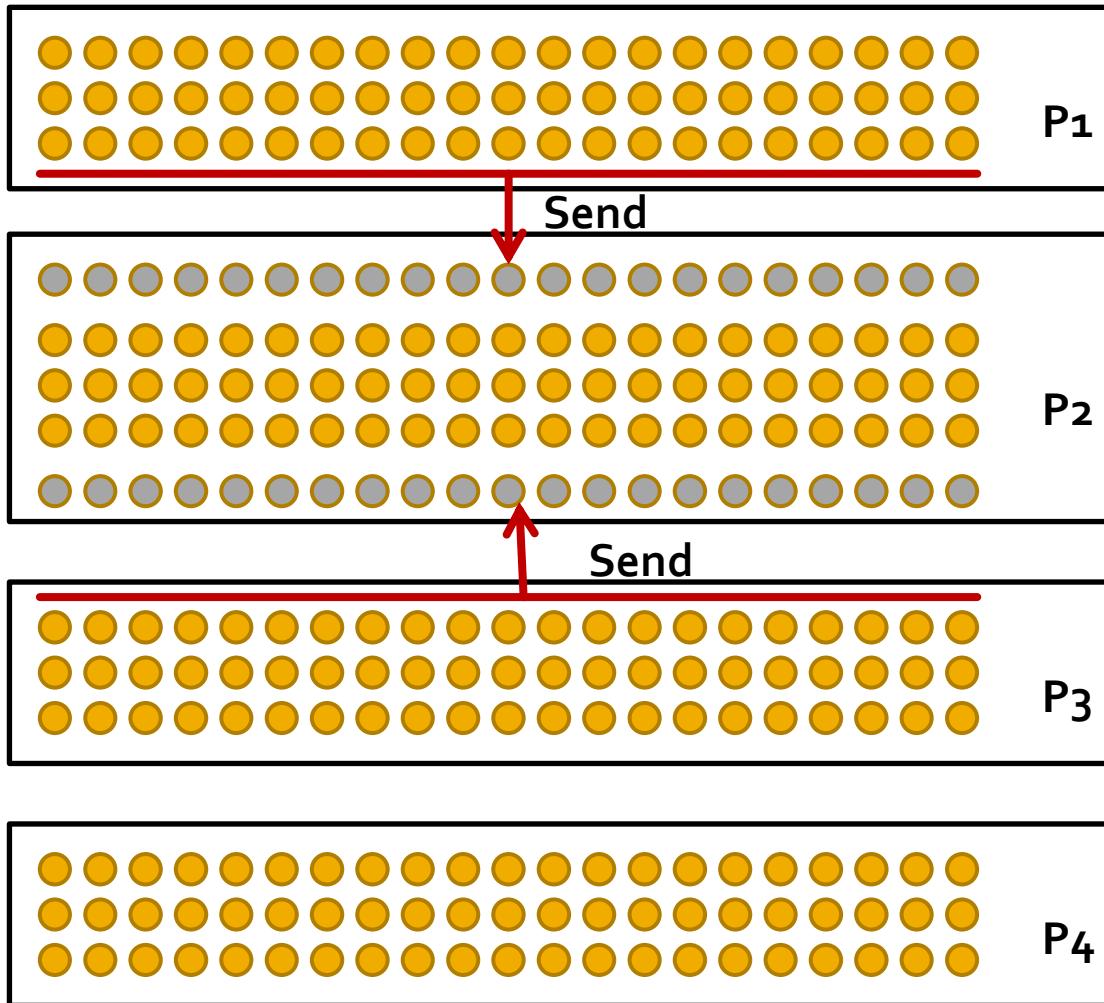
Second optimization goal

Reduce communication

Inherent communication

- Decompose a problem in tasks => communication among tasks
- Inherent communication = communication that must occur in the parallel algorithm
- Challenge: reduce inherent communication while still preserving load balancing

Solver – message passing implementation



Communication to computation ratio

Amount of communication (e.g. bytes)

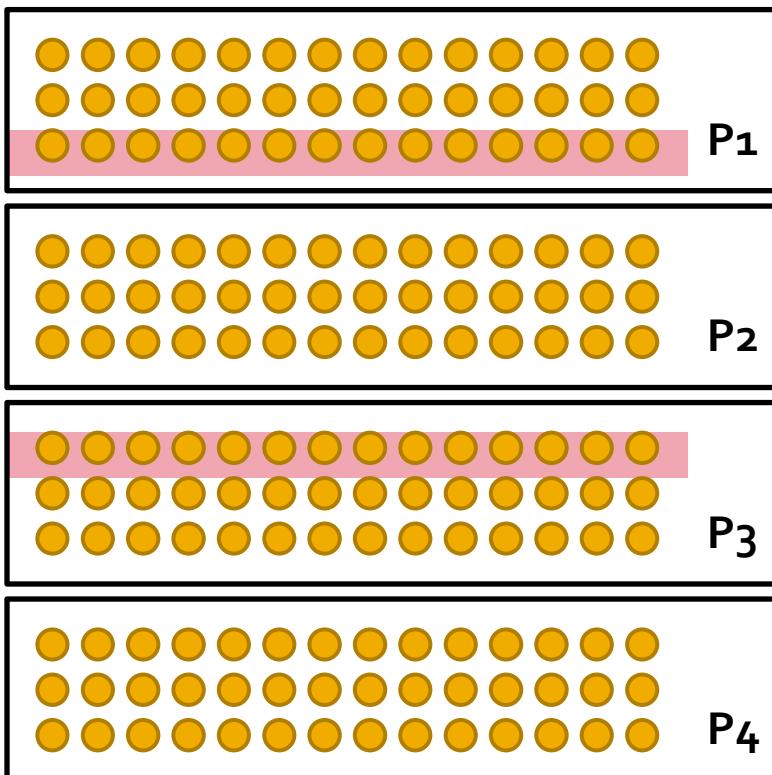
Amount of computation (e.g. instructions)

- Low communication to computation ratio is required to efficiently utilize modern parallel processors

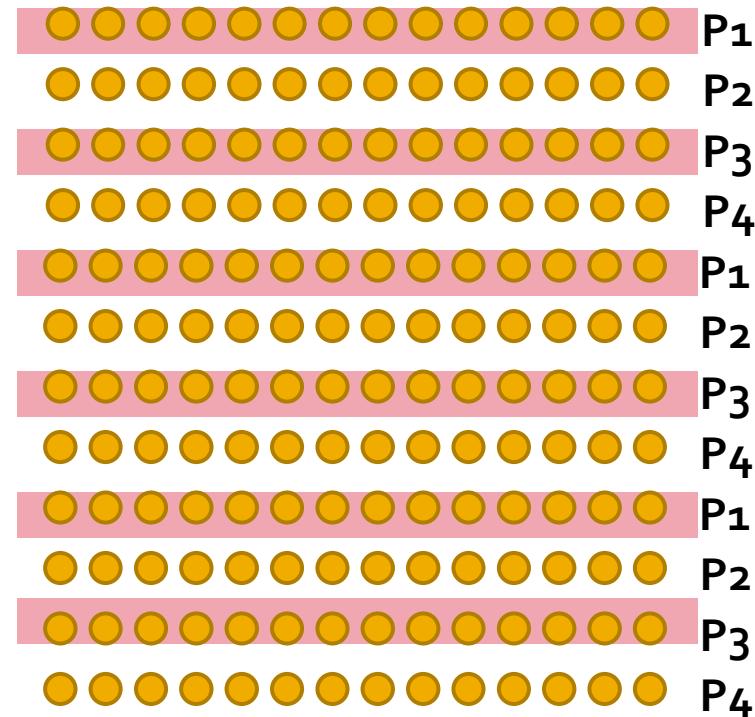
Reducing inherent communication

- Good assignment can reduce inherent communication

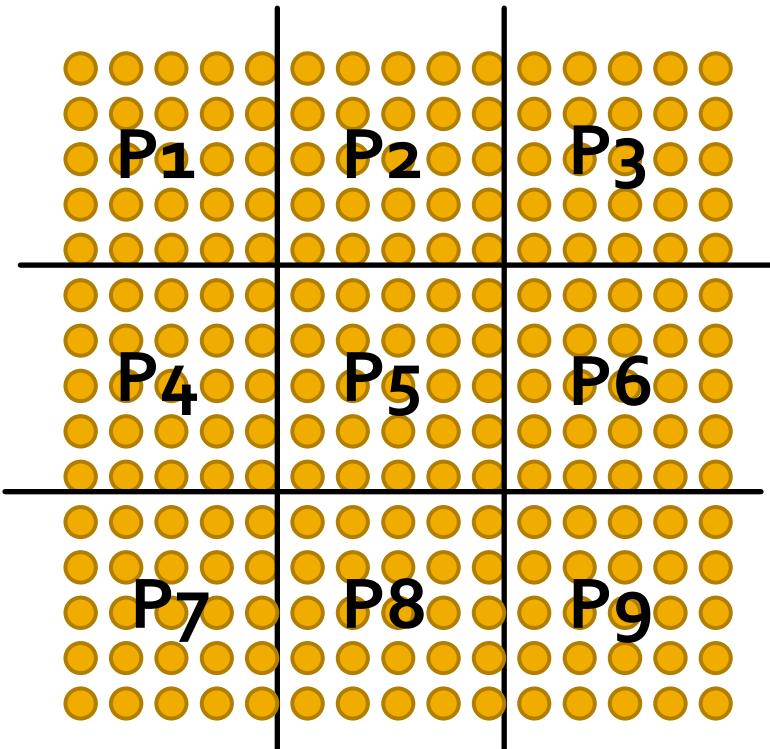
1D block assignment $\approx P/N$



1D interleaved assignment = 2



Reducing inherent communication



- Domain decomposition
- 2D block assignment
- Communication to computation ratio: $\frac{\sqrt{P}}{N}$

Extra work required for parallel execution

- Compute assignment
- Compute data values redundantly when communication cost is high
- Orchestration:
 - Process creation
 - Dynamic tasking
 - Synchronization operations
 - Structuring communication
 - Etc...

Speedup limit

$$Speedup(P) \leq \frac{SequentialWork}{\max(Work + SyncWait + Comm + Extra)}$$

Parallel Random Access Memory (PRAM)

- This model assumes that data access is free
- Used for the analysis of parallel algorithms
- Can we oversee communication cost in real systems?

Artifactual communication

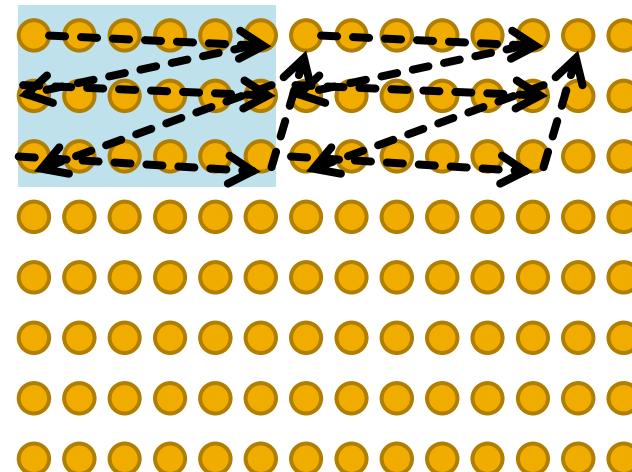
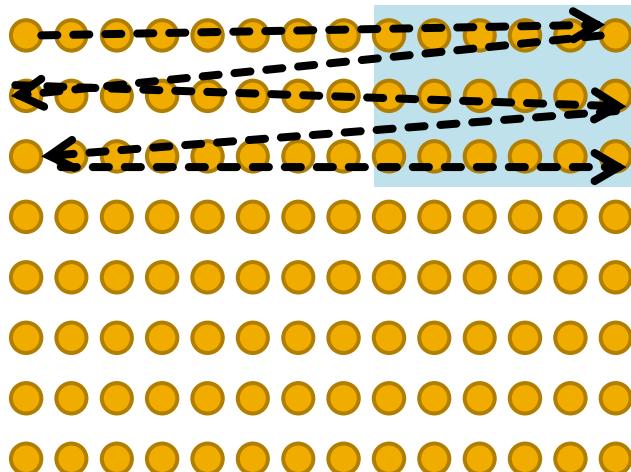
- Real parallel hardware - memory hierarchy, interconnection networks
- Artifactual communication – results from practical details of system implementation
- Sources:
 - Poor allocation of data: remote data access
 - Unnecessary data in a transfer: receiver does not use all transferred data
 - Unnecessary transfers due to system granularities: extra communication for cache coherence
 - Redundant communication of data
 - Finite replication capacity: cache is too small

Multiprocessor memory model

- Memory hierarchy of multiprocessors:
 - Cache
 - Local memory
 - Remote memory
- Memory wait time:
 - Cold start: first time data is accessed
 - Capacity miss: due to small cache size
 - Conflict miss: due to cache management policy
 - Communication miss: due to inherent or artifactual communication in parallel systems

Exploit temporal locality

- Access same memory location repeatedly in a short time-frame
- Use blocking to exploit temporal locality = accesses a subset of data that fits in a level of the hierarchy, use that data as much as possible

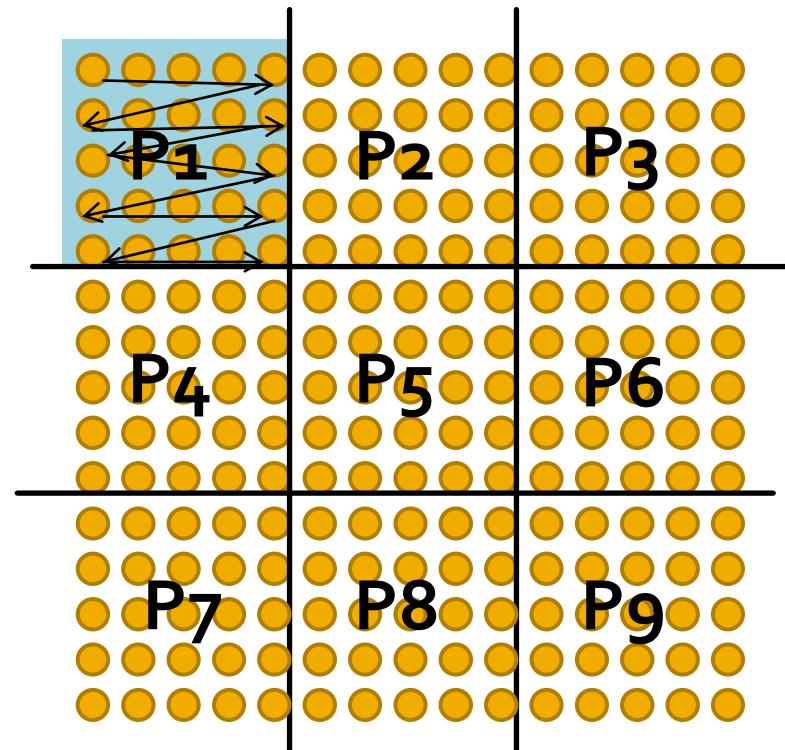
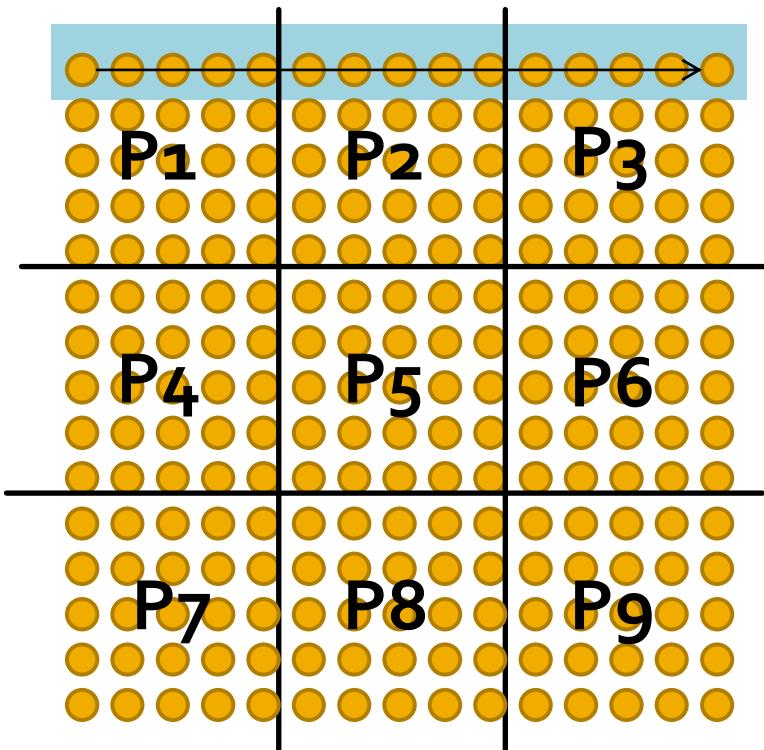


Exploit spatial locality

- Granularity of allocation: cache line size, page size
- Keep close together: the data accessed by a given processor (contiguous in the address space)
- Keep apart: unrelated data accessed by different processors
- Restructure data to interact better with the granularity of allocation

Exploit spatial locality

- Make sure: page and cache line within partition
- Change data structure from 2D to 4D to be able to fit data in the same cache line or page



Structuring communication to reduce cost

- How communication is organized and structured into messages influences:
 - Inherent communication
 - Artifactual communication
 - Communication to computation ratio
- Communication cost
 - +Overhead
 - +Delay
 - +Contention
 - -Amount of the communication cost that can be overlapped with computation

Reduce overhead

- Make messages fewer in number and hence larger
- Solutions:
 - Message passing: by programmer
 - Shared memory: by the system
- Solver example: send entire row in a single message

Reduce delay

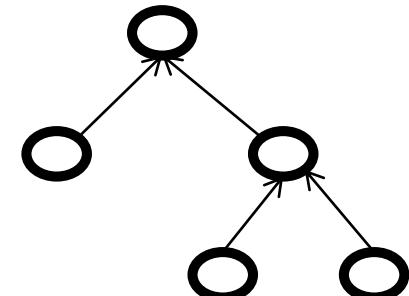
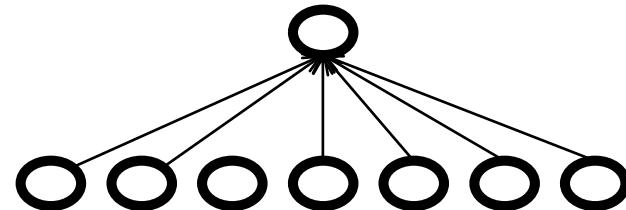
- Delay = nrHops * bitDelay(hop)
- Solution: mapping the processes to processors so that locality in network topology is exploited
- Network topology not considered as important on modern machines because of the characteristics of the machines :
 - overhead dominates hop latency
 - number of nodes usually not very large
 - machines used as general-purpose, multiprogrammed servers->mapping solved by the OS transparently

Reduce contention

- Multiprocessors communication systems have many resources:
 - Network links
 - Controllers
 - Memory systems
 - Network interfaces
- Resource can do operations at a given throughput
- Contention occurs when many requests to a resource are made within a small window of time (the resource is a “hot spot”)

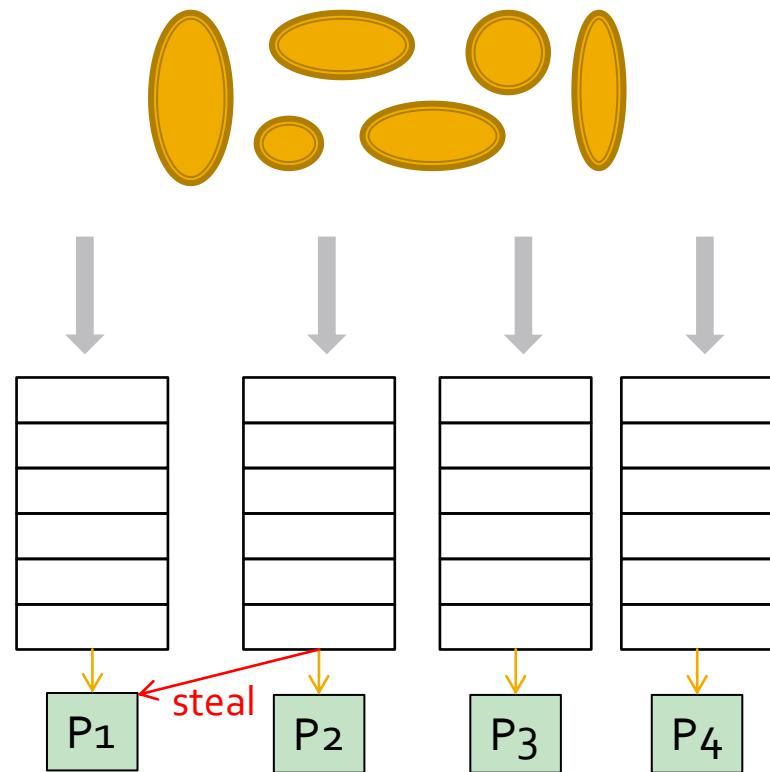
Reduce contention

- Recall: global sum example
- All processes trying to update a shared variable
- Flat communication:
 - Potential high contention
 - Low latency if no contention
- Tree structured communication:
 - Reduces contention
 - Higher latency under no contention



Example: distributed work queues

- Serve to reduce contention
- Take data from OWN work queue
- Put new work to OWN work queue



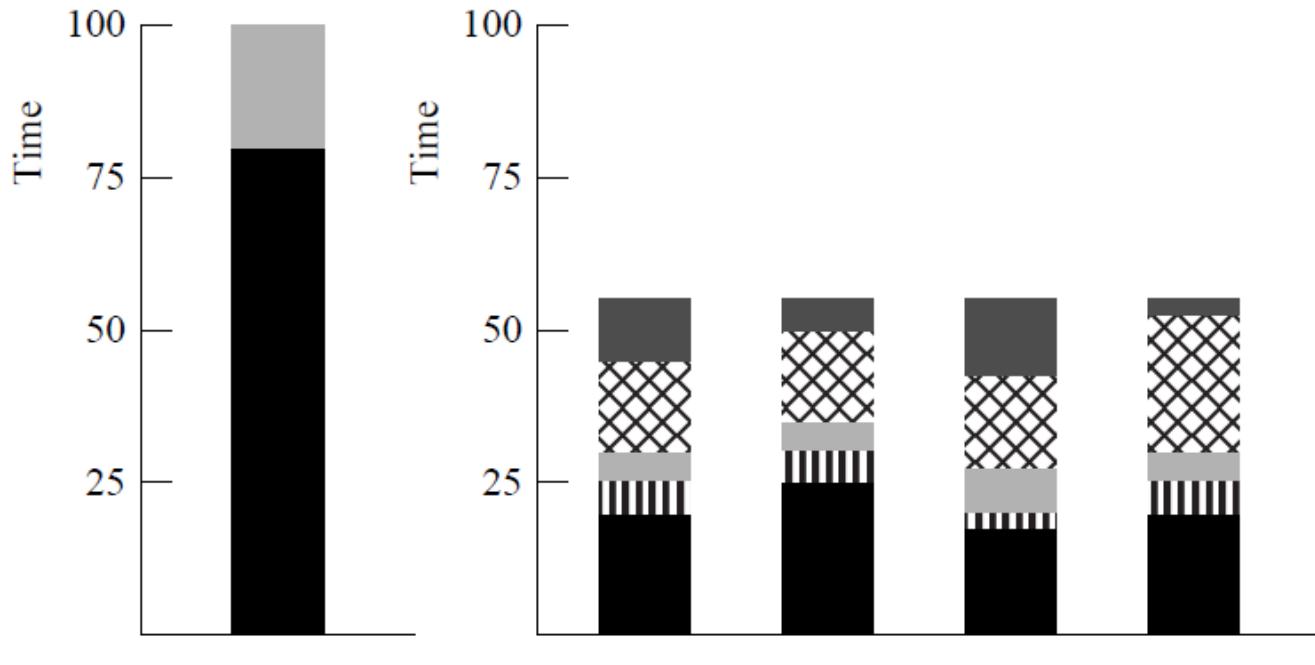
Overlapping communication with computation

- If processor would stall while communication occurs -> no parallel performance
- Hide the cost of communication
 - Overlap communication with computation
 - Overlap communication with communication
- Techniques:
 - HW: pipelining, multi-threading, pre-fetching, out-of-order, etc
 - Use asynchronous communication
 - Multithreading: requires additional concurrency in application
(more concurrency than number of execution units)

Components of parallel execution time

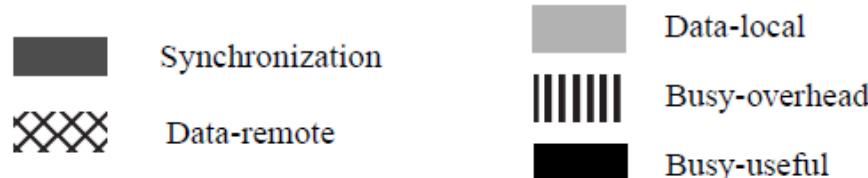
- Busy-useful: execute instructions
- Busy-overhead: execute instructions needed only for the parallel algorithm
- Data-local: wait for local data
- Data-remote : wait for remote data
- Synchronization: wait for another process (includes load imbalance and serialization)

Components of parallel execution time



(a) Sequential

(b) Parallel with four processors



Parallel algorithm models

- Data-parallel
 - each task performs similar operations on different data
 - data parallelism
 - work may be done in phases (separated by interactions)
- Task graph
 - based on the task-dependency graph to reduce communication cost
 - divide-and-conquer decomposition
- Work pool
 - dynamic assignment
 - any task can be performed by any process
 - centralized/distributed queues

Parallel algorithm models

- Master-worker
 - master process generates work and assigns it to worker processes
- Producer-consumer (pipeline)
 - a stream of data is passed through a succession of processes
 - chains of processes (no cycles)
- Hybrid

Workload-driven evaluation

Bibliography

- Culler et. al. – Chapter 4

Until now...

- Parallel execution in modern microprocessors
- Parallel programming models
- Parallel algorithms design
- Improve the performance of parallel programs

What's next?

- Measure the performance of parallel programs on a real parallel machine

Single processor case

- Chose relevant workloads -> benchmark suites (standard)
- Measure
 - Absolute performance (execution time, etc.)
 - Efficiency
- Evaluate new architectural feature through simulation -> expensive

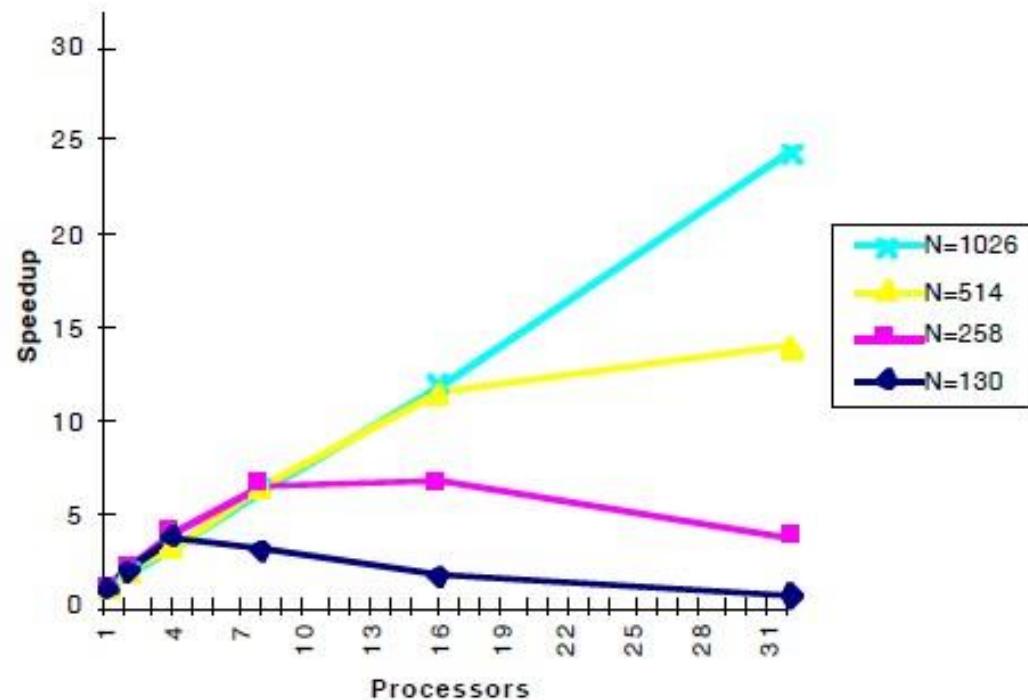
What to measure?

- Absolute performance:
 - execution time
 - operations per second
- Overhead:
 - Total parallel overhead: $T_o = PT_p - T_s$
- Speedup
 - Exec time (1) / Exec time (p)
 - Ops per sec (p) /Ops per sec (1)
- Efficiency
 - performance/unit of resource
 - $E = \text{Speedup} / P$
- Cost: PT_p

Speedup

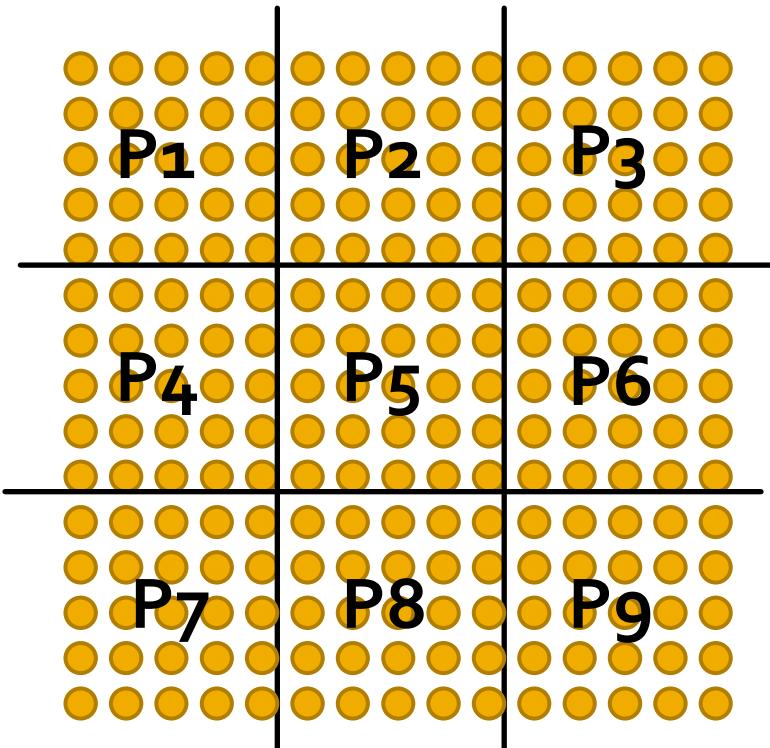
- Compare execution on p processors against:
 - Parallel version of a program running on one processor (makes yourself look good)
 - Best sequential program

Parallel machine case



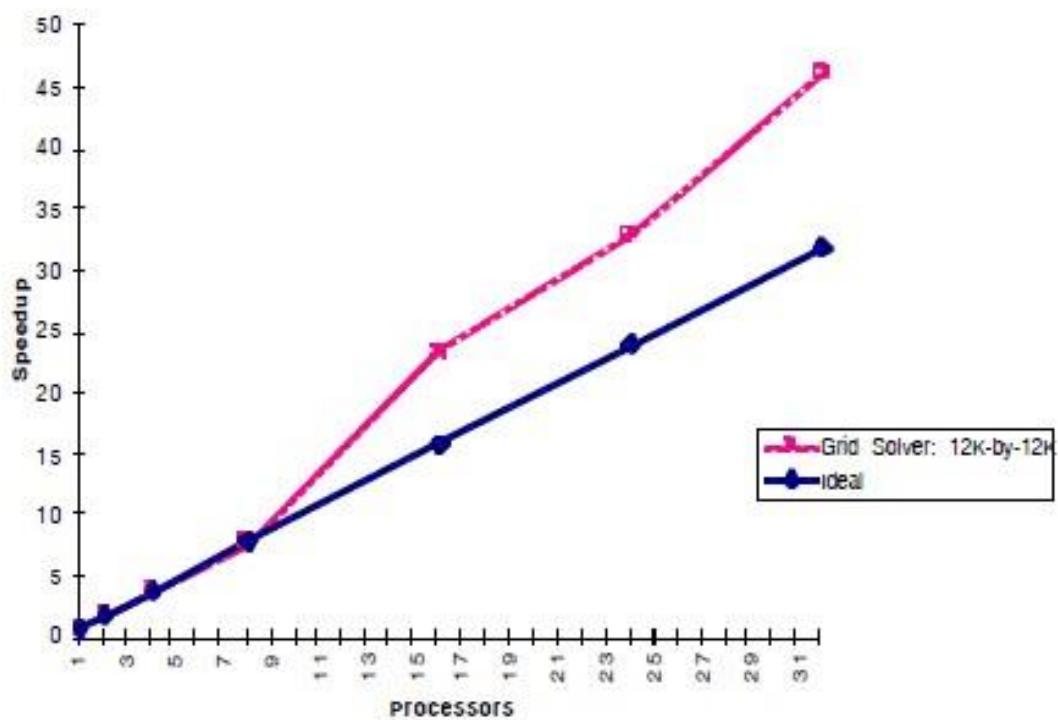
- Ocean execution on 32 processor SGI Origin 2000
- No benefit for small size problem

Solver example



- Communication to computation ratio = $\frac{\sqrt{P}}{N}$
- If N is small -> $\frac{\sqrt{P}}{N}$ is high

Superlinear speedup



- Problem is too large to fit on a smaller machine (e.g. data does not fit in cache)
- The amount of work done by the sequential algorithm is larger (e.g. exploratory decomposition)

Understanding scaling

- Complex relationships among application parameters and the number of processors
 - Communication to computation ratio
 - Load balance
 - Temporal and spatial locality
- Fixed size problem -> problematic
- Understand : **How problem size influences performance evaluation?**
- Desirable: **Scale problem size as machine sizes grow**

Terms

- **Scale a machine** = make it more (or less) powerful
 - Scale up: add cores
 - Scale down: remove cores
 - Does the architecture's performance scale?
- **Problem size** = specific problem instance or input configuration
 - Vector of input parameters: Ocean $V = (N, e, dt, T)$
 - Different from *data size* or *memory usage*

Key issues in scaling

- *Under what constraints should the problem be scaled?*
 - some property must be kept fixed as the machine scales
 - Ex: data set, execution times, matrix rows per processor
- *How should the problem be scaled?*
 - Problem size determined by more than one parameter
 - Change the problem parameters to meet the constraint
- Simplification on problem size = N (one parameter)

Scaling models

- Properties used as basis for scaling constraints:
 - User-oriented: matrix rows per processor, transactions per processor, ...
 - Resource-oriented: execution time, used memory, ...
->more general, can be used across domains
- Resource-oriented models:
 - Problem constrained (PC): fixed problem size
 - Time constrained (TC): fixed execution time of program
 - Memory constrained (MC): fixed memory usage per processor

Problem constrained scaling

- Fixed problem size
- Focus: **use machine to solve the same problem faster**

$$speedup = \frac{time(1\,proc)}{time(P\,proc)}$$

- Problems:
 - Small problems on large machines
 - Large problems on too small machines
- Examples?

Time constrained scaling

- Fixed execution time
- Focus: **complete more work in a fixed amount of time**

$$speedup = \frac{WorkDoneByPprocs}{WorkDoneBy1proc}$$

- How to measure work?
 - Execution time on a single processor (problem: superlinear speedup)
 - **Ideally, a measure of work is:**
 - Simple to understand (architecture independent)
 - Scales linearly with sequential run time

Time constrained scaling examples

- Real-time 3D graphics: more compute power allows for rendering of much more complex scene
- Modern web sites
 - want to generate complex page, respond to user in X milliseconds
 - studies show site usage directly corresponds to page load latency

Memory constrained scaling

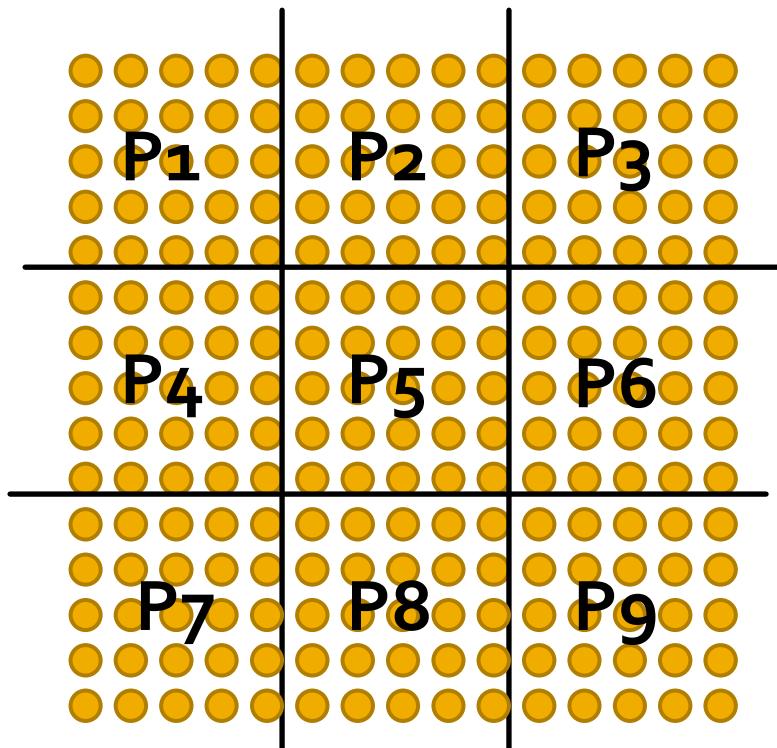
- Fixed memory usage per processor
- Focus: **run the largest problem possible without overflowing main memory**

$$speedup = \frac{WorkPerUnitTimeOnPprocs}{WorkPerUnitTimeOn1proc}$$

- Assumptions: memory resources scale with processor count
- Observations:
 - Superlinear speedup
 - Largest speedup
 - Execution time (parallel execution) can become too large
- Example: very large scientific computations, physical simulation

Solver example (PC)

- For $N \times N$ grid:
 - Memory requirement: $O(N^2)$
 - Total work: $O(N^2)$ grid elements $\times O(N)$ convergence
 $= O(N^3)$
- **N is fixed**
- Execution time: $O(1/P)$
- Elements per processor: N^2/P
- Available concurrency: fixed P
- **Comm-to-comp ratio:**
 $O(P^{1/2})$



Solver example (TC)

- Execution time: fixed at $O(N^3)$
- Let scaled grid size be $K \times K$
- Assume linear speedup: $K^3/P = N^3$ (so $K = NP^{1/3}$)
- Elements per processor: $K^2/P = N^2/P^{1/3}$
- Communication per processor: $K/P^{1/2} = O(1/P^{1/6})$
- Comm-to-comp ratio: $O(P^{1/6})$

Solver example (MC)

- Elements per processor: fixed (N^2)
- Let scaled grid size be $NP^{1/2} \times NP^{1/2}$
- Execution time: $O((NP^{1/2})^3/P) = O(P^{1/2})$
- Comm-to-comp ratio: fixed at $1/N$

- Notice:
 - Best speedup
 - Execution time increases

Summary

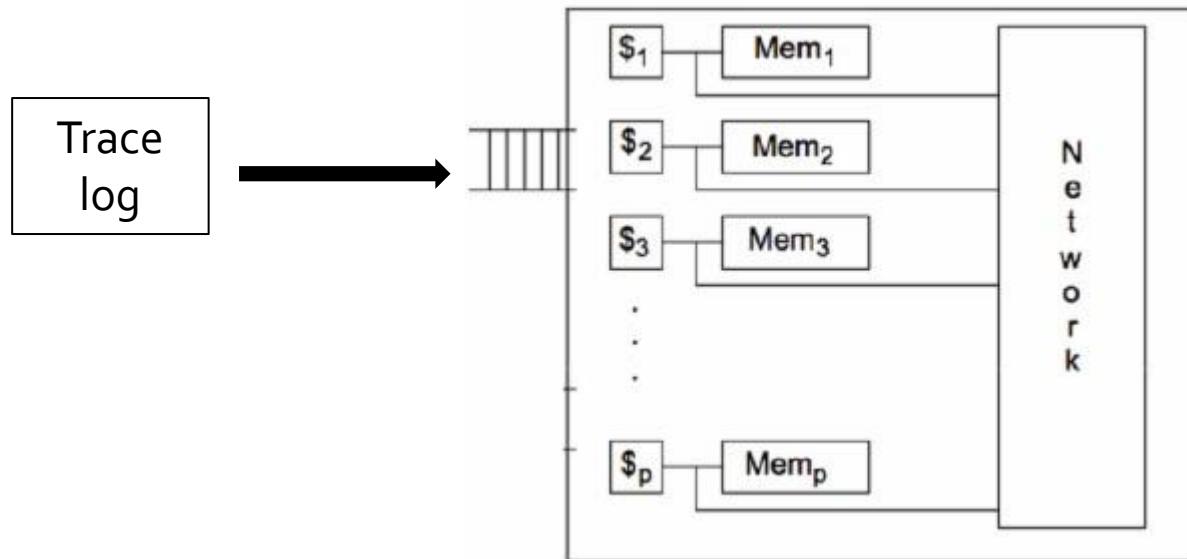
- Performance improvement due to parallelism is measured by speedup
- Speedup metrics take different forms for different scaling models
- Which model matters most is application/context specific
- Behavior of a parallel program depends significantly on the scaling properties of the problem and also the machine

Evaluating an architectural idea: simulation

- Architects evaluate architectural decisions quantitatively using hardware performance simulators
 - Evaluate new feature:
 - Run simulation with new feature
 - Run simulation without new feature
 - Compare
 - Simulate against a wide collection of benchmarks
- Design detailed simulator to test new architectural feature
 - Detailed simulation is expensive
 - Often cannot simulate full machine configurations or realistic problem sizes (must scale down workloads significantly!)
 - Does scaled-down simulation predict reality?

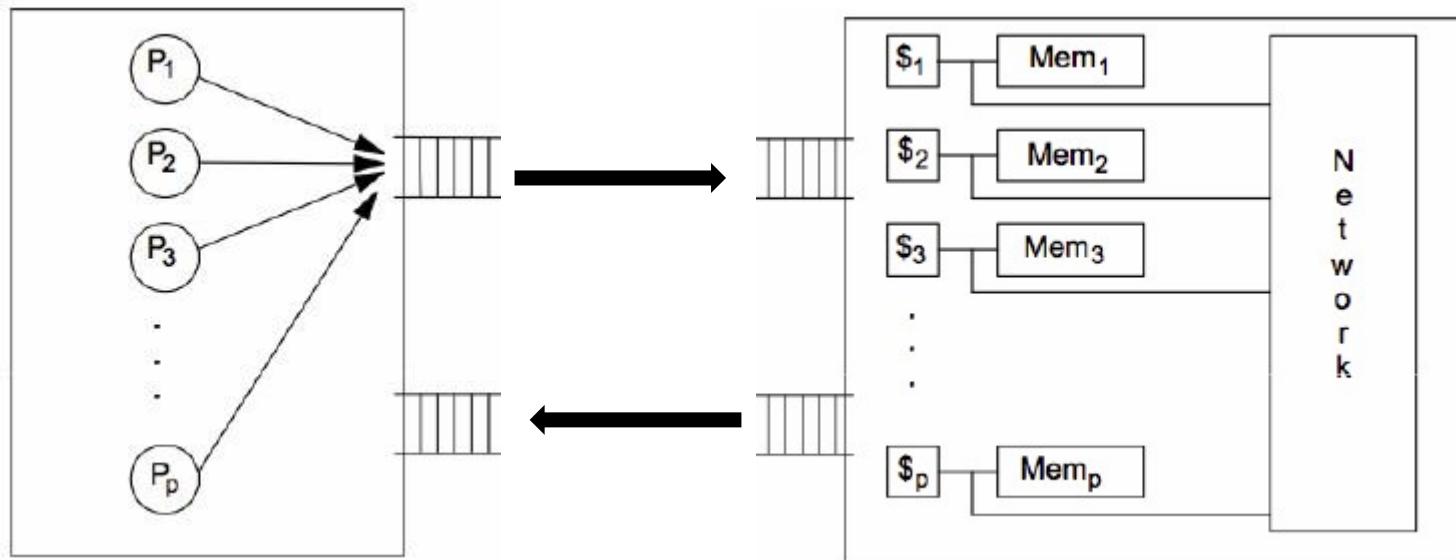
Trace-drive simulation

- Record memory access during real execution
 - Modify the program to keep trace of memory access
 - Use tools (Intel's PIN)
- Play the recording on simulator



Execution-driven simulation

- Executes simulated program in software
 - Simulated processors generate memory references, which are processed by the simulated memory hierarchy
- Performance of simulator is typically inversely proportional to level of simulated detail
 - How detailed should the simulation be?



Conclusions

- Tricky to evaluate and tune parallel software and parallel machines
- Easy to obtain misleading results
- Start by precisely stating your application performance goals
- Determine if your evaluation approach is consistent with these goals

Guidelines for evaluating performance of parallel SW

- Try the simplest parallel solution first, then measure performance
- Determine if your performance is limited by computation, memory bandwidth (or memory latency), or synchronization
- What's the best you can do in practice?

Roofline model

- Use microbenchmarks to compute peak performance (computation, memory bandwidth)
- Compare application's performance to known peak values
- Measure:
 - Only computation
 - Only data access
 - Increase of performance based on data locality
 - Program without synchronization

Use profilers/performance monitoring tools

- All modern processors have low-level event “performance counters”
 - details such as: instructions completed, clock ticks, L2/L3 cache hits/misses, bytes read from memory controller, etc.
- Profilers: Intel VTune

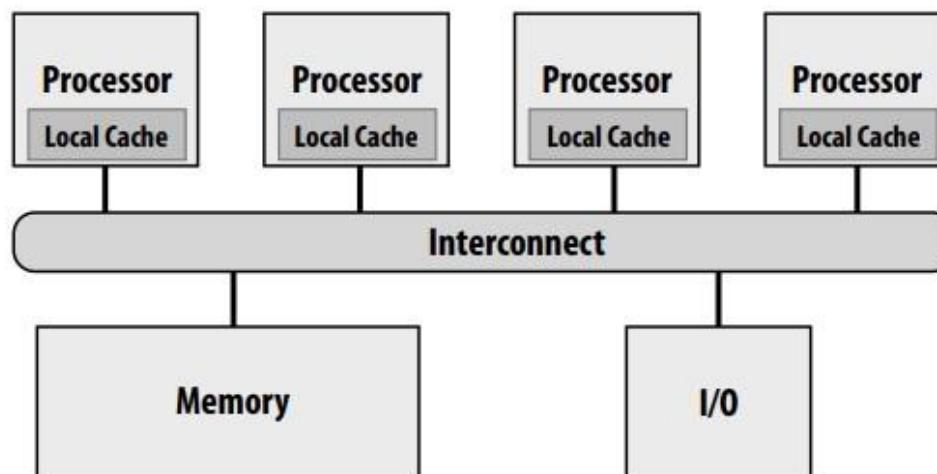
Cache coherence

Bibliography

- Culler et. al. – Chapters 5.1-5.4; 8.1-8.3

Shared memory multiprocessor

- Symmetric Multiprocessor or SMP
- Hardware directly supports the shared address space programming model
- Message passing programming model supported by an intervening software layer
- Multiprogramming support in OS



Shared memory multiprocessor

- Processors read and write to shared variables
- Expected: a read from address X will return the last value that was written at address X by any processor
- Processors replicate the contents of memory in cache
- It is possible that processors will observe a different value for the same memory location

Example

Proc						Cache P ₁	Cache P ₂	Cache P ₃	Main M
P ₁	Read u					U=5			U=5
P ₃		Read u				U=5		U=5	U=5
P ₃			Write U=7			U=5		U=7	U=7
P ₁				Read u		U=5			U=7
P ₂					Read u		U=7		U=7

- 3 processors SMP
- Assume write-through policy
- Question: What happens in the case of write-back policy?

Memory coherence problem

- A read from address X will return the last value that was written at address X by any processor
- Problem: the memory system consists of a shared main memory and replicated private caches -> abstraction of a single shared memory space

Single CPU coherence problem

- DMA data transfers
- CPU and I/O use shared buffers
- Problem: data written in buffer is in cache and is not updated in the main memory -> I/O reads old data
- Solution: do not cache pages that contain shared buffers (OS solution)

Coherence-definition

- A memory system is coherent if the results of any execution of a program are such that, for each location, it is possible to construct a hypothetical serial order of all operations to the location (i.e., put all reads/writes issued by all processors into a total order) which is consistent with the results of the execution and in which
 - operations issued by any particular processor occur in the above sequence in the order in which they were issued to the memory system by that processor, and
 - the value returned by each read operation is the value written by the last write to that location in the above sequence.

Coherence

- Write propagation: notification of a write has to reach all processors
- Write serialization: all writes to a location are seen in the same order by all processors.
- Important: the order in which things appear to happen, as detectable from the results of an execution

Shared caches

- Eliminates the problem of replication
- Problem: scalability
- Benefits:
 - One processor may pre-fetch cache lines for other processors
 - Facilitates fine-grained sharing

Memory consistency

P1

```
/* Assume initial value of A and flag is 0 */
```

```
A = 1;
```

```
while (flag == 0); /* spin idly */
```

```
flag = 1;
```

P2

```
print A;
```

- Which will be the result of the execution?
- What change will P2 see first? A or flag?
 - Update order of A and flag NOT implied by coherence
 - Need for an ordering model...

Memory consistency model

- A memory consistency model specifies constraints on the order in which memory operations must appear to be performed
- Sequential Consistency: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program.
 - Memory appears to service the requests of memory access one at a time in an interleaved manner according to an arbitrary schedule
 - Operations are atomic: the results appear to all processors after completion

Sufficient conditions for preserving sequential consistency

1. Every process issues memory requests in the order specified by the program
2. After a write operation is issued, the issuing process waits for the write to complete before issuing its next operation
3. After a read operation is issued, the issuing process waits for the read to complete, and for the write whose value is being returned by the read to complete, before issuing its next operation.

Cache coherence through bus snooping

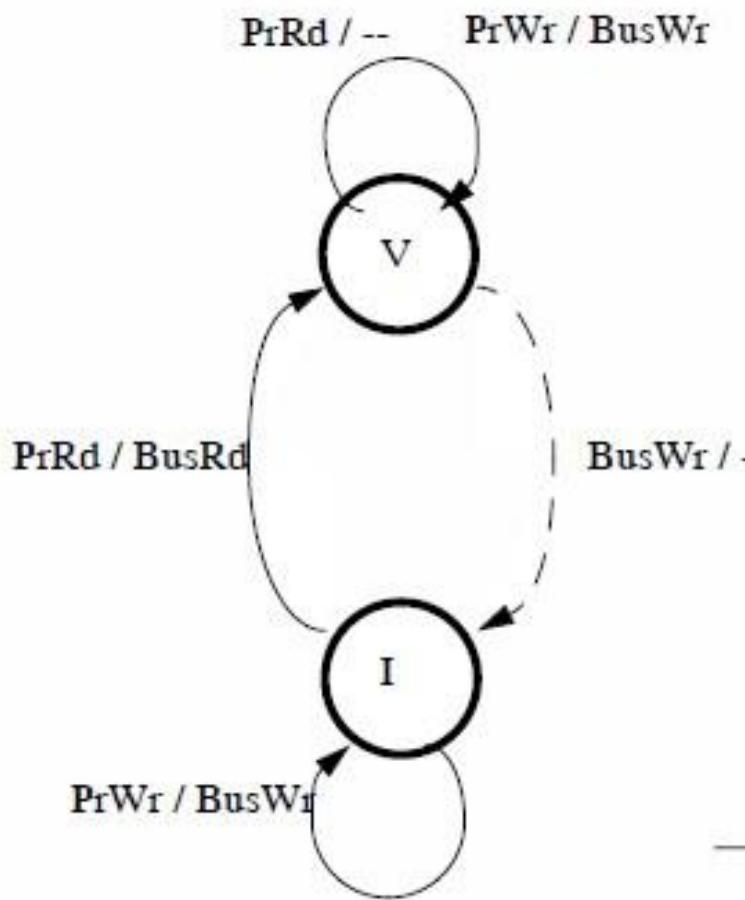
- Cache controllers ‘snoop’ on the bus and monitor the transactions
- Cache controller takes action if a bus transaction is relevant
 - Invalidate cache line
 - Update value in own cache
- Snooping-based cache coherence scheme ensures that:
 - all “necessary” transactions appear on the bus
 - each cache monitors the bus for relevant transactions and takes suitable actions.

Simple snoop-based implementation

- Assume write-through
- Cache line state: V-valid; I-invalid
- Cache controller invalidates the cache line if it detects a write transaction

Proc						Cache P ₁	Cache P ₂	Cache P ₃	Main M
P ₁	Read u					U=5(V)			U=5
P ₃		Read u				U=5(V)		U=5(V)	
P ₃			Write U=7			U=5(I)		U=7(V)	U=7
P ₁				Read u		U=7			
P ₂					Read u		U=7		

Write-through invalidation - state diagram



- States of cache line:
 - Valid (V)
 - Invalid (I)
- Processor requests:
 - reads (PrRd)
 - writes (PrWr)
- Bus transactions:
 - BusRd: The cache controller puts the address on the bus and asks for a copy
 - BusWr: The cache controller puts the address and the contents for the memory block on the bus

→ Processor initiated transactions
--> Bus-snooper initiated transactions

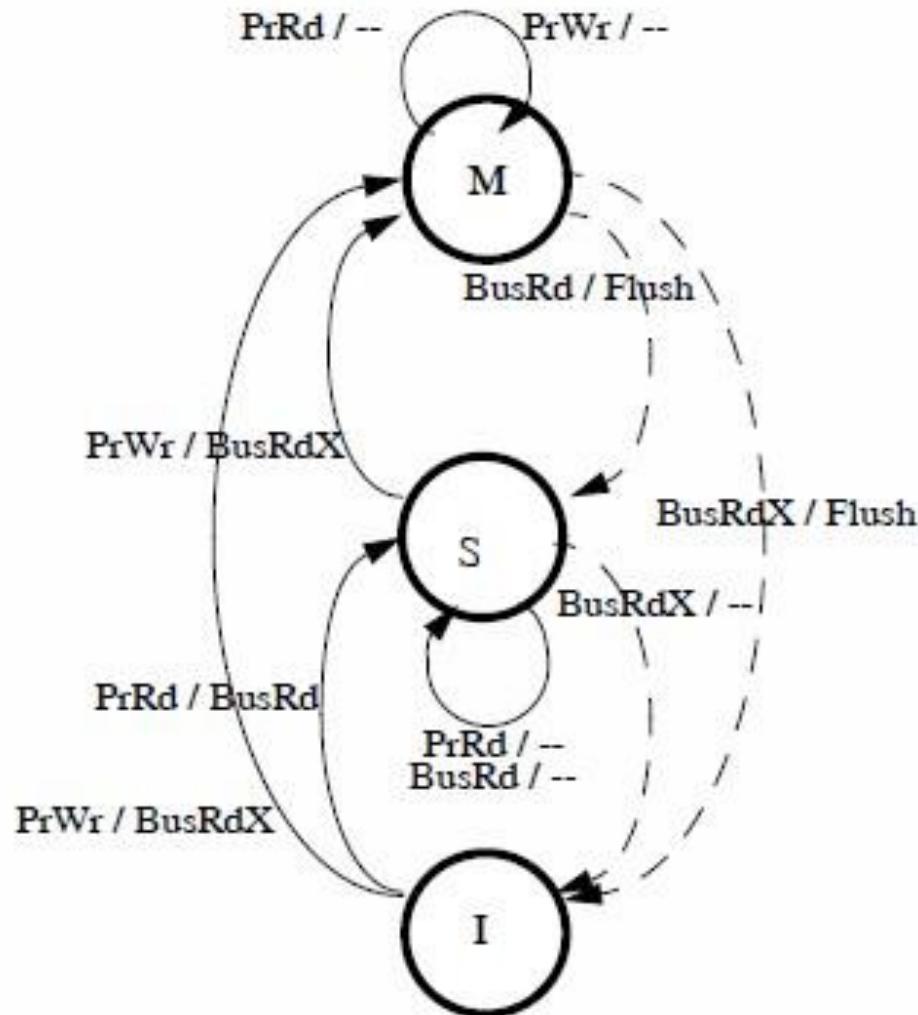
Write-through invalidation

- Assumptions
 - Writes visible to all cache controllers in the same order
 - Memory transactions are atomic
 - Invalidation applied immediately after receiving broadcast
- Write-through: inefficient
- Write-back: how to insure write propagation?

MSI- write-back invalidation protocol

- States of cache line:
 - Modified (M) – only this processor has a valid copy
 - Shared (S) - zero or more other caches may also have an up-to-date (shared) copy
 - Invalid (I)
- Processor requests:
 - reads (PrRd)
 - writes (PrWr)
- Bus transactions:
 - Bus Read (BusRd): The cache controller puts the address on the bus and asks for a copy that it does not intend to modify
 - Bus Read-Exclusive (BusRdX): The cache controller puts the address on the bus and asks for an exclusive copy that it intends to modify
 - Writeback (BusWB): The cache controller puts the address and the contents for the memory block on the bus

MSI-state diagram



MSI protocol

- Write propagation:
 - When in M state, flush line if any read from other processor
 - Invalidate if read-exclusive
- Write serialization
 - Writes in the order of BusRdX
 - Reads in order of BusRd
 - Other writes only in M state

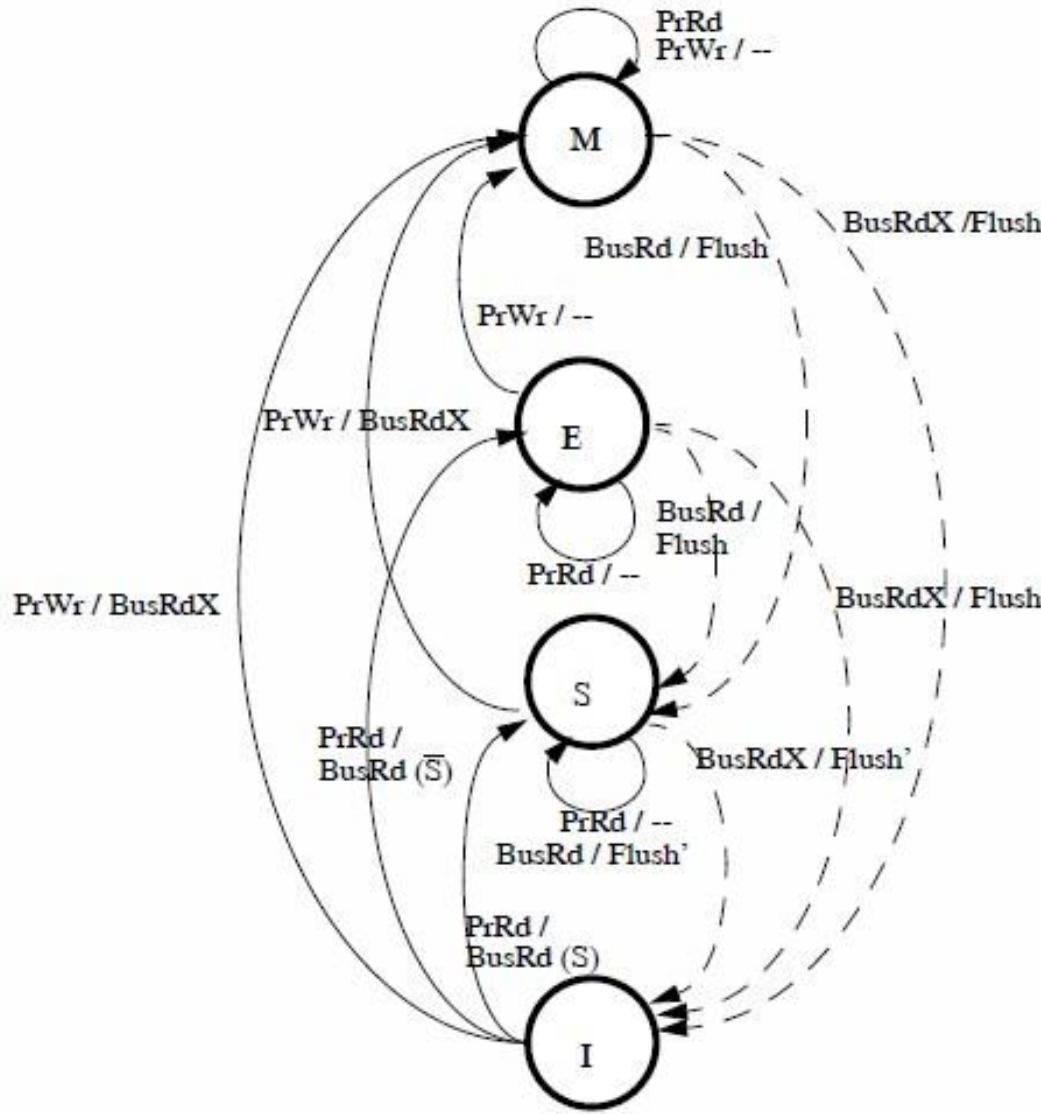
Example

<u>Proc. Action</u>	<u>State in P1</u>	<u>State in P2</u>	<u>State in P3</u>	<u>Bus Action</u>	<u>Data Supplied By</u>
1. P1 reads u	S	--	--	BusRd	Memory
2. P3 reads u	S	--	S	BusRd	Memory
3. P3 writes u	I	--	M	BusRdX	Memory
4. P1 reads u	S	--	S	BusRd	P3's cache
5. P2 reads u	S	S	S	BusRd	Memory

MESI- write-back invalidation protocol

- MSI inefficiency: process reads and then modifies data
 - Two transactions: BusRd (I->S), BusRdX(S->M)
 - Even if data is not shared
- Save the second transaction by introducing new state: exclusive (no other processor is caching the block)
- States:
 - Modified (M)
 - Exclusive-clean (E)
 - Shared (S)
 - Invalid (I)
- The same bus transactions

MESI – state diagram

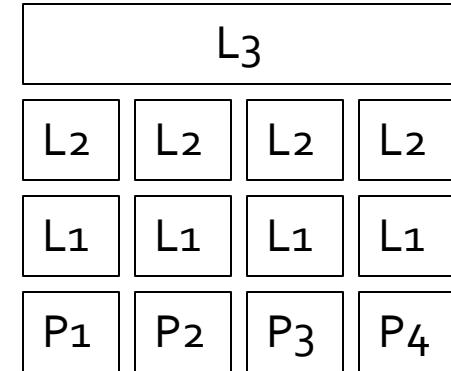


Lower-level choices

- Who should supply the block for a BusRd transaction when both the memory and another cache have a copy of it?
 - Cache-to-cache sharing
- Higher complexity of implementation, but faster (reduces latency and required memory bandwidth)

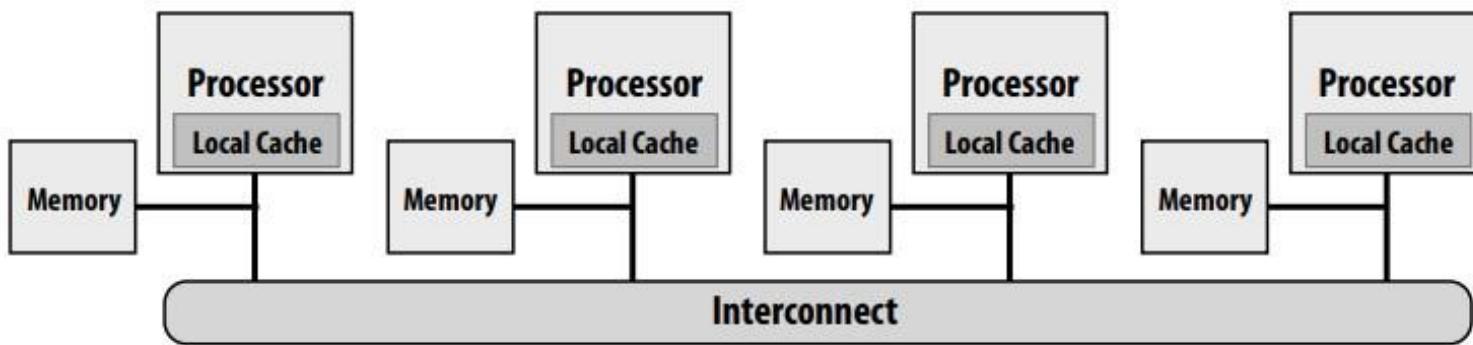
Real case

- Multi-level cache hierarchies
- Changes made in L₁ may not be visible for cache controller in L₂
- Inefficient to snoop buses independently
- **Scalability of snoop-based protocols is limited by the ability to broadcast coherence messages to all caches**



Large machines

- Locating regions of memory near processors increases scalability (NUMA)
- cc-NUMA: cache-coherent non-uniform memory access

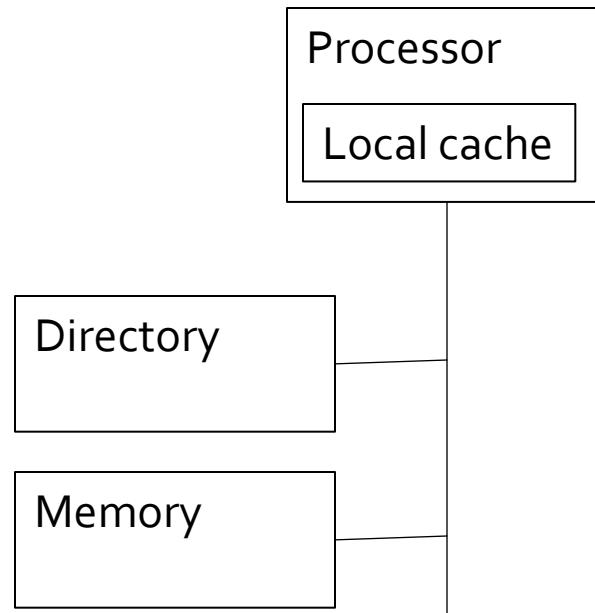


Directory-based cache coherence

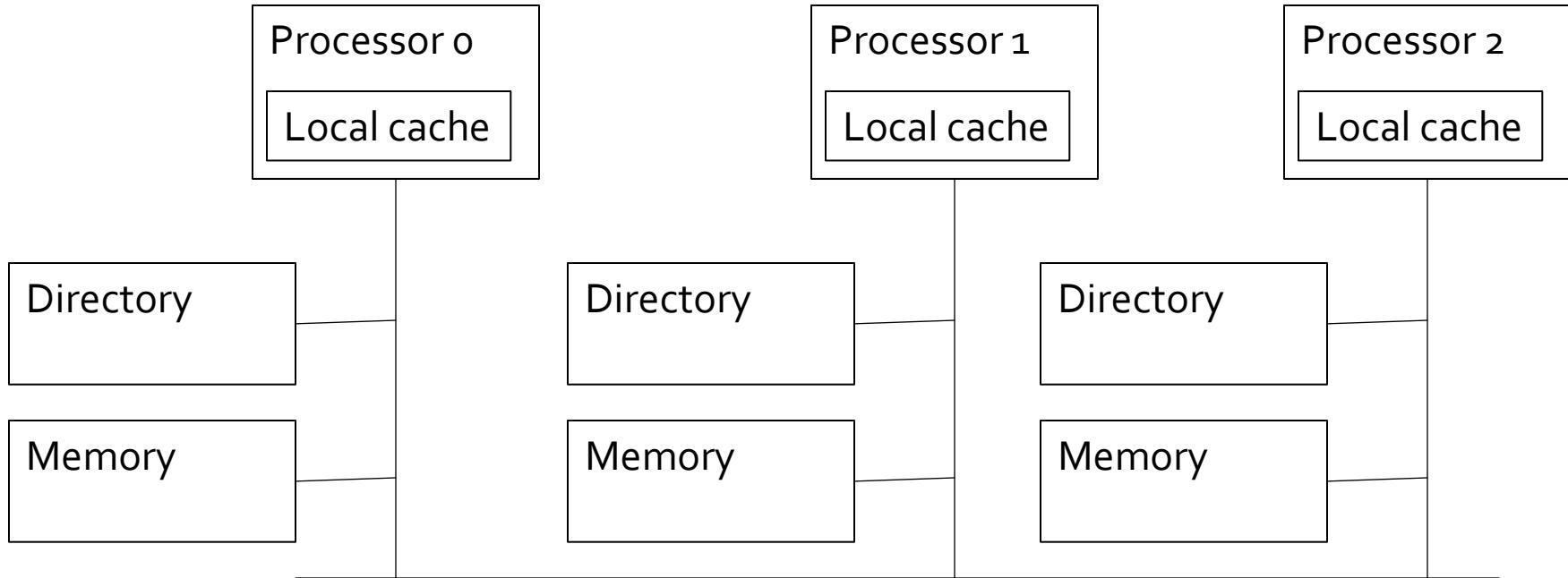
- Avoid broadcast by storing the information about the status of a line in cache in a directory
- A directory entry for a cache line contains info about the state of the cache line in all caches
- Caches look for info in the directories
- Cache coherence is maintained by point-to-point messages between caches

Simple directory

- Directory:
 - Presence bit (one for each processor)
 - Dirty bit (indicates line dirty in one of the processors' caches)
 - One directory entry for cache line of memory
- Cache line states (MSI or MESI)

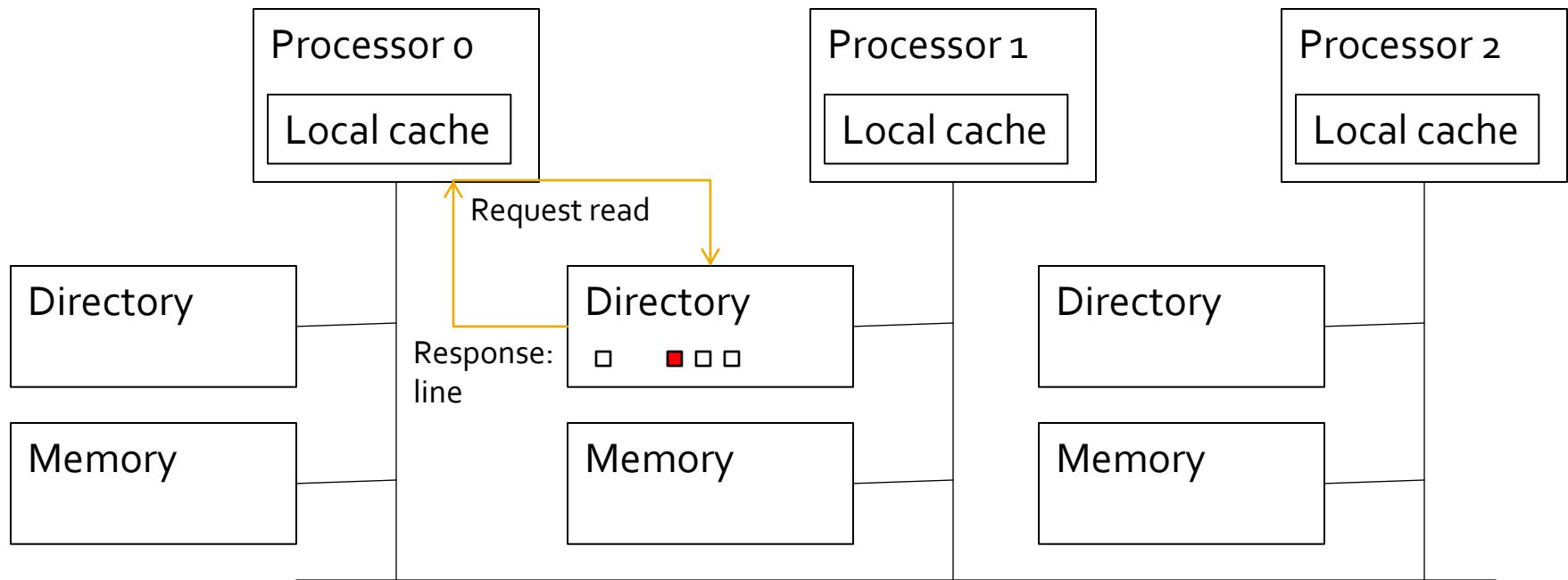


Distributed directory

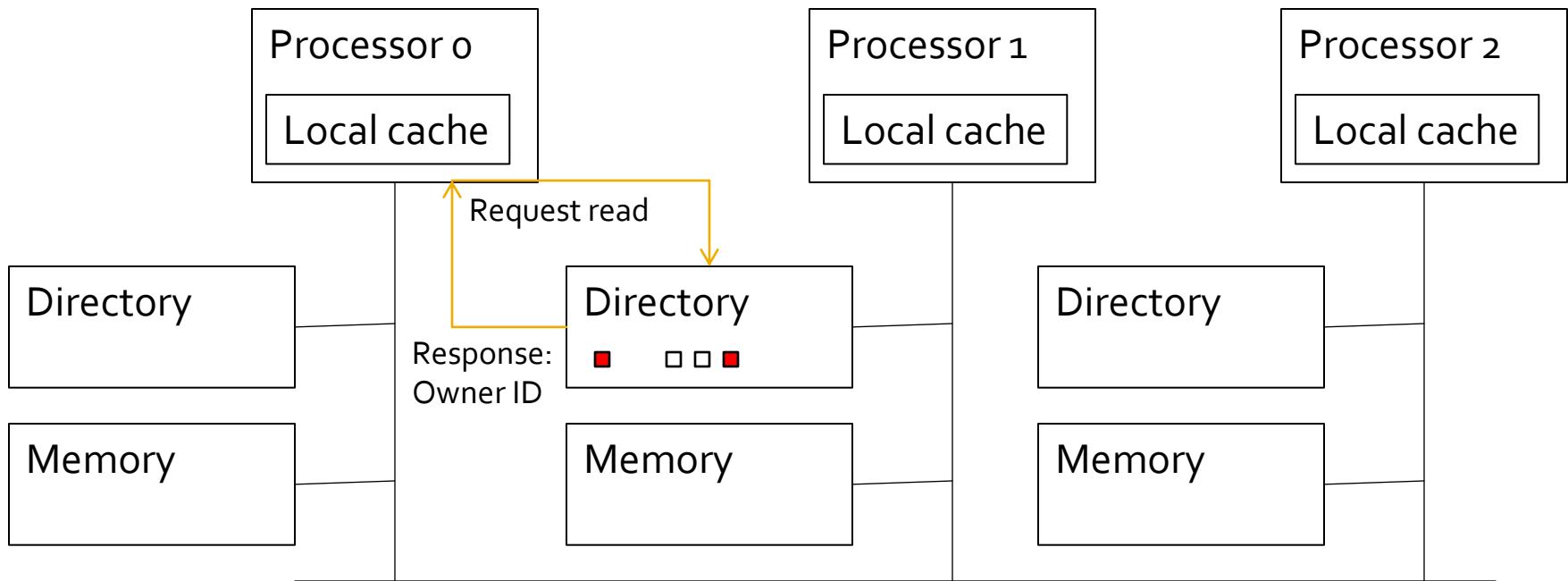


- Home node – node containing the memory holding data (cache line)
- Request node – node containing processor requesting cache line

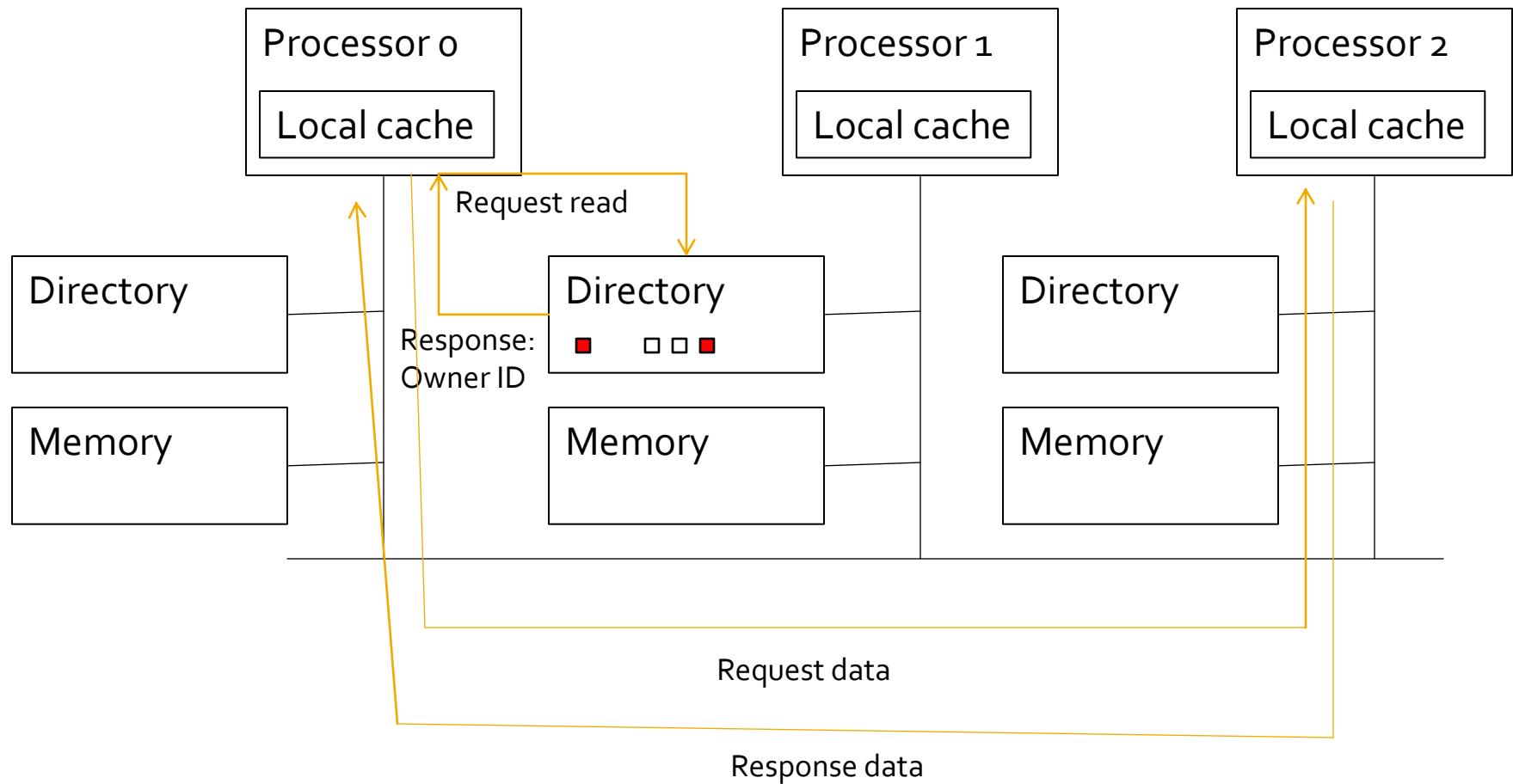
Example: read miss to clean line



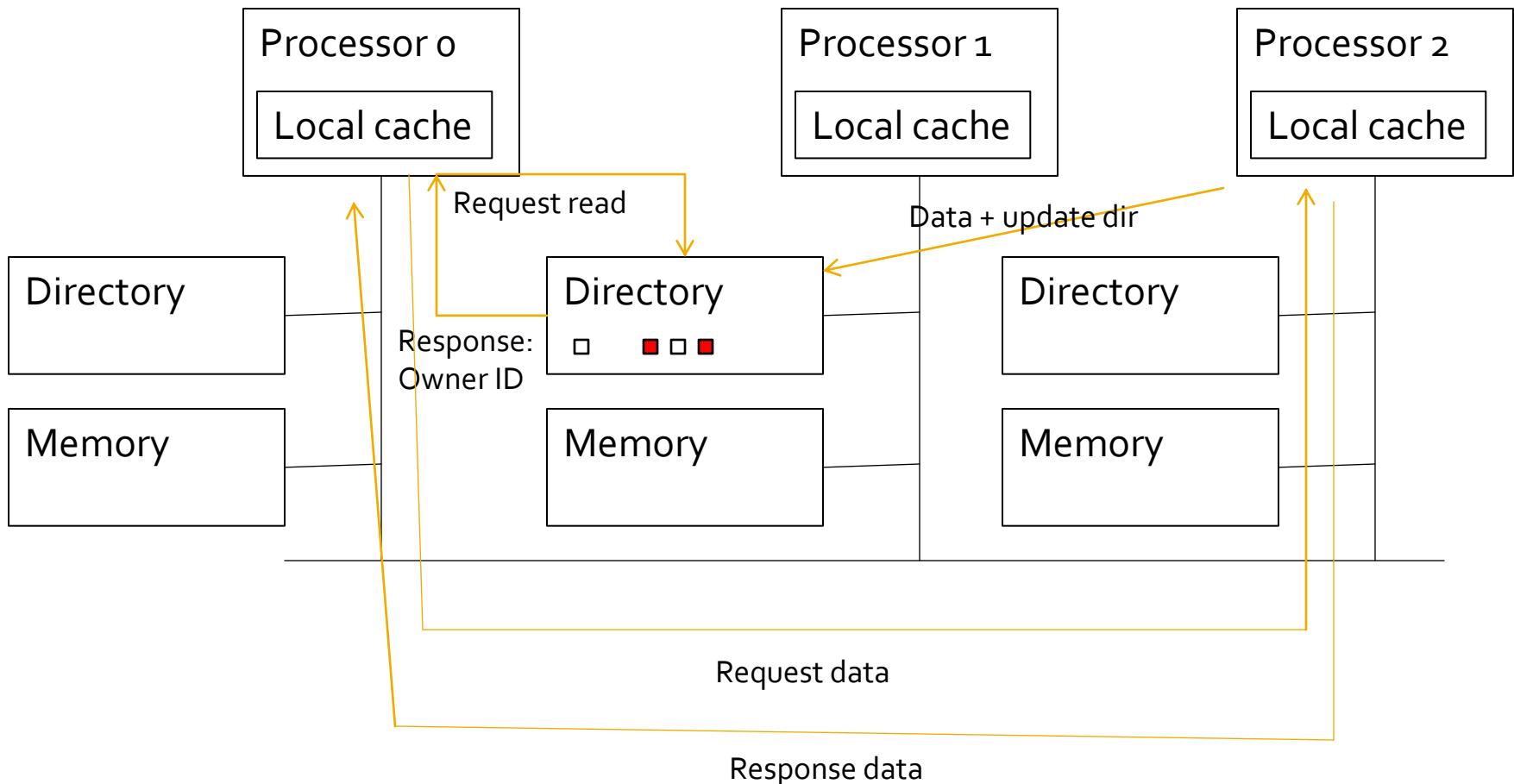
Example: read miss to dirty line



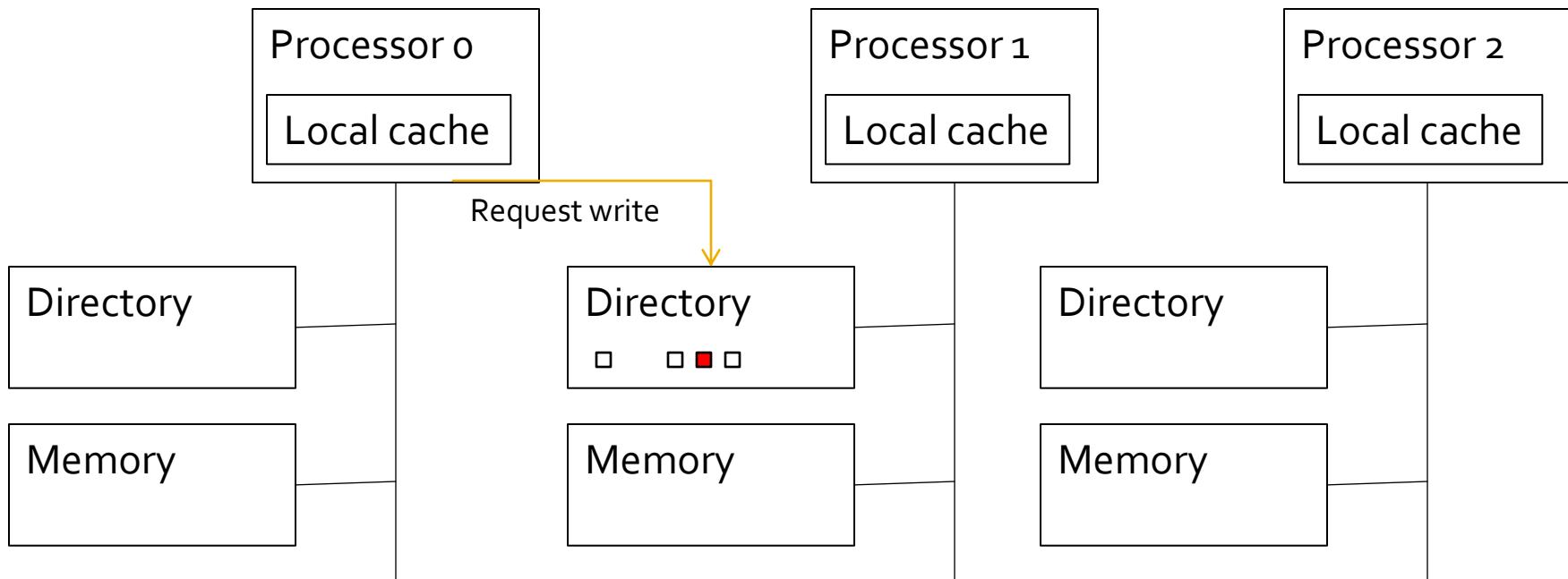
Example: read miss to dirty line



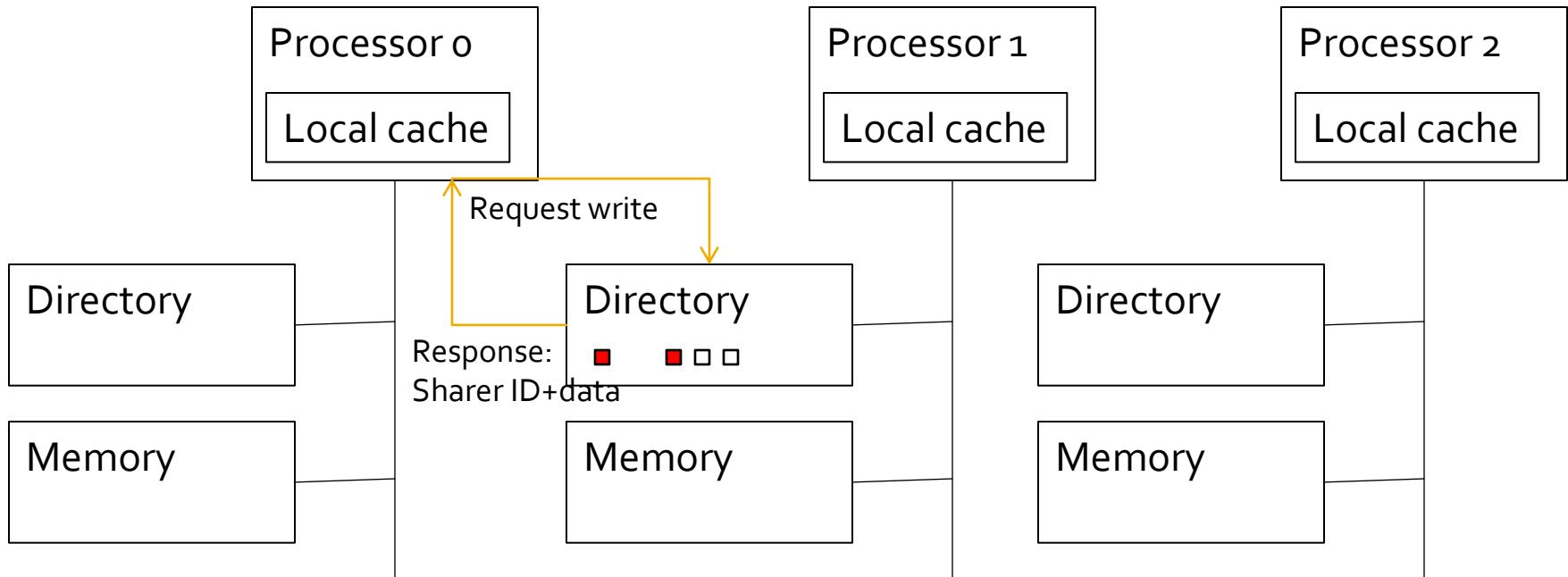
Example: read miss to dirty line



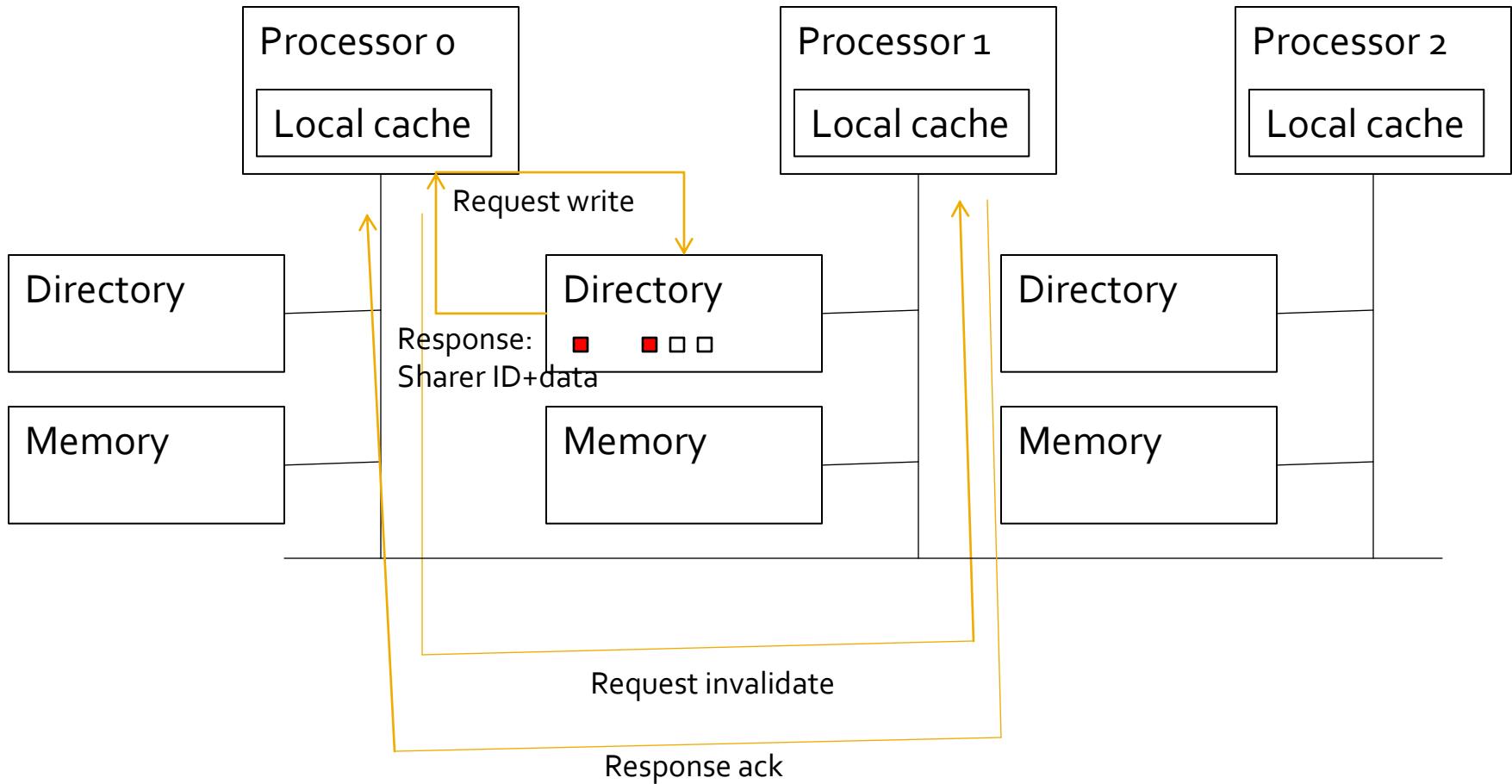
Example: write miss



Example: write miss



Example: write miss



Advantage of directories

- Reads:
 - Directory tells requesting node where to get line from
 - Point-to-point communication
- Writes:
 - Depends on the number of shares
 - If all caches share data, just like snooping protocol

Time in Distributed Computing

References

Kshemkalyani: chapter 3
Coulouris: chapter 14

Contents

1. Problem definition
2. Physical clocks synchronization
 - 2.1. Cristian algorithm
 - 2.2. Berkeley algorithm
 - 2.3. Network Time Protocol
3. Logical clocks
 - 3.1. Scalar time
 - 3.2. Vector time
 - 3.3. Efficient implementation of vector clocks - Singhal-Kshemkalyani

Problem definition

- Causality between events - fundamental issue for distributed / parallel systems
 - often in a distributed system it is necessary to be able to tell the order in which the events take place
- Causality (causal precedence relation) shows the relationship between two events (the cause and the effect)
 - the second event is a consequence of the first
- Causality is used in
 - distributed debugging
 - establishment of global breakpoints
 - implementations of causal ordering communication
 - determining the consistency of checkpoints in recovery

Problem definition

- Causality can be tracked using
 - Physical time
 - Synchronized computer clocks?
 - Logical clock
 - scalar time
 - vector time
 - matrix time

Physical clock

- Two widely used (physical) time standards
 - atomic time: the frequency of atomic oscillations
 - almost zero clock drift
 - most accurate time
 - astronomical time: the rotation of the earth
 - rotation of earth is not uniform
- International Atomic Time (TAI)
 - high-precision atomic coordinate time standard based on the notional passage of proper time on Earth's geoid.
- Coordinate Universal Time (UTC)
 - international standard for timekeeping
 - based on atomic time, but with some insertion of leap seconds
 - broadcast from radio stations
 - computers with receivers synchronize their clocks with broadcast timing signals

Physical clock

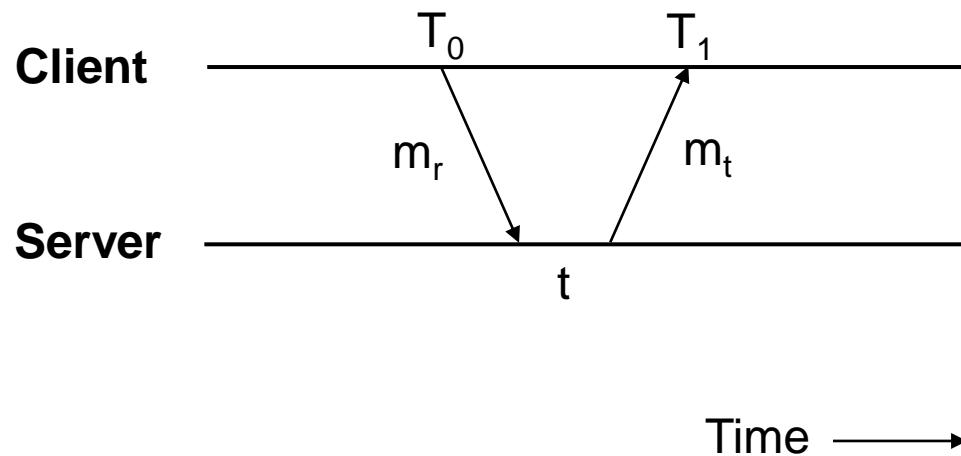
- Clock of computers may differ
 - difficult to set the same time for two clocks
 - suppose clocks are set accurately, but after a time due to the clock drift, they will differ → difficult to maintain synchronization between two clocks
- External synchronization
 - clock of a computer is synchronized with an external reference time source
- Internal synchronization
 - clocks of a collection of computers are synchronized with one another
- Local clock behind reference clock
 - local clock needs to “hurry up”
 - adjustment in one leap or in a sequence of smaller leaps
- Local clock ahead reference clock
 - local clock needs to “slow down”
 - adjustment by ignoring some clock ticks

Physical clock

- Why clock synchronization is not easy?
 - most of the computers does not have receivers (due to the cost)
 - synchronization has to be done periodically due to the clock drift
 - if a computer clock is ahead of the reference, it cannot be simply set back
 - clock synchronization can be done only by message-passing
- Why clock synchronization is needed?
 - analysis of the log information and debugging information collected from different devices in network management
 - all devices must use the same reference clock in a charging system
 - multiple systems process a complex event in cooperation

Physical clock - Cristian

- Time server maintains its clock using a UTC source
- A computer within the system synchronizes using the time server as reference

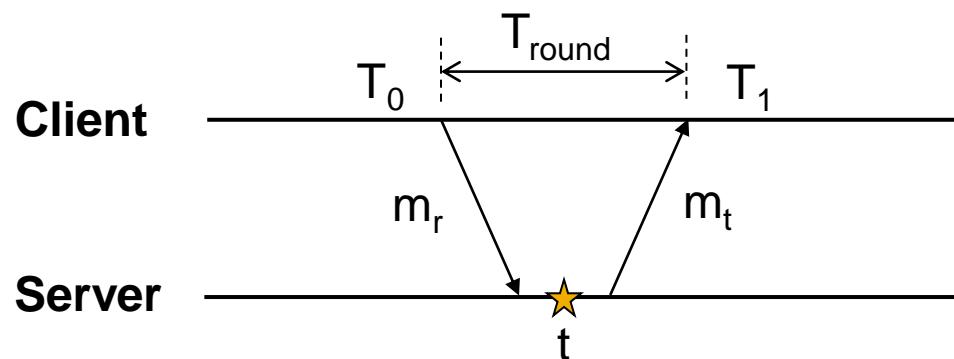


Physical clock - Cristian

- Client request the time by sending message m_r and gets time from the server in message m_t
- Client should set the clock to $t + T_{trans}$ where T_{trans} is time needed to transmit m_t from server to client
- T_{trans} varies and cannot be measured
- All we know about T_{trans} is
 - $T_{trans} = T_{min} + T_x$
 - T_{min} is time of message transmission in perfect conditions (e.g. no other messages send, no other requests to the server etc)

Physical clock - Cristian

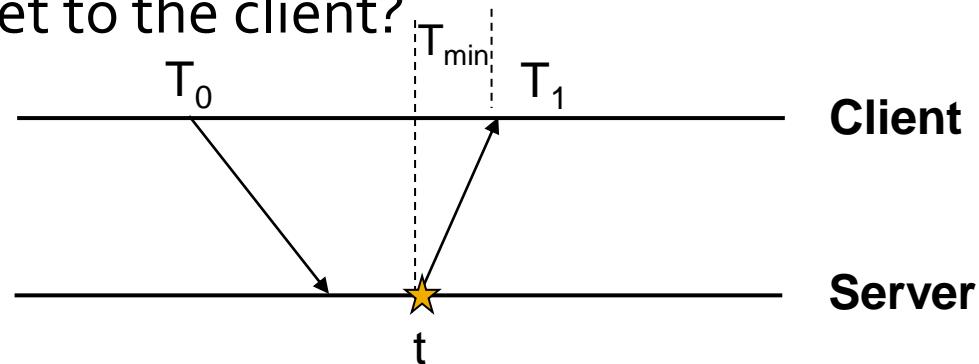
- Client records the total round-trip time T_{round}
 - $T_{\text{round}} = T_1 - T_0$
- When m_t is received by client, its clock is set to
 - $t_c = t + T_{\text{round}}/2$
 - based on the assumption that the elapsed time is equally divided before and after server put t in m_t



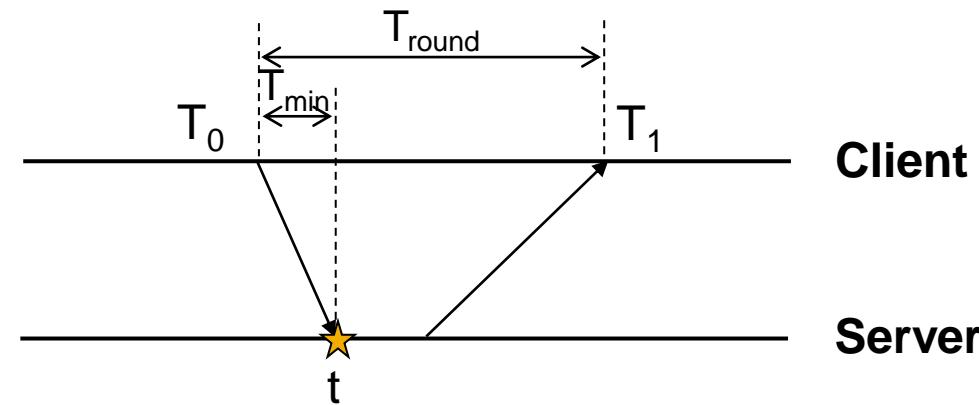
Physical clock - Cristian

- How accurate is the time set to the client?

$$t_c = t + T_{\min}$$



$$t_c = T_{\text{round}} - T_{\min} + t$$



Client clock $[t + T_{\min}, t + T_{\text{round}} - T_{\min}]$

Range width $T_{\text{round}} - 2T_{\min}$

Physical clock - Cristian

- Time at the client has an accuracy of
 $\pm((T_{\text{round}}/2) - T_{\text{min}})$
- Client clock should be
 $t + (T_{\text{round}}/2) \pm ((T_{\text{round}}/2) - T_{\text{min}})$
- How to increase the accuracy of client clock?
 - keep track of T_{round} and use the minimum value of T_{round}

Physical clock - Cristian

- Disadvantages
 - server is bottleneck / point of failure
 - use a collection of synchronized time servers
 - client sends the request to all servers and use the first reply
 - malfunctioning / impostor time server
 - detection
 - authentication

Physical clock - Berkeley

- No time server maintains the reference clock
- Purpose of time server: maintain an average time as the global time
 - used to synchronize clocks that have different rates

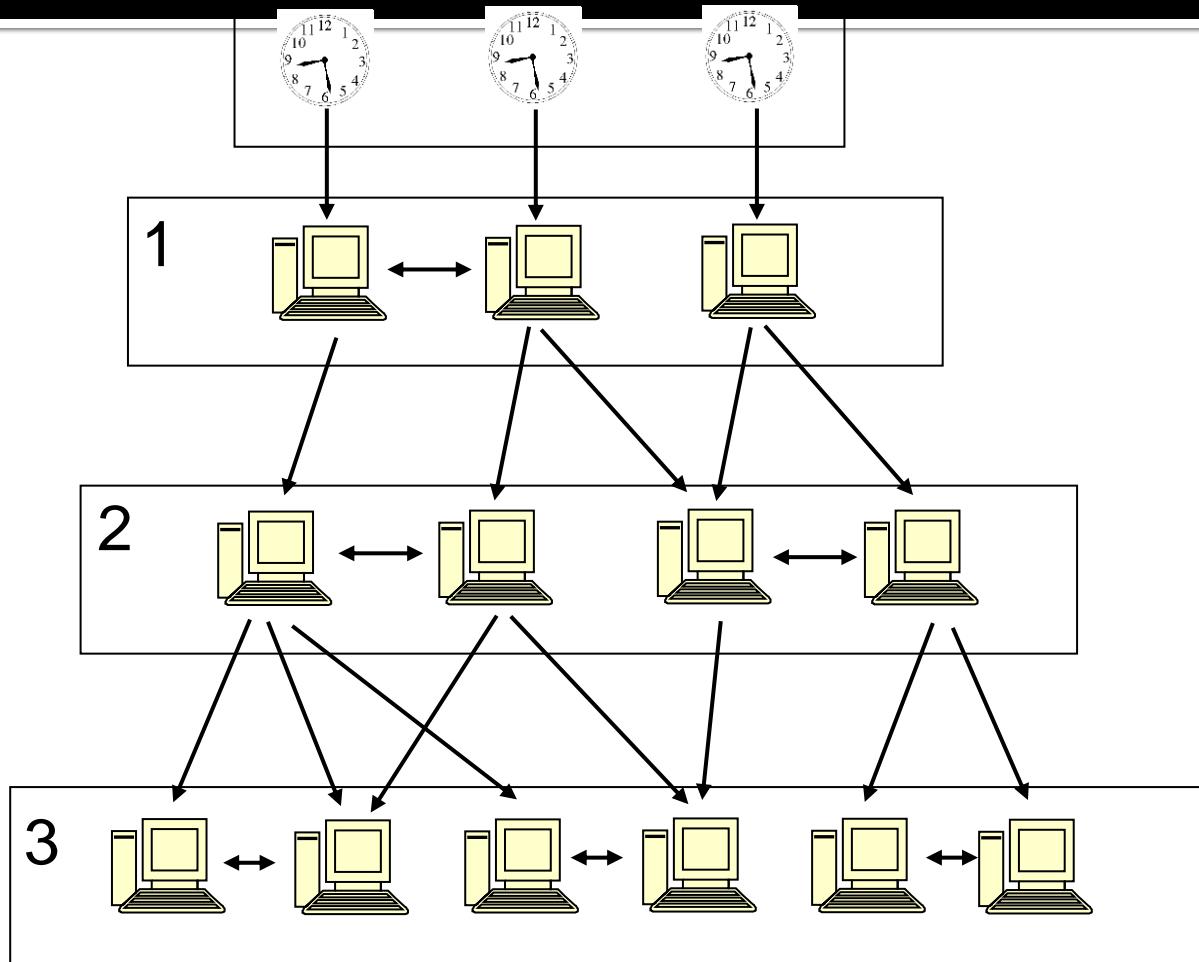
Physical clock - Berkeley

- A time server
 - polls the client in order to get their time
 - Clients reply with their time
 - estimates the client time as in Cristian's algorithm $(t + T_{\text{round}}/2)$
 - computes the average clock time
 - clocks whose values is outside an interval is not used for computing the average
 - sends back to the clients the amount (positive or negative) that each must adjust its clock with

Physical clock

- Cristan: $t_c = t + T_{\text{round}}/2$
- Berkeley: $t_c = \text{average of computers clocks}$
- Is there a better way to compute the time?
- Better?
 - more accurate – for this you have to get a better estimates of delays
 - works despite the network failures
- NTP – Network Time Protocol

Physical clock - NTP



Physical clock - NTP

- Internet standard protocol
- Hierarchical organization in stratum
 - stratum 1 – primary servers (synchronized directly to reference clocks)
 - stratum 2 – synchronized with primary servers and between them
- If some servers fail the synchronization subnet can reconfigure
 - a primary that loses its source becomes secondary
 - a secondary that loses its primary can use another primary

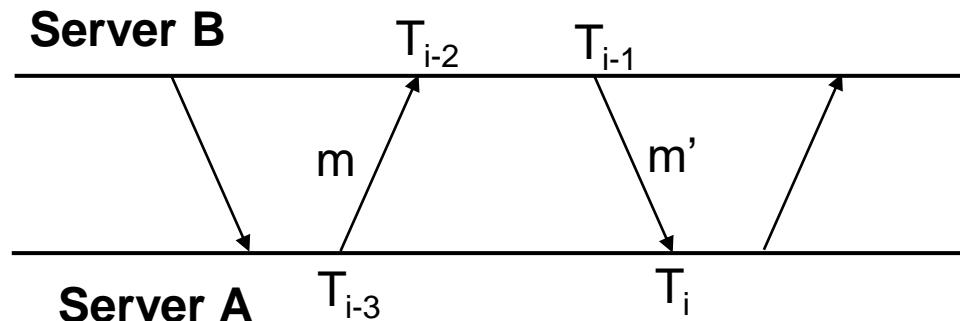
Physical clock - NTP

- Methods for synchronization
 - Symmetric mode
 - Multicast mode
 - Procedure call mode
 - ...

Physical clock - NTP

■ Symmetric mode

- host sends NTP messages to make known that it is willing to be synchronized with and to be synchronized by its peer
- typical scenario: a pair of servers exchange pairs of timing messages
 - each message sent contains timestamps of the previous message sent
- highest accuracy



Physical clock - NTP

- Symmetric mode
 - pairs of offset and delay are stored
 - t is the transmission time for m
 - t' is the transmission time for m'
 - O is the offset between clock A and clock B
 - $A(t) = B(t) + O$ $A(t), B(t)$ clocks at A and B

$$T_{i-2} = T_{i-3} + t + O$$

$$T_i = T_{i-1} - O + t'$$

assume $t = t'$

$$O_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2 \quad D_i = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$$

Physical clock - NTP

- Symmetric mode
 - the eight most recent pairs of (O_i, D_i) are stored
 - take minimum D_i - corresponding O_i will be used to estimate O_i
- a server interacts with several peers and identifies the most reliable ones
- accuracy 10s of msec / 1-50 msec over internet paths

Physical clock - NTP

- Multicast mode
 - used on high speed LAN
 - server sends time (at once) to all receivers in LAN
 - receivers reset their clocks
 - accuracy is not high
- Procedure call mode
 - clients send request to server
 - server replies with time
 - same behavior as Cristian
 - higher accuracy

Logical clocks

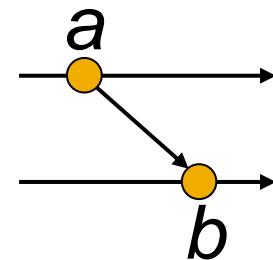
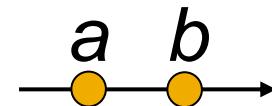
- Every process has a logical clock
 - logical clock assignes numbers to events, works as a counter, does not rely on physical time
 - every event has a timestamp
 - logical clock is modified following a set of rules
- Can causality relationship between events be inferred from their timestamps ? How?

Logical clocks

- Types of events in a distributed system:
 - Internal event: event that takes place at a process and does not affect other processes
 - Send event: a process transmits a message to another process
 - Receive event: a process gets a message from another process

Logical clocks

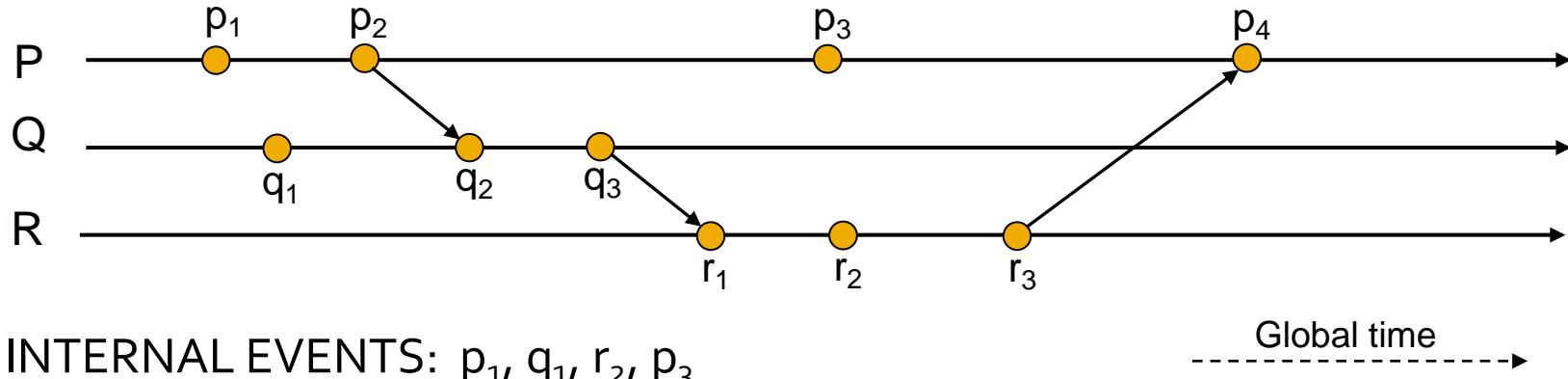
- Happened before relation is given by the following three rules:
 - if a and b are events in the same process and a comes before b , then a happened before b , $a \rightarrow b$
 - if a is sending of a message by one process and b is the receiving of the same message by another process then a happened before b , $a \rightarrow b$
 - if $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$



Logical clocks

- Happened before between two events **may** (but not necessarily) imply a causality relationship between the two events
- Happened before relation is:
 - irreflexive: $a \not\rightarrow a$
 - asymmetric: $a \rightarrow b \Rightarrow b \not\rightarrow a$
 - transitive: $a \rightarrow b$ and $b \rightarrow c \Rightarrow a \rightarrow c$

Logical clocks



INTERNAL EVENTS: p_1, q_1, r_2, p_3

SEND EVENTS: p_2, q_3, r_3

RECEIVE EVENTS: q_2, r_1, p_4

CONCURRENT EVENTS: $p_1 \parallel q_1, \dots$

ORDERED EVENTS (happened before): $p_2 \rightarrow q_2, \dots$

Logical clocks

- System of logical clocks is consistent if
 - if event a happened before event b, then the timestamp of a is smaller than the timestamp of b
for two events a and b , $a \rightarrow b \Rightarrow C(a) < C(b)$
- System of logical clocks is strongly consistent if
 - if event a happened before event b, then the timestamp of a is smaller than the timestamp of b
 - if the timestamp of a is smaller than the timestamp of b then event a happened before event b
for two events a and b , $a \rightarrow b \Leftrightarrow C(a) < C(b)$

Logical clocks

- System of logical clocks is consistent if
 - if event a causally effects event b, then the timestamp of a is smaller than the timestamp of b
for two events a and b , $a \rightarrow b \Rightarrow C(a) < C(b)$
- System of logical clocks is strongly consistent if
 - if event a causally effects event b, then the timestamp of a is smaller than the timestamp of b
 - if the timestamp of a is smaller than the timestamp of b then event a causally effect event b
for two events a and b , $a \rightarrow b \Leftrightarrow C(a) < C(b)$

Logical clocks – scalar time (Lamport)

■ Implementation rules

- each process P_i increments C_i between any two successive events; if a and b are two successive events in P_i and $a \rightarrow b$, then

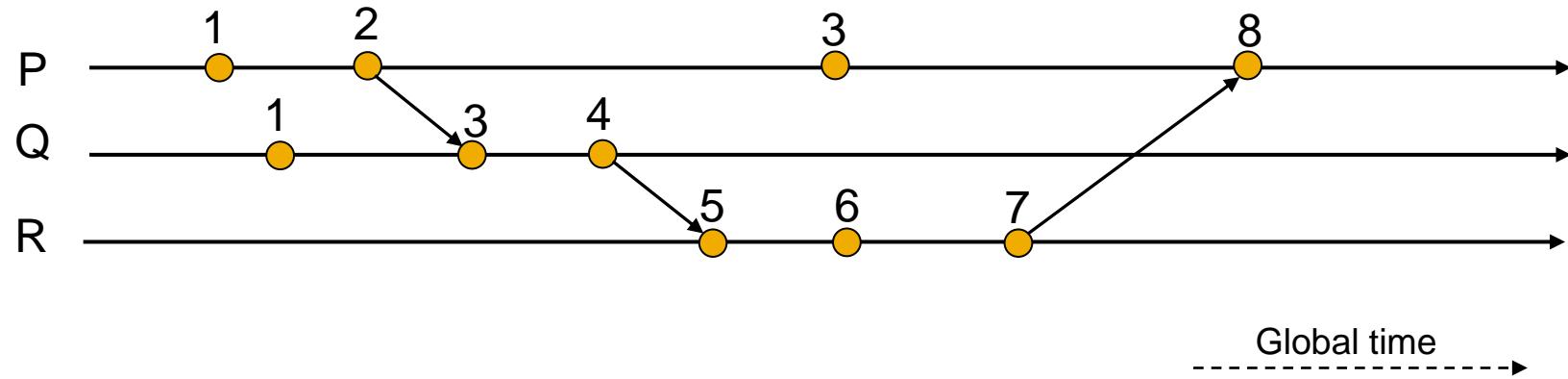
$$C_i(b) = C_i(a) + d, d > 0$$

- (a) a is event of sending of a message m by process P_i to P_j ; the message m contains a timestamp $T_m = C_i(a)$
(b) b is event of receiving of message m by P_j ; upon receiving m by process P_j :

$$C_j = \max(C_j, T_m) + d, d > 0$$

Logical clocks – scalar time

Implementing logical clocks



Logical clocks – scalar time

- Basic properties
 - total order
 - no strong consistency
 - event counting

Logical clocks – scalar time

- Total order
 - scalar time can be used to totally order events in a distributed system, if a tie-breaking mechanism is used
 - What “tie” is?
 - two or more processes within the distributed system can have identical timestamps \Rightarrow
 e_1 and e_2 , $C(e_1) = C(e_2)$ $e_1 \parallel e_2$
 - tie-breaking mechanism is needed to order such events

Logical clocks – scalar time

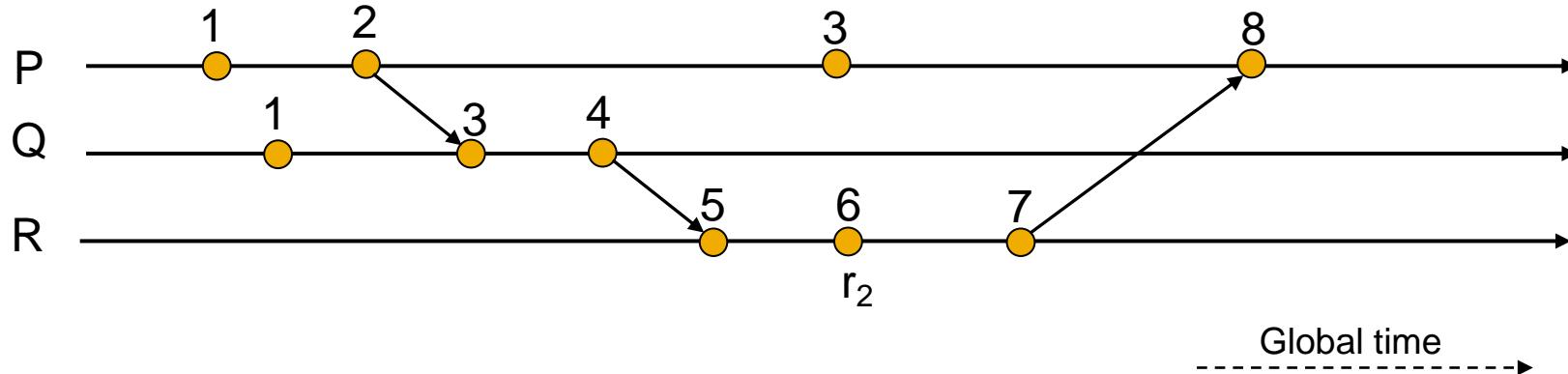
- Total order - tie-breaking mechanism
 - tie is broken based on process identifier
 - events that occur at the same logical time are independent (not related) they can be ordered arbitrary
 - timestamp of an event is denoted by a tuple (t, i)
 - t is its time of occurrence
 - i is the identity of the process where it occurred
- total order relation \prec on two events x and y with timestamps (h,i) and (k,j) , is defined as follows:
$$x \prec y \Leftrightarrow (h < k \text{ or } (h = k \text{ and } i < j))$$

Logical clocks – scalar time

- No strong consistency
 - does not satisfy strong consistency property
for two events e_1 and e_2 , $C(e_1) < C(e_2)$ $e_1 \rightarrow e_2$
 - simply looking at the event clocks one cannot decide if the events are or are not causally related

Logical clocks – scalar time

- Event counting
 - if $d = 1$, e is the event with timestamp h then $h-1$ is the number of events that may causally produced the event e (at different processes)
 - 5 events precede event r_2



Logical clocks – vector time

- Each process keeps a clock which is an integer vector of length n ($n = \text{no. of processes}$)
 - $VC_i[i]$ is the number of events that P_i has timestamped

Local logical clock

- $VC_i[j], j \neq i$ is P_i 's latest knowledge of P_j time
 - $VC_i[j] = x$, P_i knows that local time at P_j has progressed until x

Global logical clock

Logical clocks – vector time

■ Implementation rules

- each process P_i increments $VC_i[i]$ between any two successive events; if a and b are two successive events in P_i and $a \rightarrow b$, then

$$VC_i[i] = VC_i[i] + d, \quad d > 0$$

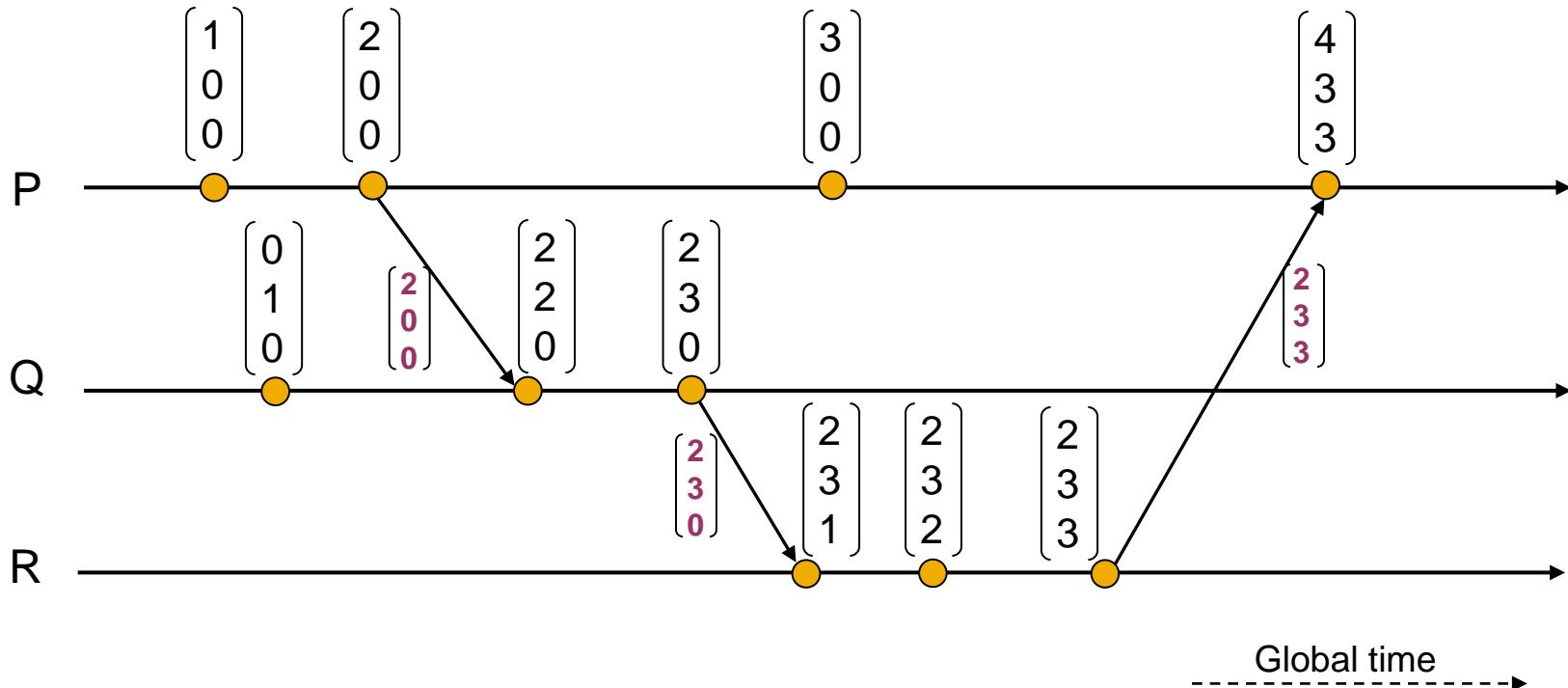
- (a) a is event of sending of a message m by process P_i to P_j ; the message m contains a timestamp $T_m = VC_i(a)$
(b) b is event of receiving of message m by P_j ; upon receiving m by process P_j :

$$VC_j[k] = \max(VC_j[k], T_m[k]), \text{ for } k \neq j$$

$$VC_j[j] = \max(VC_j[j], T_m[j]) + d, \quad d > 0$$

Logical clocks – vector time

- Implementing logical clocks
 - On the receiving of messages, a process learns about the more recent clock values of the rest of the processes in the system



Logical clocks – vector time

- Comparison between two vector clocks

$VC_i = VC_j \Leftrightarrow$ there is $x: VC_i[x] = VC_j[x]$

$VC_i \leq VC_j \Leftrightarrow$ there is $x: VC_i[x] \leq VC_j[x]$

$VC_i < VC_j \Leftrightarrow VC_i \leq VC_j$ and there is $x: VC_i[x] < VC_j[x]$

$VC_i \parallel VC_j \Leftrightarrow \neg(VC_i < VC_j)$ and $\neg(VC_j < VC_i)$

Logical clocks – vector time

- Basic properties
 - strong consistency: by examining the vector timestamp of two events, we can determine if the events are causally related
 - event counting
 - if $d = 1$, i^{th} component of vector clock at process P_i , $VC_i[i]$, denotes the number of events that have occurred at P_i until that instant
 - event e has timestamp VC , then $VC[k]$ denotes the number of events executed by process P_k that causally precede e

Logical clocks – efficient implementation

- Large number of processes leads to large vector clocks (size of vector clock = no of processors)
 - messages sent within distributed system are piggybacked with large vector clocks
- Straightforward implementation of vector clocks is inefficient
- How to efficiently implement vector clocks?
 - reduce vector clock size?
 - send only a part of the vector clock, not all? What part?

Logical clocks – efficient implementation

- For large system, between successive messages sent from P_i to P_j only a few entries of the vector clock VC_i change
 - Rarely: all entries of VC_i have been modified since last message, so the whole VC_i has to be sent
 - Often: only few entries of VC_i have been modified since last message, so these entries have to be sent
- When P_i sends a message to P_j , it sends in the timestamp T_m only the entries of VC_i that differ since the last message sent to P_j
 - differential technique
- When P_j receives the timestamp it updates its VC_j according to the second rule
- Disadvantage – each process has to keep the timestamp of the last message sent to each process
 - $O(n^2)$ storage overhead at each process

Singhal-Kshemkalyani

- How to reduce the storage overhead to $O(n)$?
- Each process P_i , apart from VC_i keeps two more vectors each of size n
 - LS (Last Sent)
 - LU (Last Update)
- How to use Last Send and Last Update to figure out those entries of VC_i which differ since last message sent to P_j ?

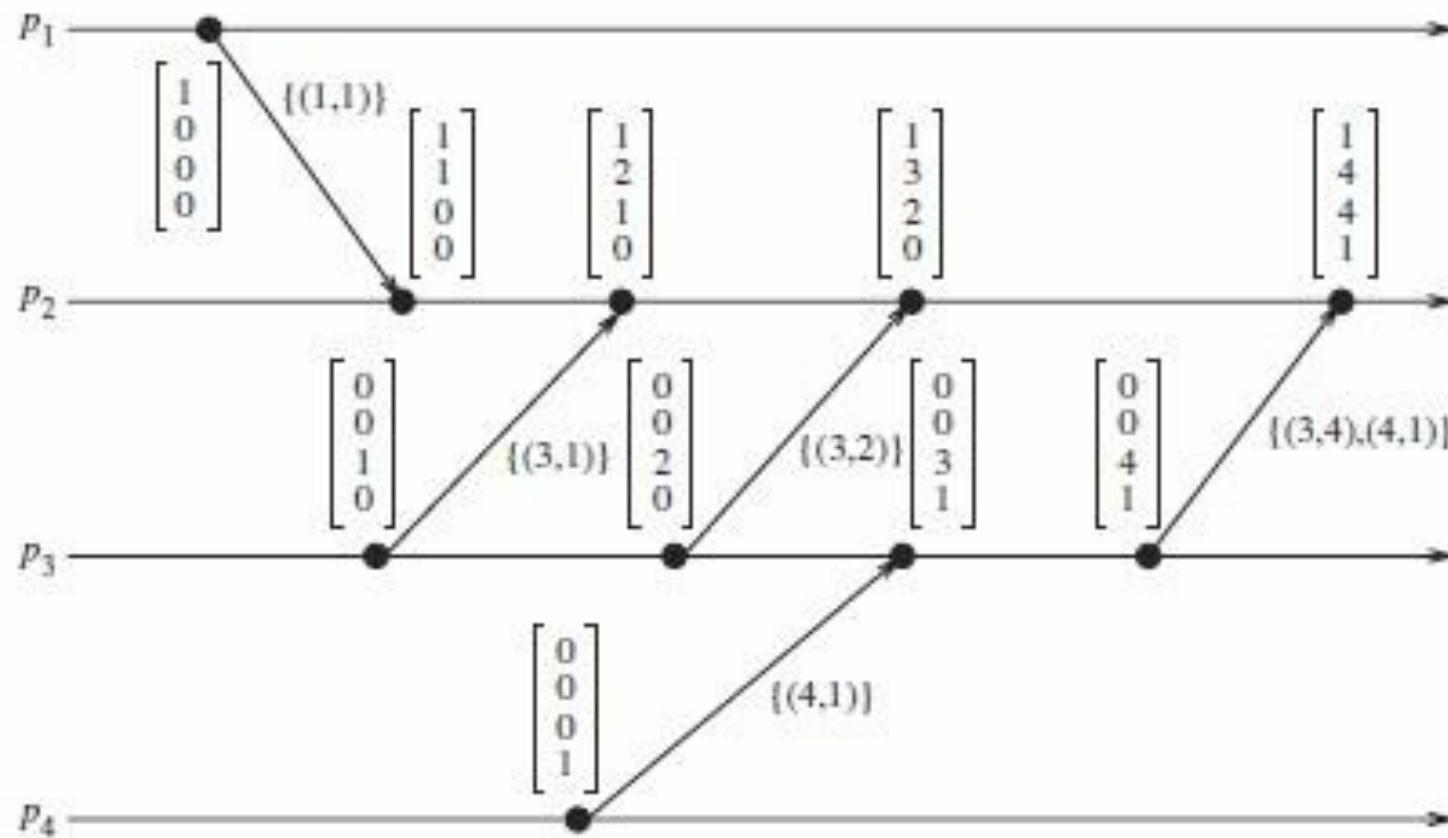
Singhal-Kshemkalyani

$LS_i[j]$ = value of $VC_i[i]$ when process P_i last sent a message to P_j
 $LS_i[j]$ needs to be updated only when P_i sends a message to P_j

$LU_i[j]$ = value of $VC_i[i]$ when process P_i last updated the entry $VC_i[j]$
 $LU_i[j]$ needs to be updated only when the receipt of a message causes P_i to update $VC_i[j]$

If $LS_i[j] < LU_i[k]$ then $VC_i[k]$ has changed since last message sent to P_j , therefore $VC_i[k]$ will be part of the timestamp

Singhal-Kshemkalyani



Causal ordering

Contents

1. Problem definition
2. Why causal order is important?
3. Causal ordering protocols based on vector clock
 - 3.1. Birman-Schiper-Stephenson
 - 3.2. Schiper-Eggli-Sandoz

System model

- Distributed system consisting of a collection of N processes P_i ,
 $i=1, 2, \dots, N$
- Each process executes on a single processor
- Processes communicates only by message passing
- Each process P_i has a state s_i
- Each process executes a sequence of events
- There are three types of events: send, receive and operation that transforms the state
- There is no global time

Problem definition

- Often in a distributed system it is necessary to be able to answer the question:
 - has event e_1 at process P_1 been responsible for causing event e_2 at process P_2 ?
- Having the answer one can establish the relationships between events, i.e. the order in which the events take place
- Causal order denotes a partial ordering defined over a set of events
- Causal order is based on happened before relation

Logical clocks

- Scalar clocks
 - $a \rightarrow b$ implies $C(a) < C(b)$
 - $C(a) < C(b)$ not necessarily implies $a \rightarrow b$, i.e. simply looking at the event clocks one cannot decide if the events may be or may not be causally related
- Vector clocks
 - $a \rightarrow b$ implies $VC(a) < VC(b)$
 - $VC(a) < VC(b)$ implies $a \rightarrow b$, i.e. simply looking at the event clocks one can decide if the events may be or may not be causally related

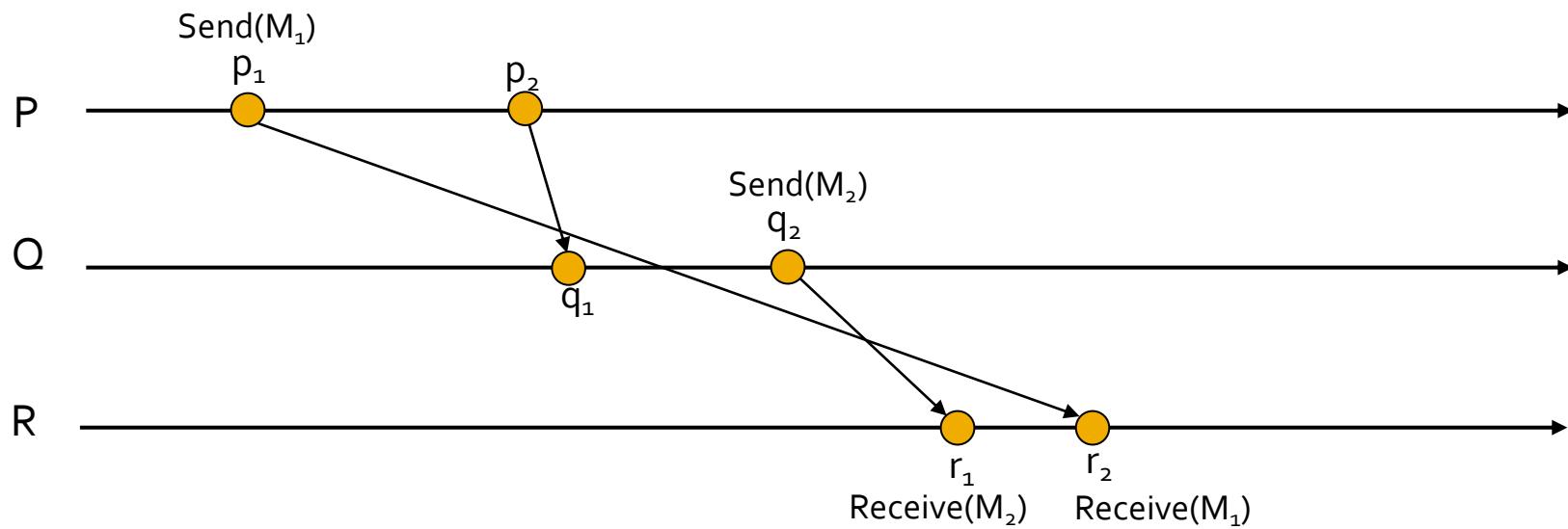
Causal ordering of events and messages

- So far we have discussed about causal ordering of events, i.e. specifying which of two events occurred first
- Causal ordering of messages implies that the causal relationship among "message send" events correspond to "message receive" events

Causal ordering of events and messages

- Message delivery is said to be causal if the order in which messages are received is consistent with the order in which they are sent.
- If $\text{Send}(M_1) \rightarrow \text{Send}(M_2)$, then $\text{Receive}(M_1) \rightarrow \text{Receive}(M_2)$ if M_1 and M_2 are received by the same process, i.e. all messages are processed in the order that they were created
- A message is dependent on other messages that were sent before it
- The message can be delivered only when no causality constraints are violated, otherwise, the message is not delivered immediately but is buffered until all the preceding messages are delivered

Violation of causal ordering of messages



Why causal order is important?

- Maintaining consistency in replicated databases
- Monitoring a distributed system
- Resource allocation

Causal ordering protocols

- Causal ordering protocol based on causal history
- Causal ordering protocols based on clocks
 - physical clock (common clock or perfectly synchronized clock)
 - vector clocks

Causal ordering protocol based on vector clocks

Idea: A process receives a message sent, delays it if necessary, and then delivers it in an order according to causality. Therefore a message is delivered only if the messages preceding it have been delivered.
Otherwise the message is buffered.

Algorithms:

Birman-Schiper-Stephenson
Schiper-Eggli-Sandoz

Birman-Schiper-Stephenson algorithm

- The set of processes communicate using broadcast
- P_1, P_2, \dots, P_n processes
- t^m is the vectorstamp for message m
- each process P_i has:
 - a (kind of) vector clock VC_i

Birman-Schiper-Stephenson algorithm

- C is not exactly a vector clock as the updating procedure does not entirely follow the one for the vector clocks.
- There are three basic principles to this algorithm:
 - All messages are time stamped by the sending process.
[Note: This time is separate from the global time talked about in the previous section. Instead, each element of the vector corresponds to the number of messages sent (including this one) to other processes.]
 - A message can not be delivered until:
 - All the messages before this one have been delivered locally.
 - All the other messages that have been sent out from the original process has been accounted as delivered at the receiving process.
 - When a message is delivered, the clock is updated.

Birman-Schiper-Stephenson algorithm

P_i broadcasts a message m

- P_i increments $VC_i[i]$ and sets the timestamp $t^m = VC_i$
- $VC_i[i] - 1$ indicates how many messages from P_i precede m

Birman-Schiper-Stephenson algorithm

P_j ($j \neq i$) receives a message m with timestamp t^m from P_i ; it delays the delivery of the message until:

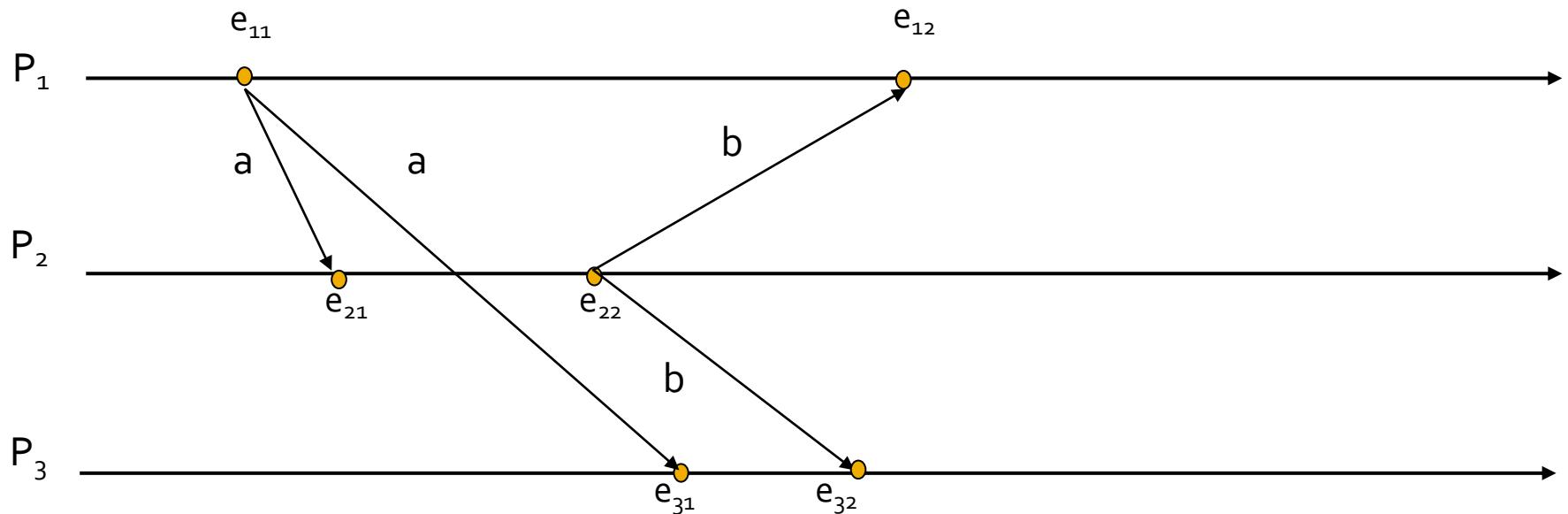
1. $VC_j[i] = t^m[i] - 1$ (this step checks that P_j has received all the messages from P_i that precede m)
2. for all $k \leq n$, $k \neq i$, $VC_j[k] \geq t^m[k]$ (this step checks that P_j has received all those messages received by P_i before sending m)

Birman-Schiper-Stephenson algorithm

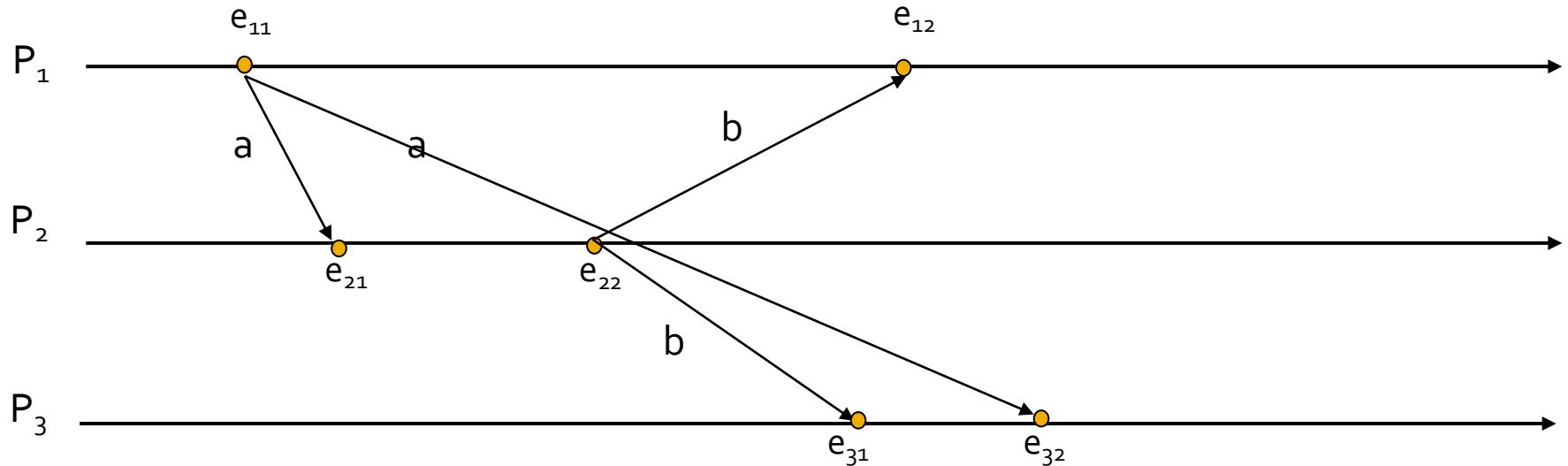
When the message is delivered to P_j

- P_j updates its vector clock as follows:
$$VC_j[k] = \max(VC_j[k], t^m[k])$$
- checks buffered messages to see if any can be delivered

Birman-Schiper-Stephenson algorithm



Birman-Schiper-Stephenson algorithm



Schiper-Eggle-Sandoz algorithm

- does not require broadcast
- P_1, P_2, \dots, P_n processes
- each process P_i has
 - a vector clock VC_i
 - a vector of tuple V_i having length of $n-1$
 - tuple (j, t^m) where j is the destination process and t^m the vector timestamp of the last message sent by P_i to P_j

Schiper-Eggle-Sandoz algorithm

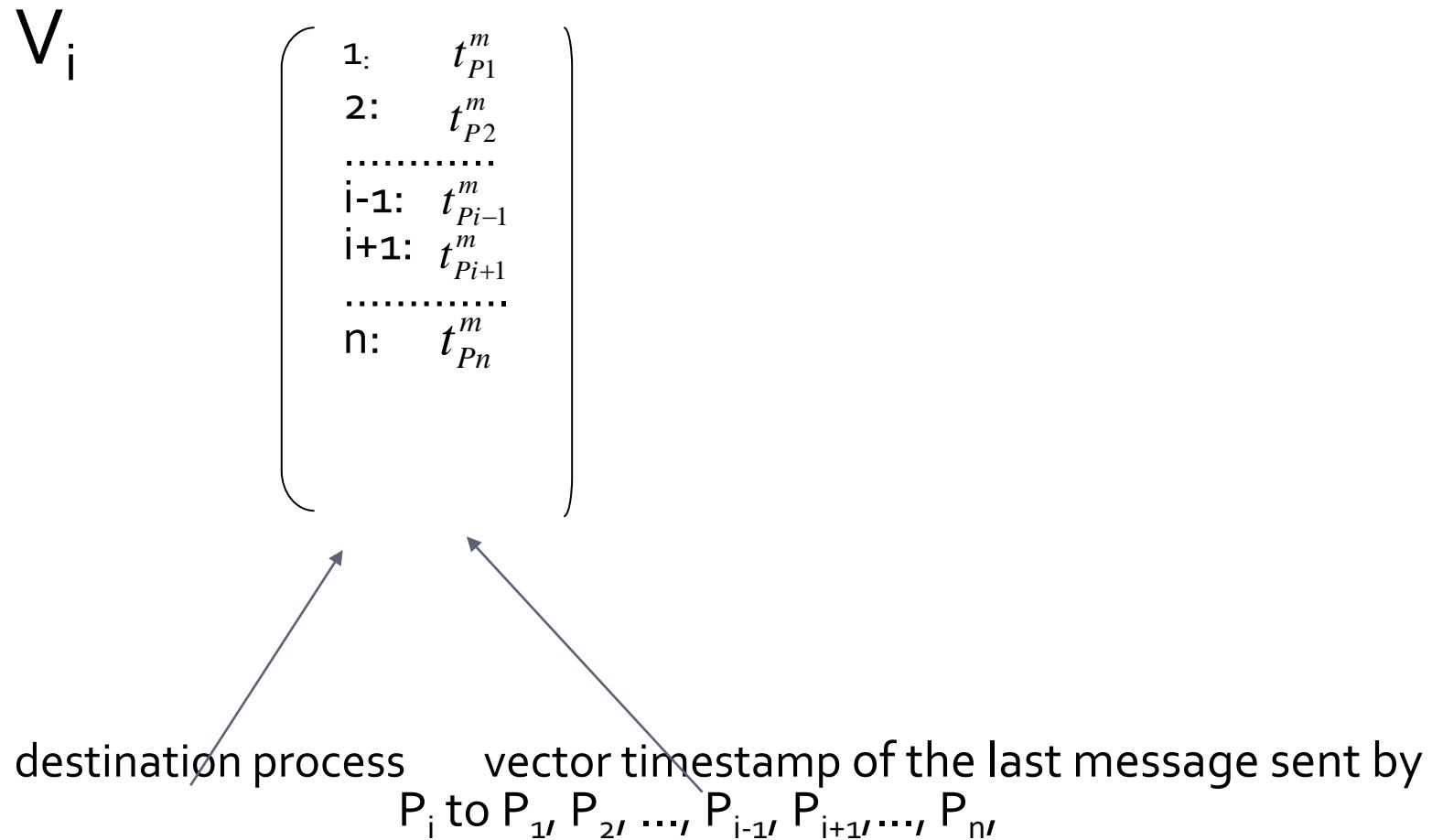
- VC is a vector clock.
- Instead of maintaining a vector clock based on the number of messages sent to each process, the vector clock for this protocol can increment at any rate it would like to and has no additional meaning related to the number of messages currently outstanding.
- **Sending a message:**
 - All messages are timestamped and sent out with a list of all the timestamps of messages sent to other processes.
 - Locally store the timestamp that the message was sent with.

Schiper-Eggle-Sandoz algorithm

■ Receiving a message:

- A message cannot be delivered if there is a message mentioned in the list of timestamps that precedes this one.
- Otherwise, a message can be delivered, performing the following steps:
 - 1. Merge in the list of timestamps from the message:
 - - Add knowledge of messages destined for other processes to our list of processes if we didn't know about any other messages destined for one already.
 - - If the new list has a timestamp greater than one we already had stored, update our timestamp to match.
 - 2. Update the local logical clock.
 - 3. Check all the local buffered messages to see if they can now be delivered.

Schiper-Eggle-Sandoz algorithm



Schiper-Eggle-Sandoz algorithm

P_i send a message to P_j

1. P_i increments $VC_i[i]$ and set $t_{Pj}^m = VC_i$
2. P_i sends m timestamped t^m and V_i to P_j

3. $V_i[j] = t_{Pj}^m$

Pair (P_j, t_{Pj}^m) was not sent to P_j

Schiper-Eggle-Sandoz algorithm

P_j ($j \neq i$) receives a message m with timestamp t^m and V_i from P_i :

$V_i[j]$ is not set \rightarrow message can be delivered

$V_i[j]$ is set and $V_i[j] \leq VC_j \rightarrow$ message can be delivered

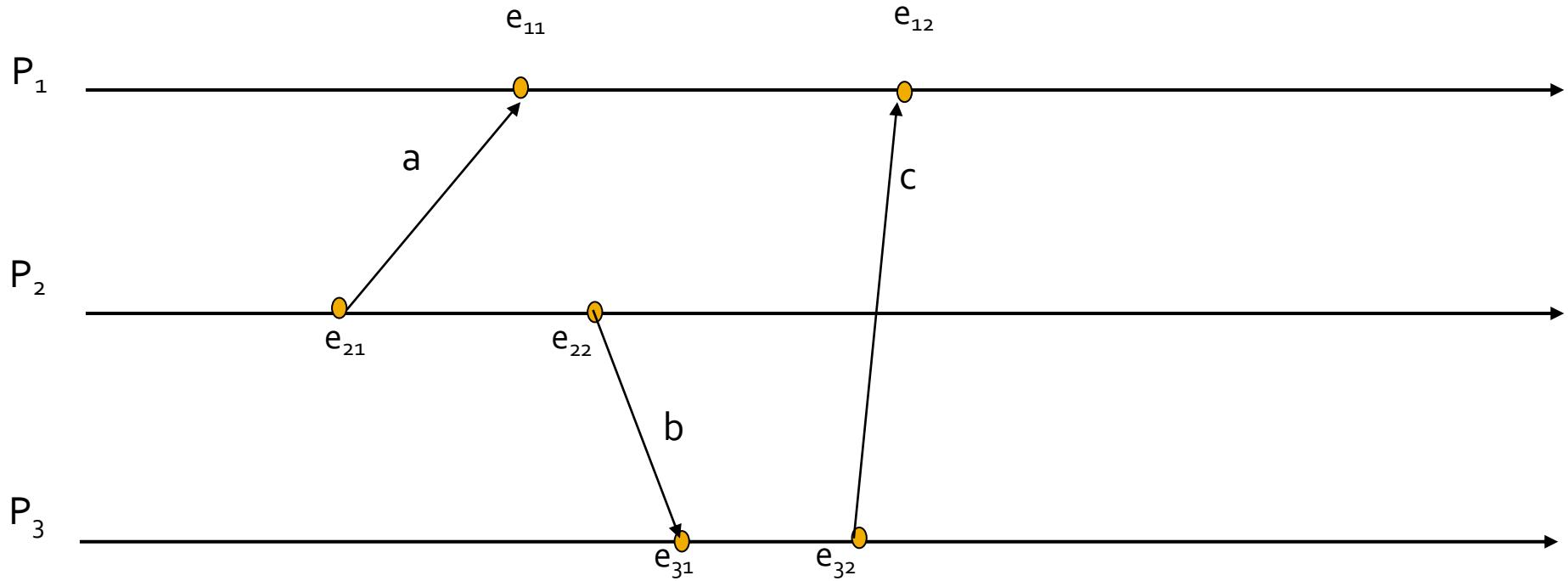
$V_i[j]$ is set and $V_i[j] > VC_j \rightarrow$ message cannot be delivered (some events occurred at other processes, but P_j does not know about)

Schiper-Eggle-Sandoz algorithm

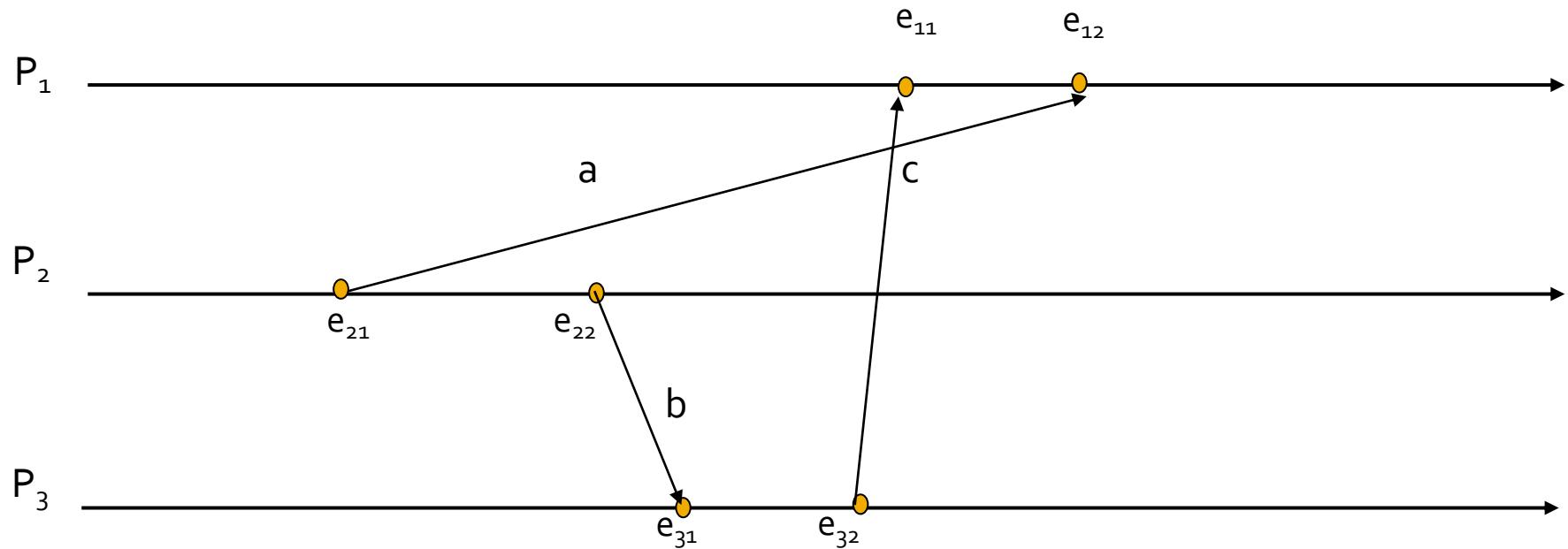
When the message is delivered to P_j

- P_j updates the elements of V_j with the corresponding elements of V_i , except for $V_j[j]$, as follows
 - if $V_j[k]$ is not initialized then $V_j[k] = V_i[k]$
 - if $V_j[k]$ is initialized then $V_j[k] = \max(V_j[k], V_i[k])$
- P_j updates its vector clock (using t_{Pj}^m)
- checks buffered messages to see if any can be delivered

Schiper-Eggli-Sandoz algorithm



Schiper-Eggli-Sandoz algorithm (Homework)



Distributed mutual exclusion

Contents

1. Problem definition

2. Why distributed mutual exclusion is important?

3. Mutual exclusion algorithms

3.1. Token-ring

3.2. Suzuki-Kasami

3.3. Central coordinator

3.4. Lamport

3.5. Ricart-Agrawala

Problem definition

- N processes share a resource that can be accessed by one process at a certain time
- The process accessing the critical resource is said to be in the critical section (CS)
- Conditions to be satisfied:
 - Safety: only one process can access the shared resource at a time
 - Liveness: if a process requests to access the shared resource it should eventually be given a chance to do so

Why distributed mutual exclusion is important?

- Basically, the same motivation as for non-distributed mutual exclusion
- Exclusive access of any shared resources
 - N users want to print to a shared printer

System model

- Distributed system consisting of a collection of N processes P_i ,
 $i=1, 2, \dots, N$
- Each process executes on a single processor
- Processes communicates only by message passing
- Each process P_i has a state s_i
- Each process executes a sequence of events
- There are three types of events: send, receive and operation that transforms the state
- There is no global time

Distributed mutual exclusion algorithms

- Used in an environment where the communication is done by message passing
- Token-based algorithms
 - Token-ring
 - Suzuki-Kasami
- Non-token based algorithms
 - Central coordinator
 - Lamport
 - Ricart-Agrawala

Token based mutual exclusion algorithms

Idea:

- a unique token moves among the processors;
- a processor can enter the CS only if it has the token

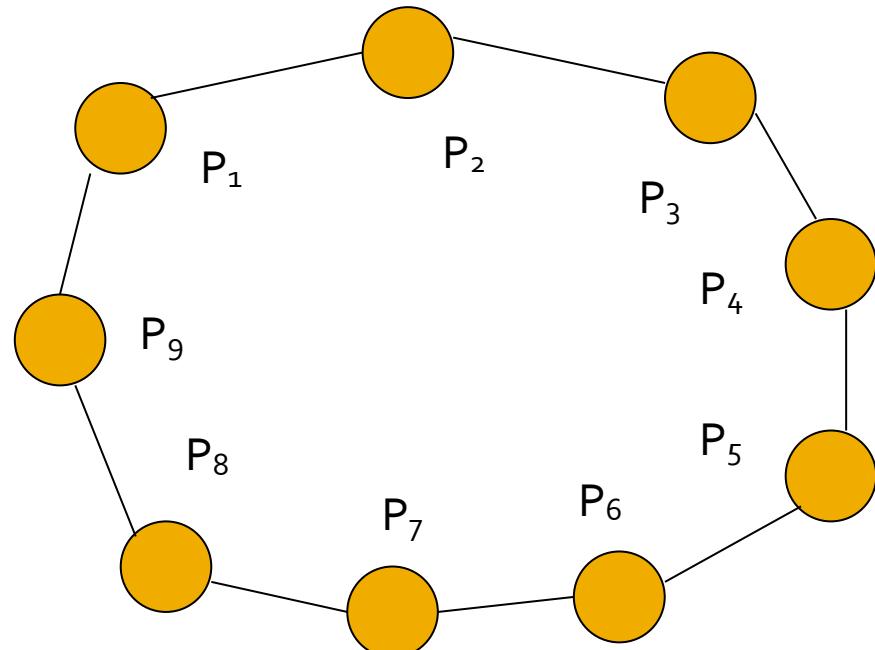
Algorithms:

Token ring

Suzuki-Kasami

Token ring algorithm

- Processors are arranged in a logical ring
- First processor P_1 gets token



Token ring algorithm

- When process P_i gets the token, it checks if it wants to enter CS
 - If it wants to enter CS, simply it enters CS
 - If it does not want to enter CS, it sends the token to $P_{(i+1) \bmod N}$
- Properties
 - Simple algorithm
 - Suitable for small N
 - Problem with missing token

Suzuki-Kasami's algorithm

- P_1, P_2, \dots, P_n processors
- Each processor P_i holds a timestamp vector T
 - $T_i[j]$ is the timestamp of the most current request from P_j known by P_i
- Token holds a vector X
 - $X[i]$ is the timestamp of the last request from P_i that has been served
- One can easily find if a processor has requests that have not been yet served by comparing T and X

Suzuki-Kasami's algorithm

- P_i requests CS
 - P_i increase $T_i[i]$ by 1 and broadcasts *request* ($T_i[i]$, i) to all processors
 - When P_j receives the *request* ($T_i[i]$, i)
 - Update $T_j[i]$ to $\max(T_j[i], T_i[i])$
 - If it does not have the token does nothing
 - If it has the token and does not need it, then it releases the token

Suzuki-Kasami's algorithm

- **P_j releases the CS**
 - P_j sets $X[j] = T_j[j]$
 - Check T_j starting with $T_j[j+1]$
 - If for some k : $T_j[k] > X[k]$ then P_k has a waiting request, send the token to P_k
- **P_i enters the CS**
 - If P_i has the token

Suzuki-Kasami's algorithm

- Requires N messages per request
- Problem with missing token

Non-token based mutual exclusion algorithms

Idea:

- When a processor wants to enter the critical section, it sends a timestamped message to the other processors.
- In case there are many nodes that try to enter the critical section the one with the lowest timestamp wins.

Algorithms:

Central coordinator (it does not obey the above idea)

Lamport

Ricart-Agrawala

Central coordinator algorithm

- P_1, P_2, \dots, P_n processors,
- C central coordinator – grants access to CS
- **P_i requests CS**
 - P_i sends a *request* to C and waits for a reply
- **P_i enters the CS**
 - when it gets the *reply* from coordinator
- **P_i releases the CS**
 - notifies the coordinator with a *release* message

Central coordinator algorithm

- Properties
 - simple
 - only three messages per CS use
 - Requests are granted in order in which they are received
- Problems
 - Coordinator – performance bottleneck
 - Coordinator – point of failure

Lamport's algorithm

- P_1, P_2, \dots, P_n processors
- Each processor has a queue where mutual exclusion requests are kept, the requests are ordered by their timestamps
- Each processor maintains a Lamport logical clock
- Messages are delivered in FIFO order between every pair of processors

Lamport's algorithm

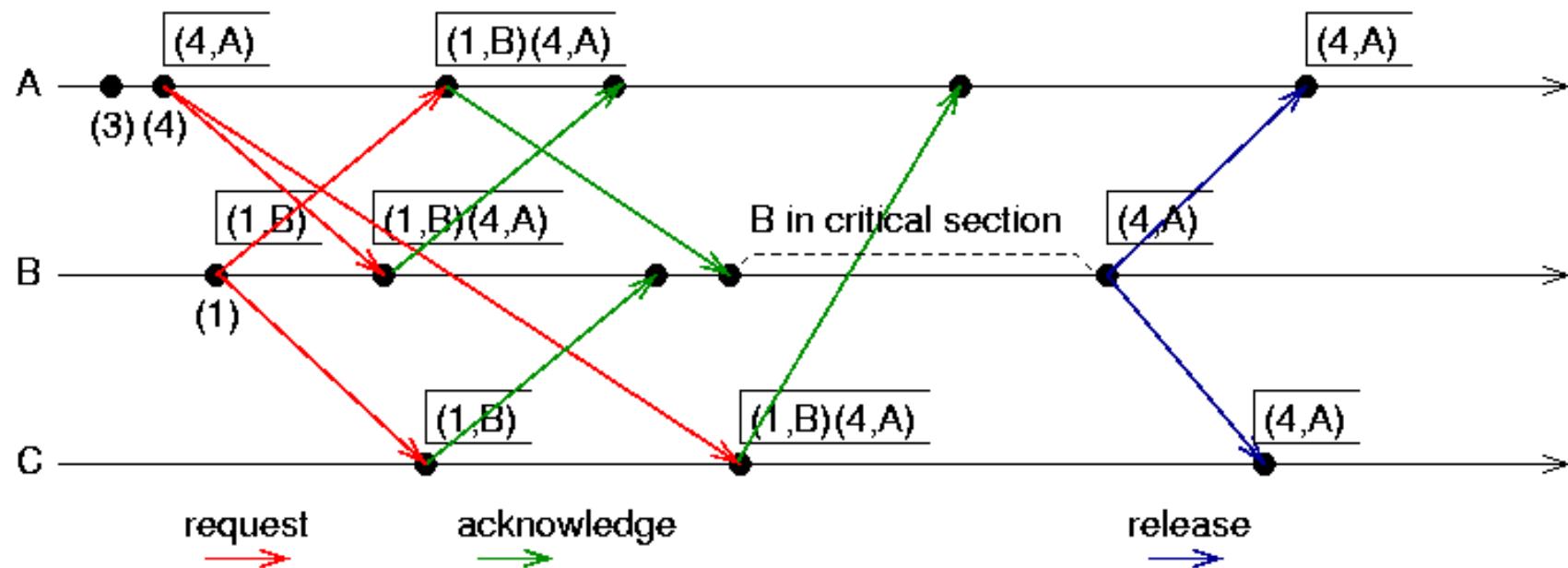
■ P_i requests CS

- P_i broadcasts the request $request(i, LC_i)$ to all processors
- P_i puts the request in its own queue (in the order of the timestamps of the requests)
- When P_j receives a request, it returns a timestamped reply $reply(j, LC_j)$ to P_i
- P_j puts the request in its own queue

Lamport's algorithm

- P_i enters the CS
 - If P_i has received (reply) messages from all processors with timestamps larger than its own and its request is at the top of its queue

Lamport's algorithm



Lamport's algorithm

- P_i releases the CS
 - Upon exiting CS, P_i removes its request from the request queue and sends a timestamped release $release(LC_i)$ message to all processors
 - When P_j receives a release message it deletes the request from its queue (this way its own request can get to the top of the queue)

Lamport's algorithm

- The algorithm requires:
 - Total ordering of events
 - All processors to be alive
- Performance
 - $3(N-1)$ messages per request

Ricart-Agrawala's algorithm

- P_1, P_2, \dots, P_n processors
- No request queue
- Each processor maintains a Lamport logical clock
- Does not need FIFO channels
- Optimization of Lamport's algorithm as the *release* messages are merged with the *reply* messages

Ricart-Agrawala's algorithm

- P_i requests CS

- P_i broadcast the request $request(i, LC_i)$ to all processors
- When P_j receives a request
 - If it has not requested or it is not executing a CS, then it returns a timestamped reply $reply(j, LC_j)$ to P_i
 - If it has requested a CS but $LC_j > LC_i$, then it returns a timestamped reply $reply(j, LC_j)$ to P_i
 - Otherwise, postpones the reply

Ricart-Agrawala's algorithm

- **P_i enters the CS**
 - If P_i has received reply messages from all processors
- **P_i releases the CS**
 - Upon exiting CS, P_i sends *reply(i, LC_i)* to all the postponed requests

Ricart-Agrawala's algorithm

- The algorithm requires:
 - Total ordering of events
 - All processors to be alive
- Performance
 - $2(N-1)$ messages per request

Leader election

Contents

1. Problem definition
2. Why leader election is important?
3. Leader election algorithms
 - 3.1. In general networks
 - FloodMax, OptFloodMax
 - 3.2. In synchronous / asynchronous ring
 - LeLann, Chang-Roberts, Hirschberg-Sinclair, Franklin
(leader - max identity)
 - Peterson (leader – arbitrary process)
 - 3.3. In anonymous ring
 - Itai-Rodeh

Problem definition

- Elect a unique leader in a network
- All processors have the same local algorithms
- Each computation terminates with one process elected as a leader

Why leader election is important?

- When a leader is selected many other properties of network can be computed:
 - count the number of vertices
 - assign unique identifiers to vertices
 - build a tree
- Possible to execute centralized protocols in decentralized environments
- Recover from token-loss for token-based protocols

Model of distributed system

- **Synchronous distributed system:**
 - lower and upper bounds on the execution time of processes, messages are received within a known bounded time, drift rates between local clocks have a known bound, global physical time (with a certain precision)
 - predictable behavior
- **Asynchronous distributed system:**
 - no lower and upper bounds on the execution time of processes, messages are not received within a known bounded time, drift rates between local clocks do not have a known bound no global physical time (logical time is needed)
 - unpredictable behavior

Leader election algorithms

- communication mechanism
 - asynchronous
 - synchronous
- process name
 - unique identifier
 - anonymous
- network topology
 - ring
 - tree
 - graph
- number of processors
 - known by the algorithm
 - not known

FloodMax
OptFloodMax



strongly connected
graph
processes know
diameter =
*the shortest distance
between the two most
distant nodes in
the network*

FloodMax algorithm

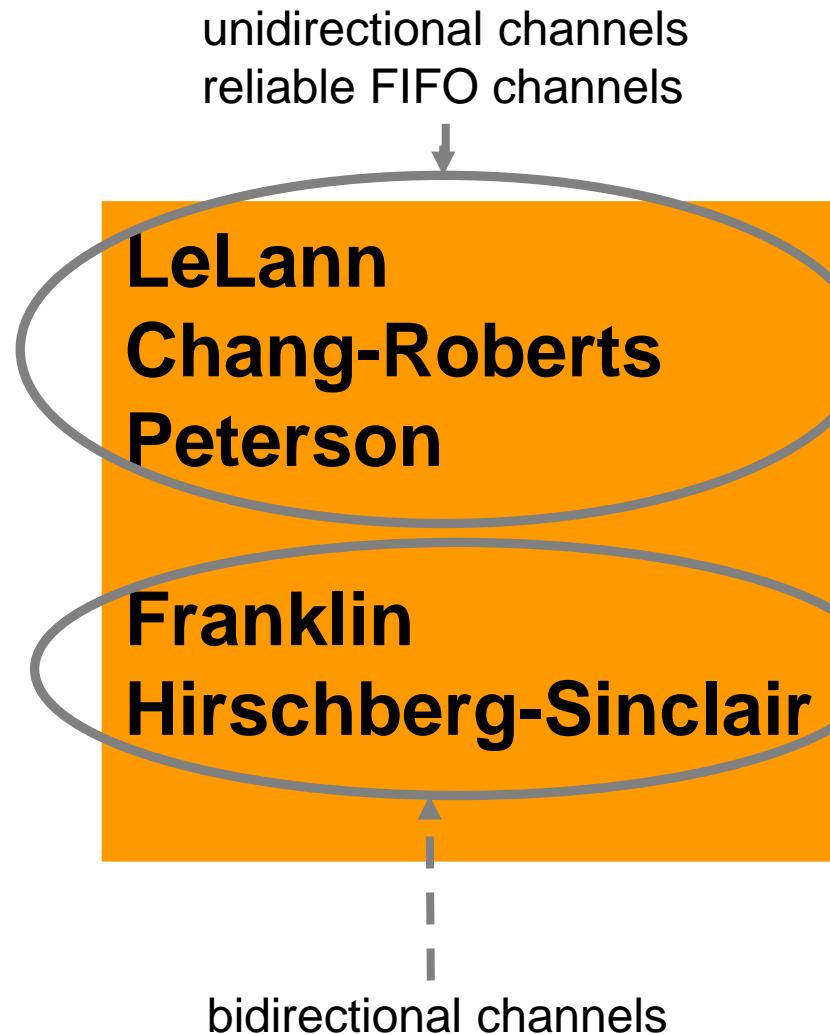
- process maintains a record of the maximum identifier seen so far
- at each round, each process propagates this maximum to all of its outgoing edges
- after $diam$ rounds, if the maximum value seen is the process's own identifier, the process elects itself leader, otherwise it is a non-leader
- time complexity: $diam$
- communication complexity : $diam * E$ (E is the number of directed edges in the graph)

OptFloodMax algorithm

- reduced communication complexity
- processes can send their max-uid only when they first learn about them, not at every round
- to reduce the number of messages even more, if a process i receives a new message from a process j that is both an incoming and outgoing neighbor then i need not send a message in the direction of j at the next round

Leader election algorithms

- **communication mechanism**
 - asynchronous
 - synchronous
- **process name**
 - unique identifier
 - anonymous
- **network topology**
 - ring
 - tree
 - graph
- **number of processors**
 - known by the algorithm
 - not known



LeLann algorithm

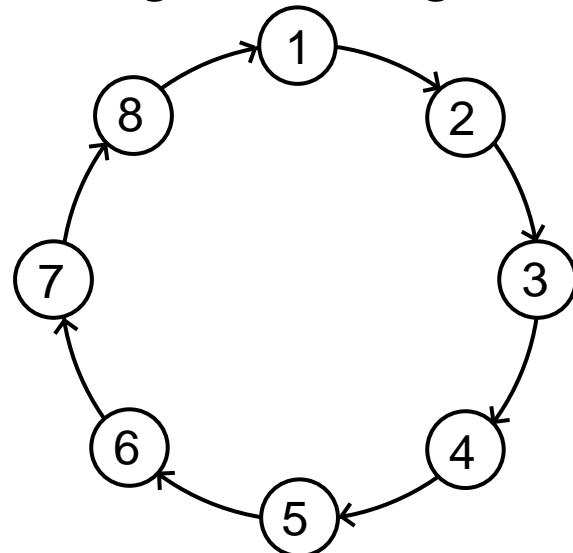
- initially every process is in *unknown* state
- every process sends a message with its own id to its neighbor
- when a process P_i gets a message:
 - $u_i < u_j$: P_i gets the state *non-leader* and pass on u_j
 - $u_i > u_j$: P_i pass on u_j
 - $u_i = u_j$: if the state of P_i is *unknown* then P_i is leader (if the state of P_i is *non-leader* then nothing)
- time complexity: n
- communication complexity: always requires n^2 messages

Chang-Roberts algorithm

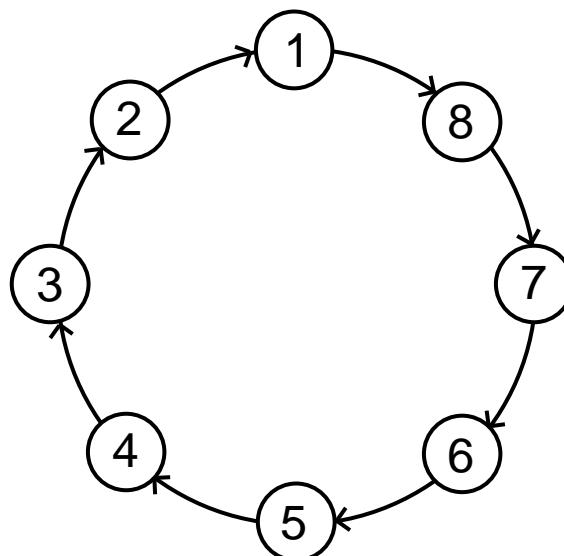
- initially every process is in *unknown* state
- every process sends a message with its own id to its neighbor
- when a process P_i gets a message:
 - $u_i < u_j$: P_i gets the state *non-leader* and pass on u_j
 - $u_i > u_j$: the message with u_j is discarded
 - $u_i = u_j$: P_i is leader

Chang-Roberts algorithm

- time complexity: $2n$
- communication complexity:
 - $2n-1$ messages in the best case
 - $n(n+1)/2$ messages in the worst case
 - $n\log_2(n)$ messages in average case



Best case scenario: $O(n)$



Worst case scenario: $O(n^2)$

Comparison

LeLann

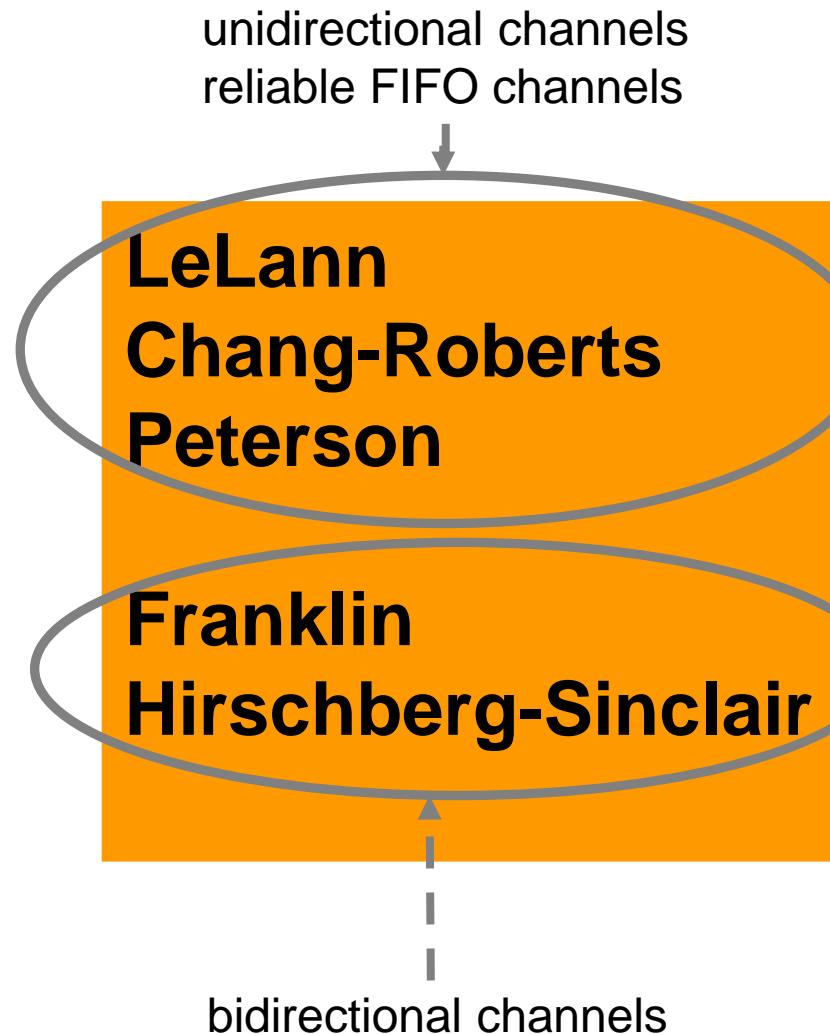
- when a process P_i gets a message:
 - $u_i < u_j$: P_i gets the state *non-leader* and pass on u_j
 - $u_i > u_j$: P_i pass on u_j
 - $u_i = u_j$: if the state of P_i is *unknown* than P_i is leader
- nodes terminate

Chang – Roberts

- when a process P_i gets a message:
 - $u_i < u_j$: P_i gets the state *non-leader* and pass on u_j
 - $u_i > u_j$: the message with u_j is discarded
 - $u_i = u_j$: P_i is leader
- only the leader terminates

Leader election algorithms

- **communication mechanism**
 - asynchronous
 - synchronous
- **process name**
 - unique identifier
 - anonymous
- **network topology**
 - ring
 - tree
 - graph
- **number of processors**
 - known by the algorithm
 - not known

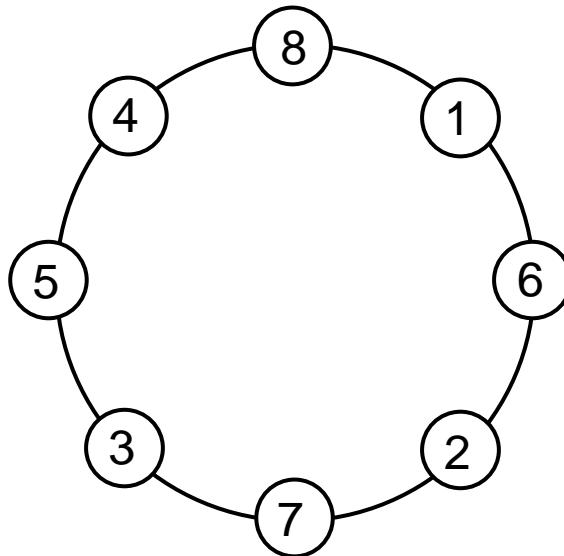


Franklin algorithm

- initially each process is in *active* mode
- every process sends a message with its own id to its neighbors
- when a process P_i gets messages from P_{i-1} and P_{i+1} :
 - if $u_i < u_{i+1}$ or $u_i < u_{i-1}$, P_i becomes passive
 - passive nodes pass on messages
- terminates when a node gets its own identifier
 - only the node with max id get back its own identifier
 - leader announces the other processes
- worst-case message complexity **O(nlog₂n)**

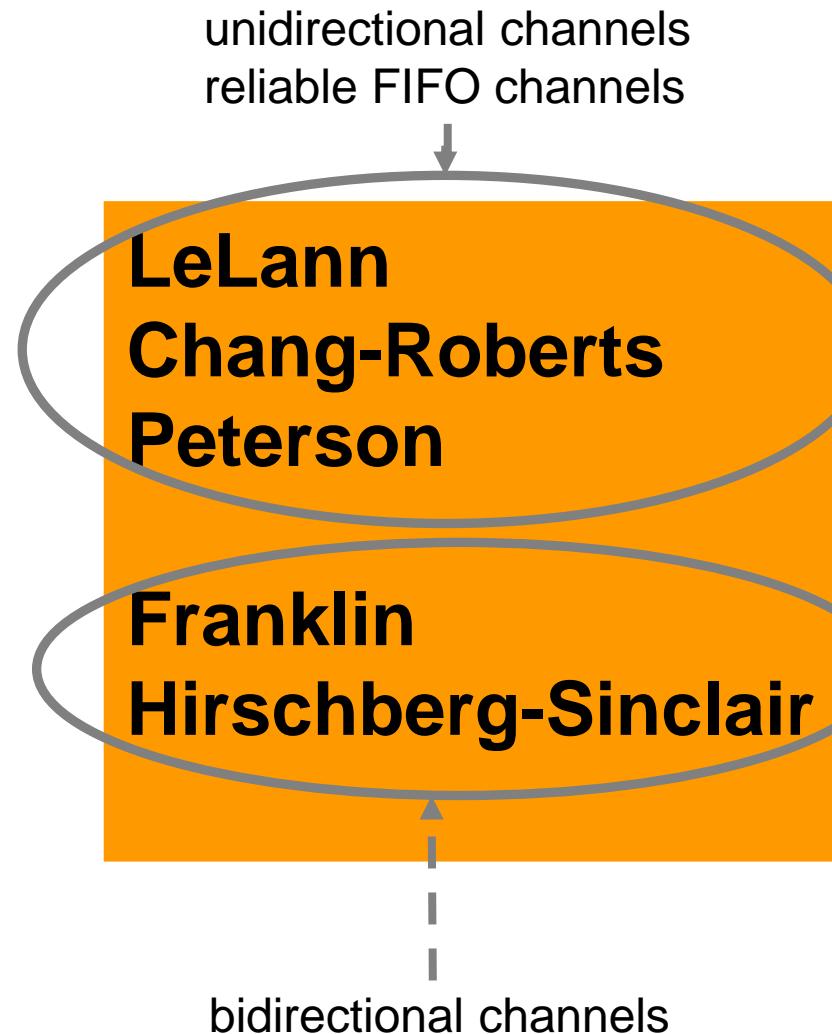
Franklin algorithm

- worst-case message complexity $O(n \log_2 n)$



Leader election algorithms

- **communication mechanism**
 - asynchronous
 - synchronous
- **process name**
 - unique identifier
 - anonymous
- **network topology**
 - ring
 - tree
 - graph
- **number of processors**
 - known by the algorithm
 - not known



Peterson algorithm

- initially each process is in *active* mode
- **phase:** each process P_i sends its identifier to the next and second-next processes in the clockwise direction
 - waits to learn the temporary identifiers from its 2 active predecessors
 - at process P_i if $u_{i-1} > u_{i-2}$ and $u_{i-1} > u_i$, then process P_i remains active, adopting the identifier u_{i-1} of its counterclockwise neighbor as a new “temporary identifier”
 - P_i will be involved in the next phase
 - if the condition doesn't hold, process P_i becomes passive for the remainder of the execution

Peterson algorithm

- algorithm terminates
 - at any phase when process P_i gets an identifier with the same value as its own temporary identifier
 - P_i selected as leader
 - P_i announces the other processes about the leader selection

Peterson algorithm

- algorithm execution is divided into phases
 - in each phase, the number of active processes is reduced by a factor of at least 2, so at most $\log_2 n$ phases exist
- communication complexity
 - during each phase, each process (*active* or *relay*) sends 2 messages
 - at most $\lceil \log n \rceil + 1$ phases exist
 - hence at most $2n(\lceil \log n \rceil + 1)$ messages are sent in every execution of the algorithm → **O(n log₂ n)**

Hirschberg-Sinclair algorithm

- initially each process is in active mode
- every process sends a message with its own id to 2^k distance away $k = 0, 1, \dots$
 - phase k : P_i sends a message containing its identifier u_i in both directions which travels the distance 2^k
 - if both messages return safely, then P_i continues with the next phase
- when P_j receives a message:
 - $u_i < u_j$: P_j discards the message
 - $u_i > u_j$: P_j relays the message
 - $u_i = u_j$: P_j elects itself as a leader
 - algorithm terminates for the selected leader
 - leader announces the other processes
- worst-case message complexity to **O($n \log_2 n$)**

Leader election algorithms

- **communication mechanism**
 - asynchronous
 - synchronous
- **process name**
 - unique identifier
 - anonymous
- **network topology**
 - ring
 - tree
 - graph
- **number of processors**
 - known by the algorithm
 - not known

process with the largest identity becomes the leader

LeLann
Chang-Roberts
Hirschberg-Sinclair
Franklin

Peterson

an arbitrary process is elected as leader
not necessarily the process with the
maximum or minimum identifier

Leader election algorithms

Ring Types	Authors	Algorithm Complexity		
		Best Case	Worst Case	Average Case
Unidirectional Ring	LeLann	$O(n^2)$	$O(n^2)$	$O(n^2)$
	Chang-Roberts	$O(n)$	$O(n^2)$	$O(n \log_2 n)$
	Peterson		$O(n \log_2 n)$	
Bidirectional Ring	Franklin		$O(n \log_2 n)$	
	Hirschberg-Sinclair		$O(n \log_2 n)$	

Leader election in anonymous networks

- anonymous network: nodes do not have unique identifiers → nodes are indistinguishable
- deterministic algorithms – impossible for leader election
- probabilistic algorithms for leader election
 - a node picks random identity
 - deterministic algorithm is used to work with pseudo identities
 - deterministic algorithm has to be able to work with duplicate identities

Leader election algorithms

- **communication mechanism**
 - asynchronous
 - synchronous
- **process name**
 - unique identifier
 - **anonymous**
- **network topology**
 - **ring**
 - tree
 - graph
- **number of processors**
 - **known by the algorithm**
 - not known

Itai-Rodeh

Itai-Rodeh algorithm

- probabilistic algorithm
- each process maintains a data structure containing:
 - 1) its identity
 - 2) state $\in \{\text{active}, \text{passive}, \text{leader}\}$
 - 3) round number
- at the beginning of an election round the active processes send messages containing
 - 1) id and round from the process
 - 2) hop
 - 3) bit

Itai-Rodeh algorithm

- initially: each p_i randomly selects its identity id_i and sends the message $(\text{id}_i, 1, 1, \text{true})$
- a passive process p_i receives a message $(\text{id}, \text{round}, \text{hop}, \text{bit})$
 - $\text{hop}++$
 - pass on the message
- an active process p_i receives a message $(\text{id}, \text{round}, \text{hop}, \text{bit})$
 - 5-step algorithm

Itai-Rodeh algorithm

1. if $\text{hop}=n$ and $\text{bit}=\text{true}$ then the leader is p_i
2. if $\text{hop}=n$ and $\text{bit}=\text{false}$ then p_i takes a new identity and increments round (i.e. moves to the next round) and sends the message $(id_i^{\text{new}}, \text{round}_i + 1, 1, \text{true})$
3. if $(\text{round}, \text{id})=(\text{round}_i, \text{id}_i)$ and $\text{hop}< n$ then p_i passes on the message $(\text{id}, \text{round}, \text{hop}+1, \text{false})$
4. if $(\text{round}, \text{id})>(\text{round}_i, \text{id}_i)$, the process p_i becomes passive and passes on the message $(\text{id}, \text{round}, \text{hop}+1, \text{bit})$
5. if $(\text{round}, \text{id})<(\text{round}_i, \text{id}_i)$ the p_i discards the message

Itai-Rodeh algorithm

- round numbers are essential if the channels are not FIFO
- if the channels are FIFO, round numbers are redundant:
 - if an active process receives a message, then the round number of the message is equal to the round number of the process

Itai-Rodeh algorithm

- Leader election without round numbers
 - 1. if $\text{hop} = n$ and $\text{bit} = \text{true}$ then the leader is p_i
 - 2. if $\text{hop} = n$ and $\text{bit} = \text{false}$ then p_i takes a new identity and increments round (i.e. moves to the next round) and sends the message $(id_i^{\text{new}}, 1, \text{true})$
 - 3. if $\text{id} = id_i$ and $\text{hop} < n$ then p_i passes on the message $(id, \text{hop}+1, \text{false})$
 - 4. if $\text{id} > id_i$, the process p_i becomes passive and passes on the message $(id, \text{hop}+1, \text{bit})$
 - 5. if $\text{id} < id_i$ the p_i discards the message

Itai-Rodeh algorithm

- Leader election without bits
 - When an active process p_i detects a name clash, meaning that it receives a message with its own identity and hop counter smaller than n , it is not necessary for p_i to wait for its own message to return. Instead p_i can immediately select a new random identity and send a new message. This algorithm is obtained by the previous algorithm. All occurrences of bits are omitted.
 - 1. if $\text{hop} = n$ then the leader is p_i
 - 2. if $\text{id} = \text{id}_i$ and $\text{hop} < n$ then p_i selects a new random identity and send the message $(\text{id}_i^{\text{new}}, 1)$
 - 3. if $\text{id} > \text{id}_i$, the process p_i becomes passive and passes on the message $(\text{id}, \text{hop}+1)$
 - 4. if $\text{id} < \text{id}_i$ the p_i discards the message

Quiz

- How does Chang-Roberts improve LeLann algorithm?
- What are FIFO channels?



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Towards Smart Network Devices: FPGAs in the Network

17/03/2023 – UTCN Lecture

Zsolt István

Systems Group at TU Darmstadt

The Systems Group at TU Darmstadt

Hello World! The Systems Group at TU Darmstadt is a collaboration of professors from the Computer Science department, pursuing research and teaching in systems topics together.

Core Topics in the Systems Group



Data and AI Systems

Conducting research in several areas of databases and data management in conjunction with artificial intelligence (AI), focusing on Systems for AI and AI for Systems

[→ Learn more](#)



Distributed and Networked Systems

Working in the intersection of distributed systems, specialized hardware and data management, with the goal of making data-intensive systems more efficient

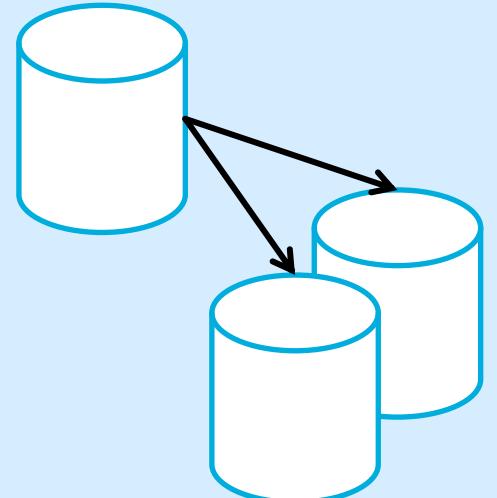
[→ Learn more](#)

Outline

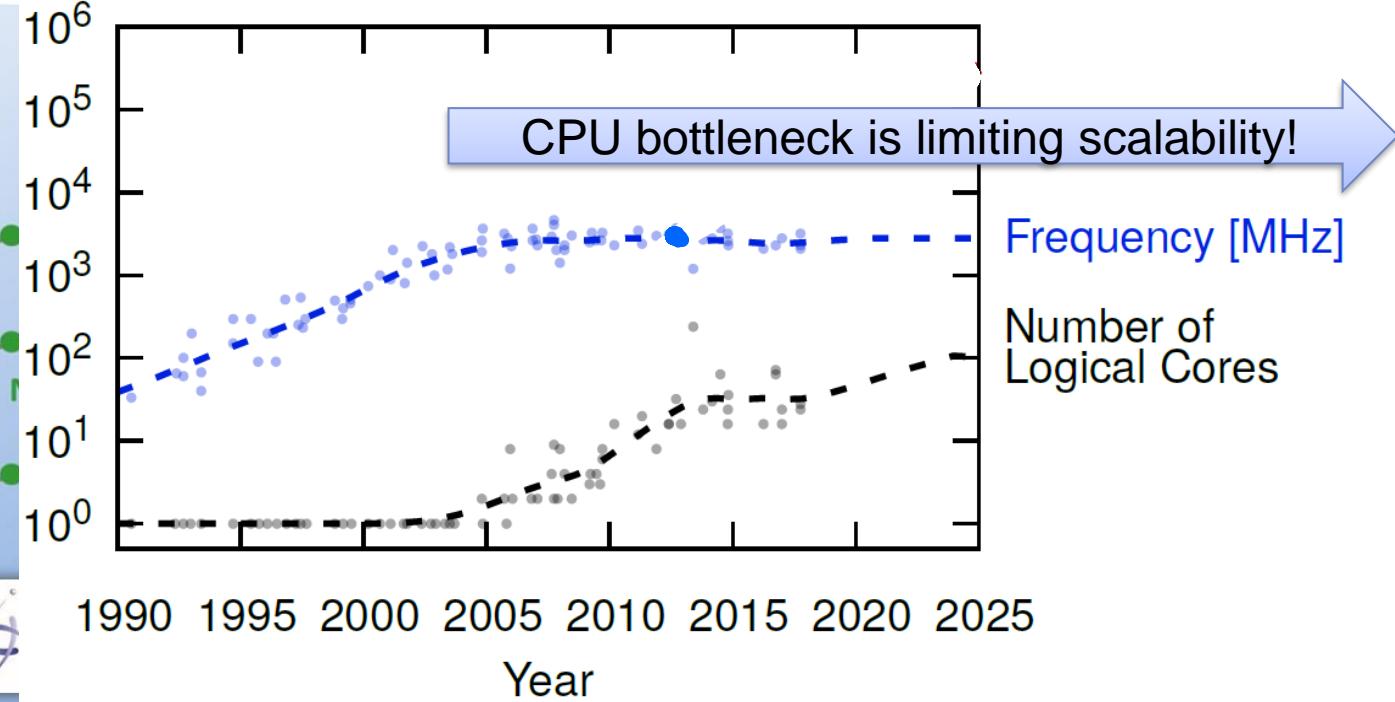
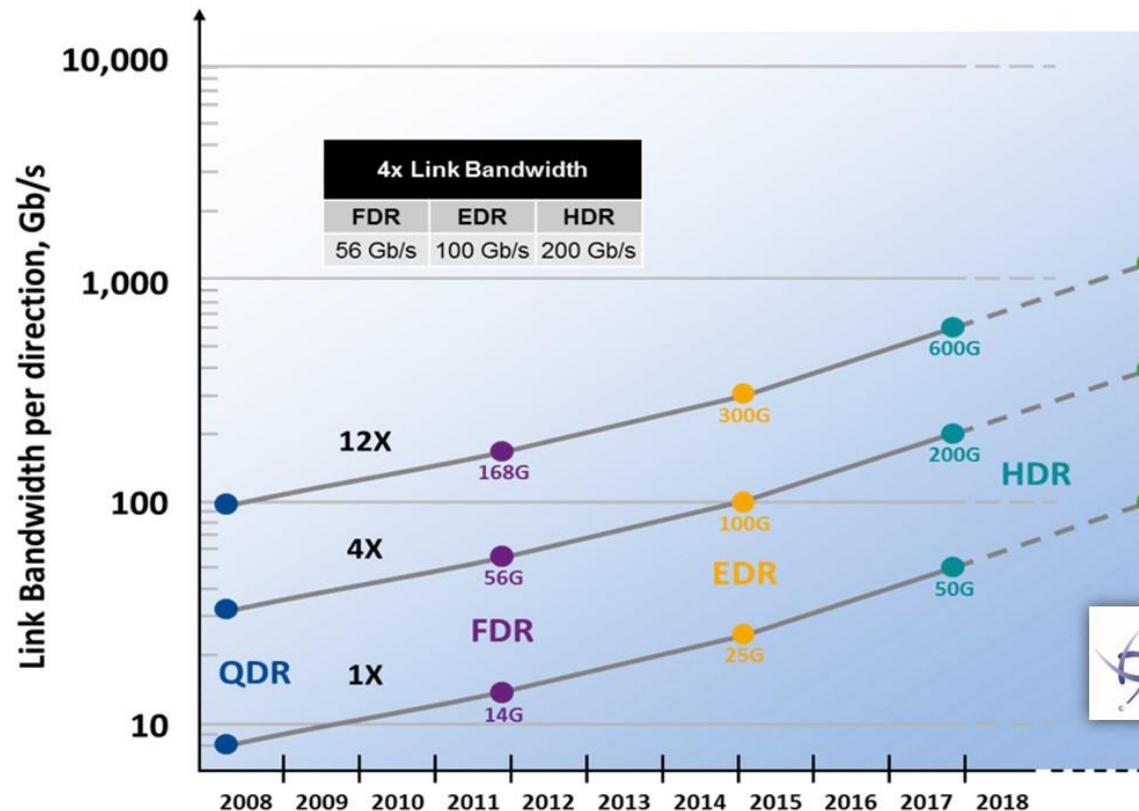
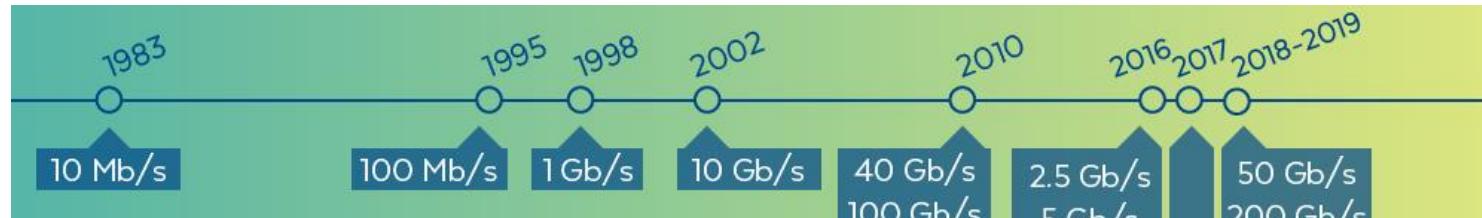
- Motivation for Specialized Networking
- Specialization – Concepts and Devices
- The Coordination Bottleneck

Network bottlenecks in datacenter and cloud

- Distributed systems are everywhere (Spark, ML, databases...)
- Data movement bottlenecks
 - Get data across at HIGH BANDWIDTH
- Coordination bottlenecks
 - Get to agreement with LOW LATENCY
- Typically both need to be solved for scalability



Faster and faster links should solve all issues, no?

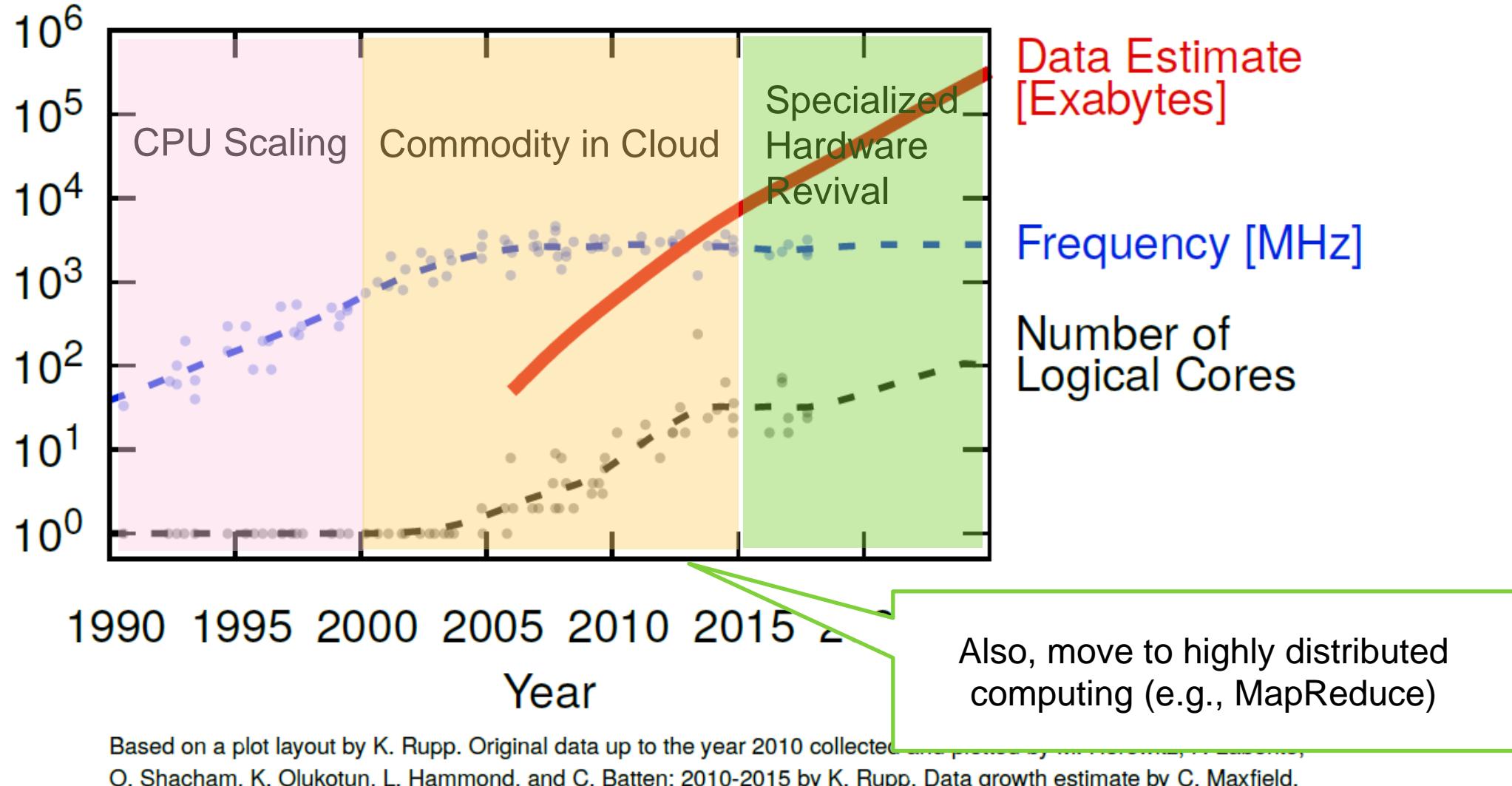


Based on a plot layout by K. Rupp. Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten; 2010-2015 by K. Rupp. Data growth estimate by C. Maxfield.

Outline

- Motivation for Specialized Networking
- Specialization – Concepts and Devices
- The Coordination Bottleneck

Specialization – An economic view

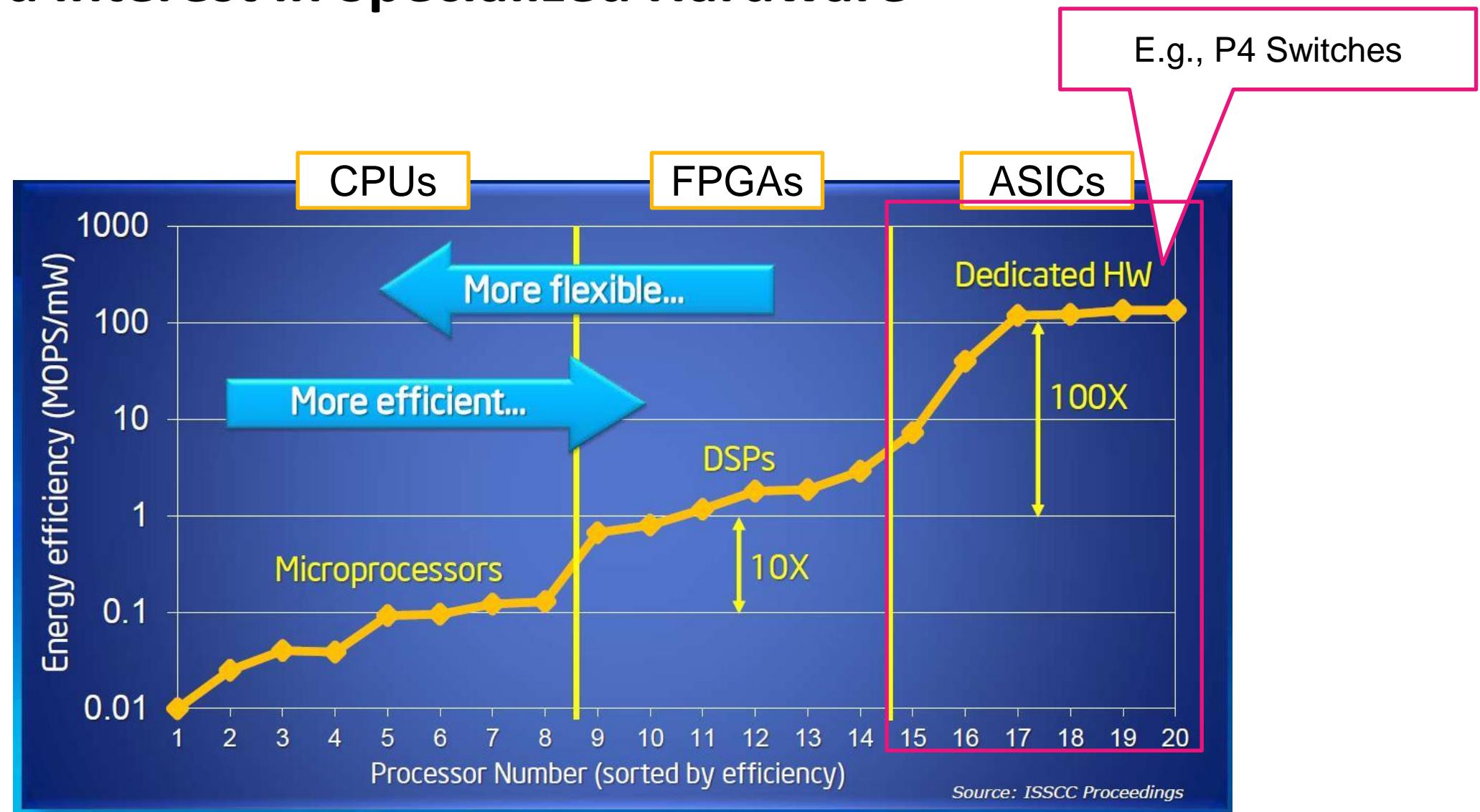


Specialized Hardware is Everywhere

- Accelerators (GPGPUs, TPUs, FPGAs, ...)
- Fast and Smart Networking (NICs, Switches,
- Smart Drives, ZNS, ...
- Motherboards with spec. chips
- ...

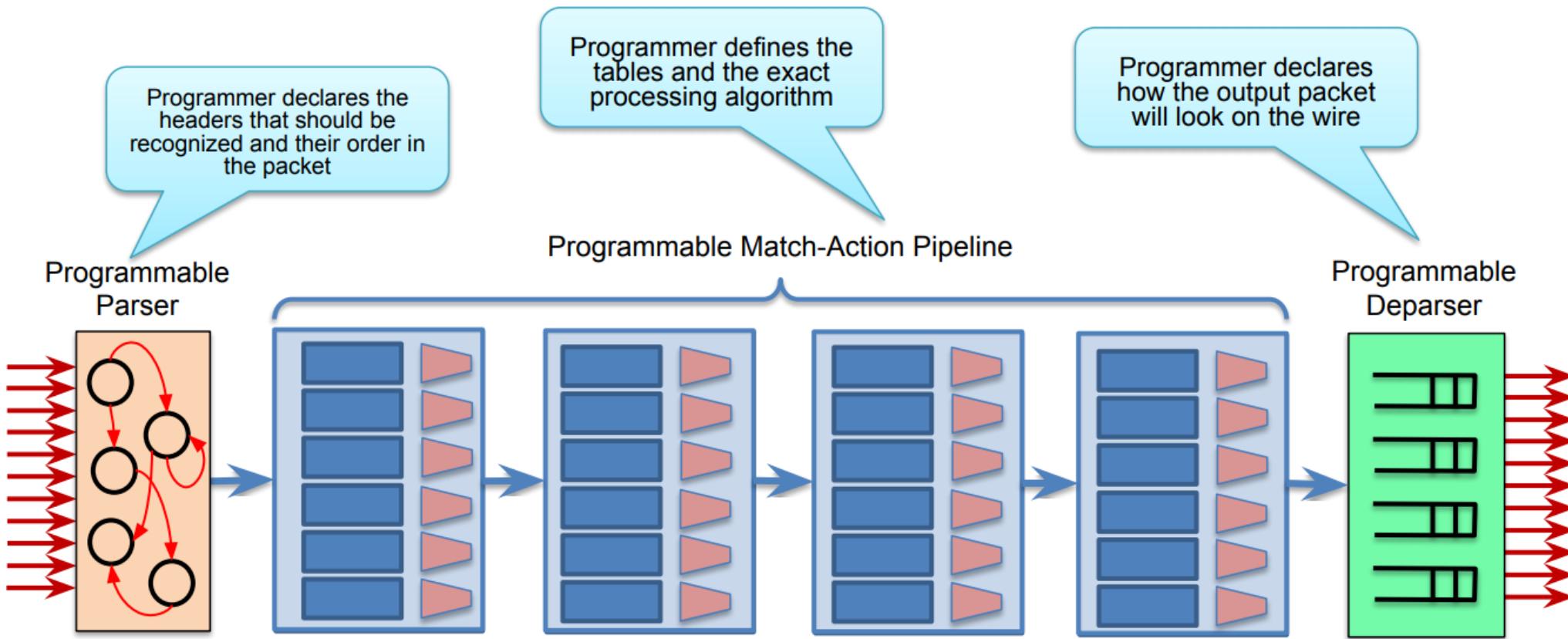


Renewed interest in Specialized Hardware

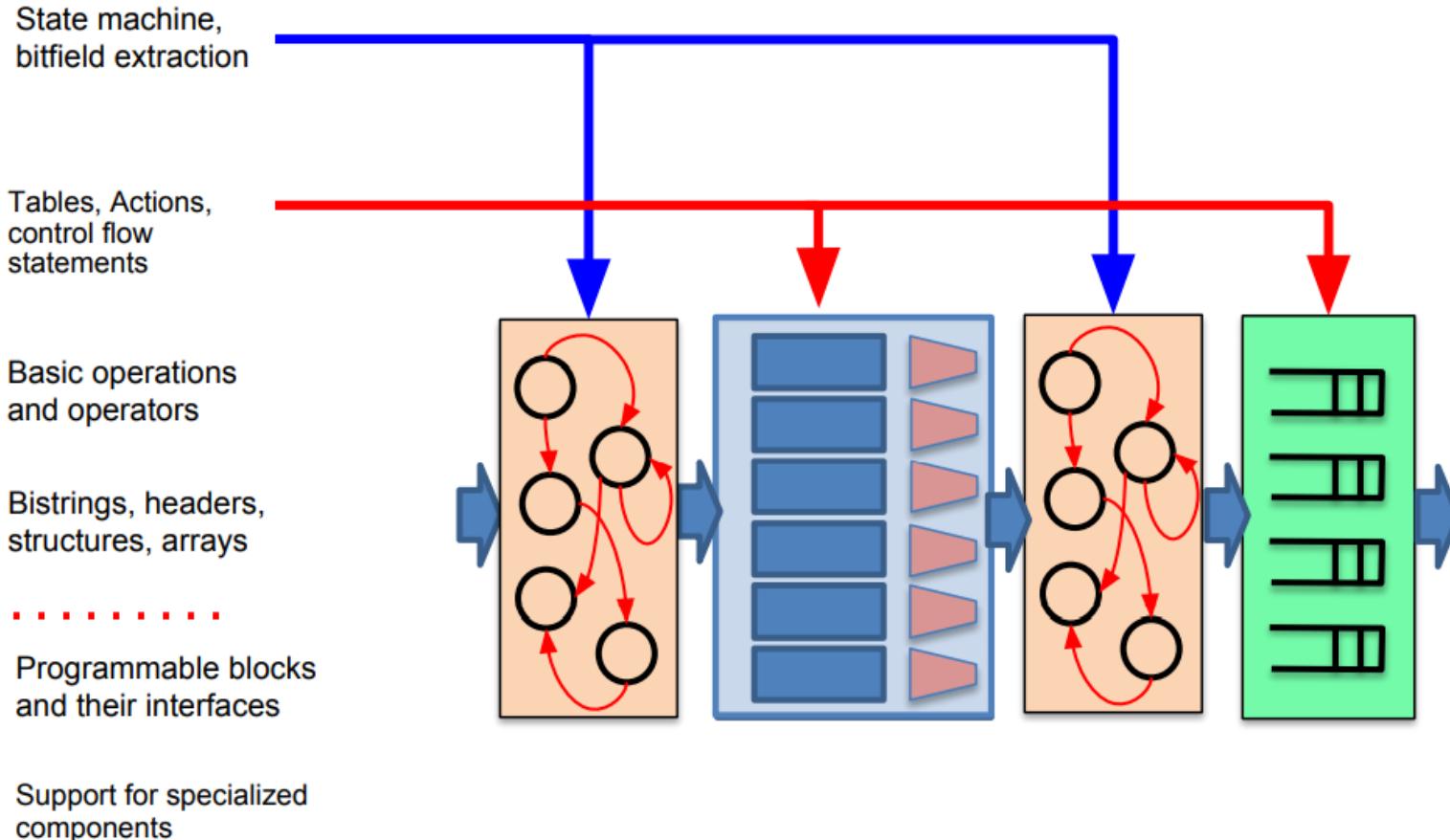
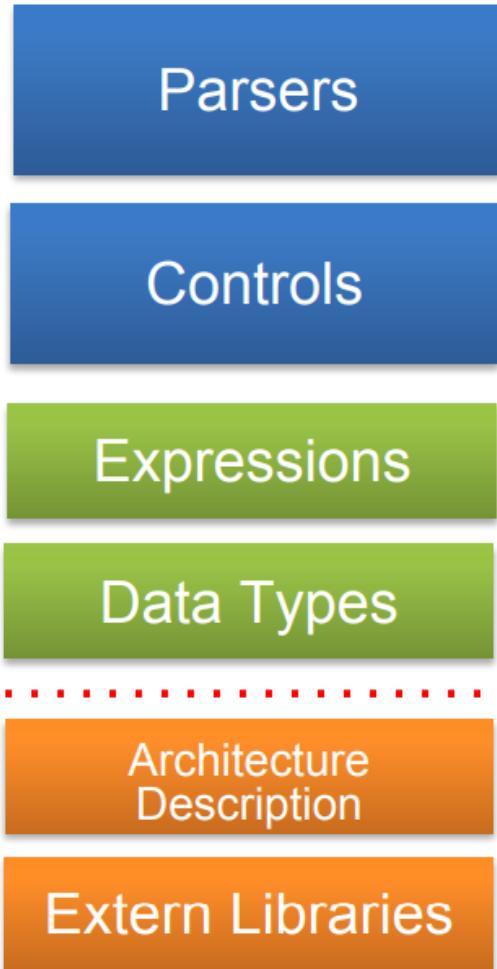


Protocol Independent Switch Architecture (PISA)

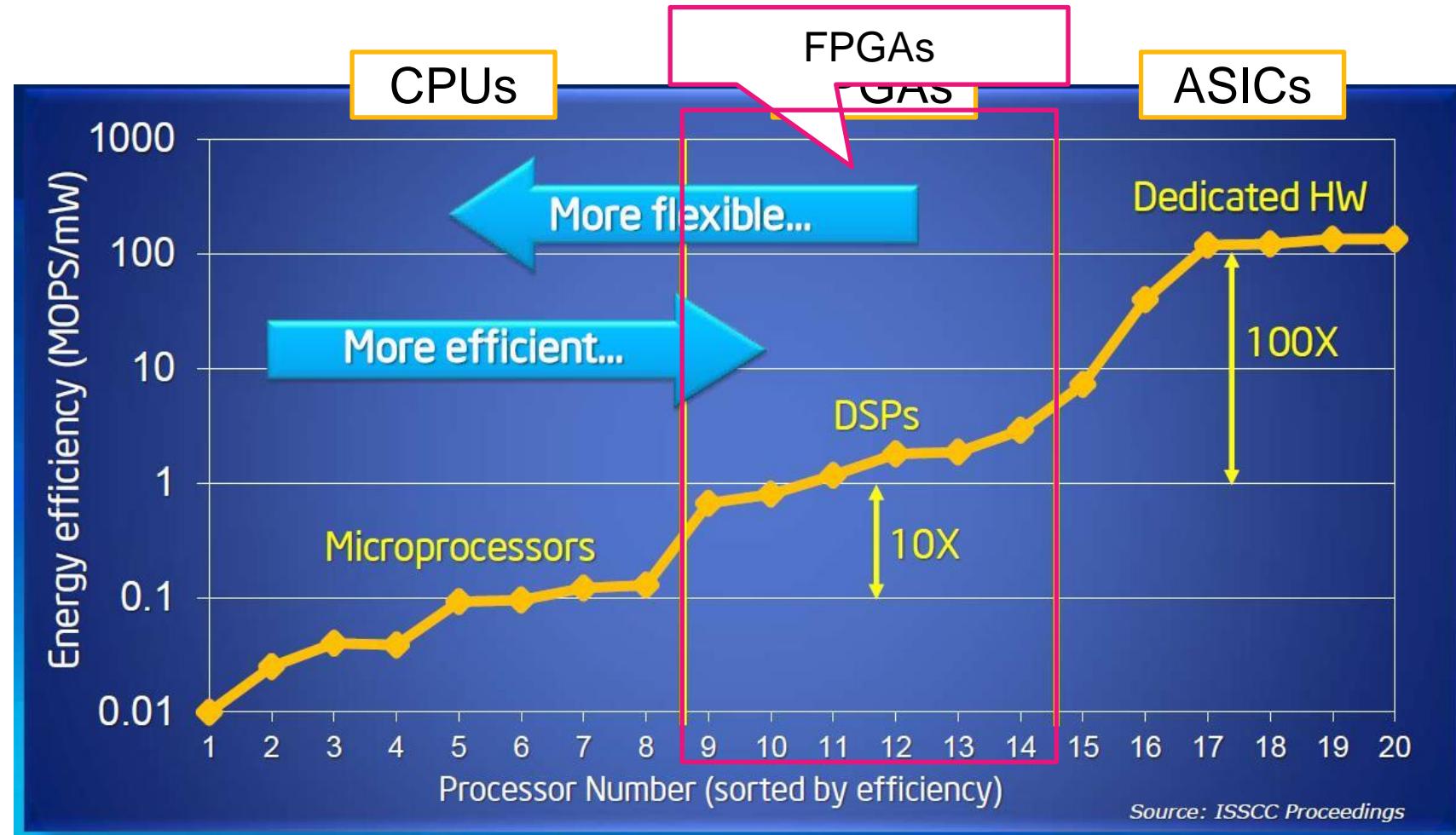
- Switch ASIC programmed with P4
- The chip “knows” only networking-related processing



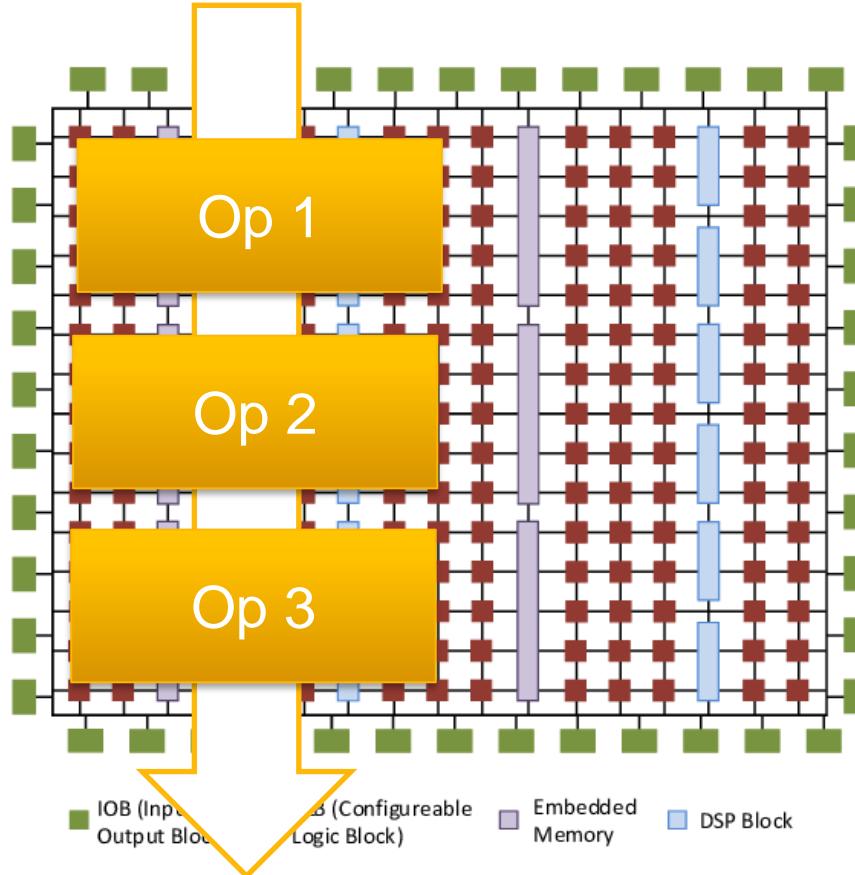
P4₁₆ Language Elements



Renewed interest in Specialized Hardware



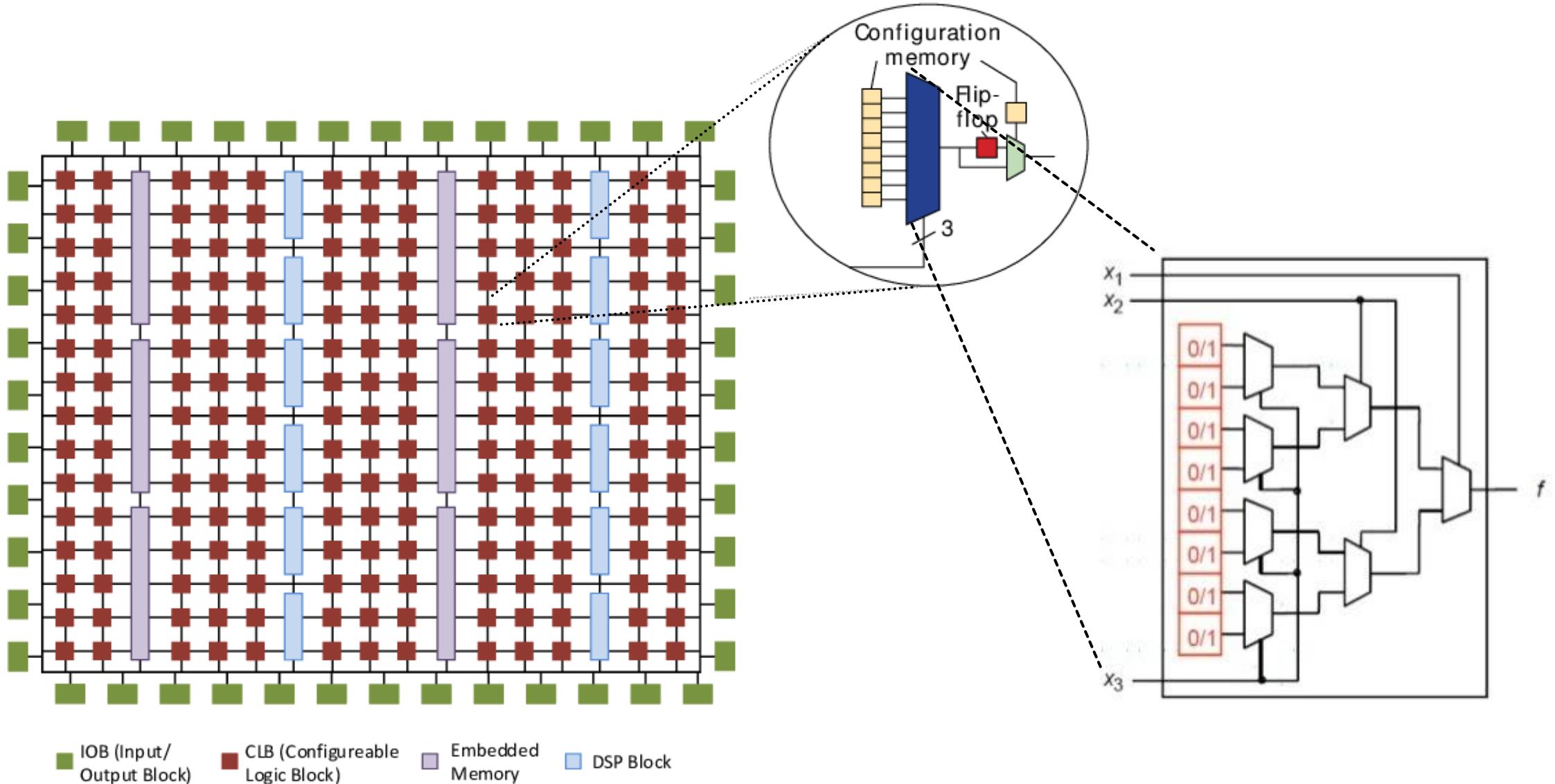
Re-programmable Specialized Hardware



Field Programmable Gate Array (FPGA)

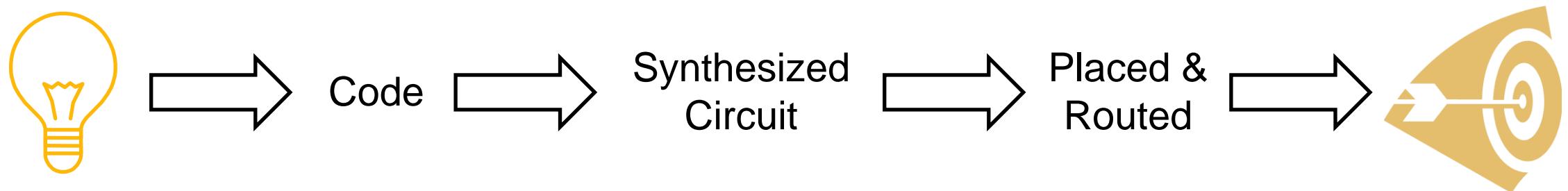
- Free choice of architecture
- Fine-grained pipelining, communication, distributed memory
- Tradeoffs
 - all “code” occupies chip space
 - frequencies in 100s MHz range

What is inside a Logic Block?



Programming FPGAs

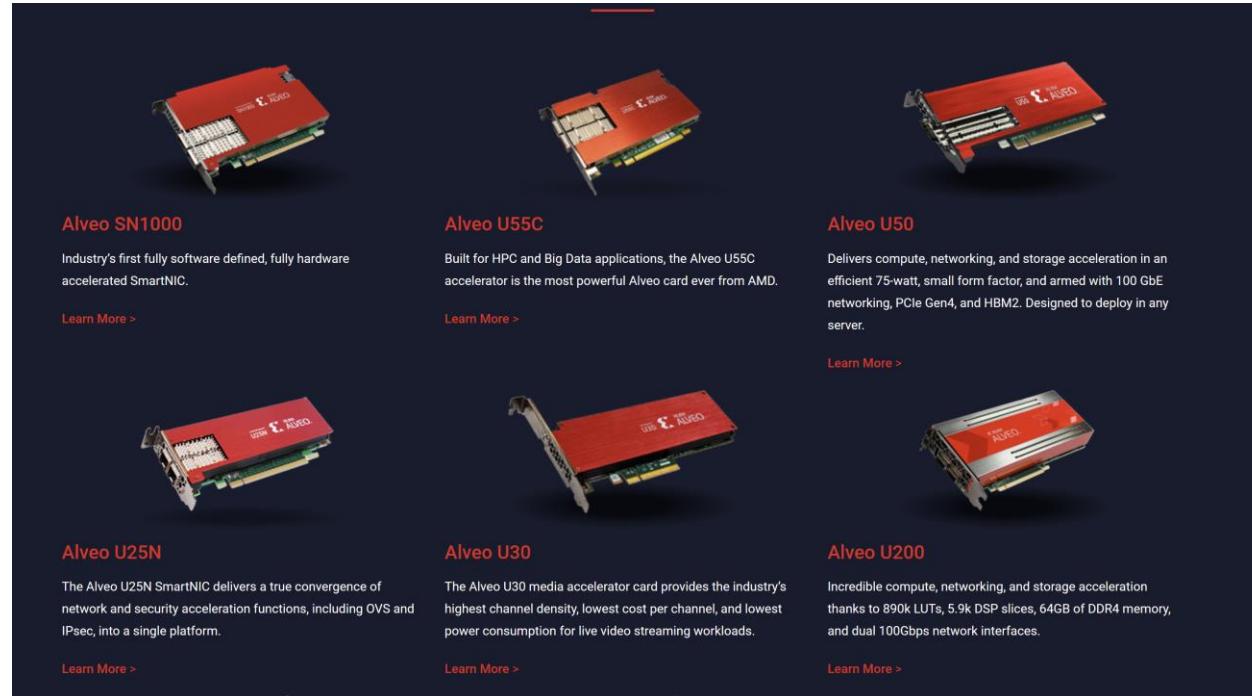
- Challenge: adapting algorithms to the parallelism of the FPGA



- Coding: Hardware definition languages, high level languages
- Synthesis: Produce a logic-gate level representation (any FPGA)
- Place & route: Circuit that gets mapped onto specific FPGA

In-network and Network-attached FPGAs

- Modern FPGAs offer several high speed (up to 100Gbps) network ports
- Operate with or without host CPU support



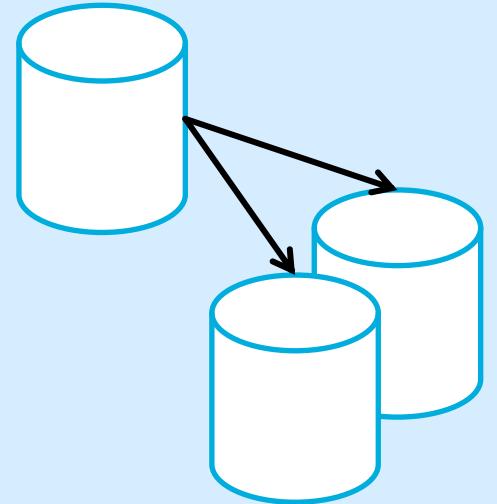
Outline

- Motivation for Specialized Networking
- Specialization – Concepts and Devices
- The Coordination Bottleneck
 - **Different ways of alleviating it using specialization**

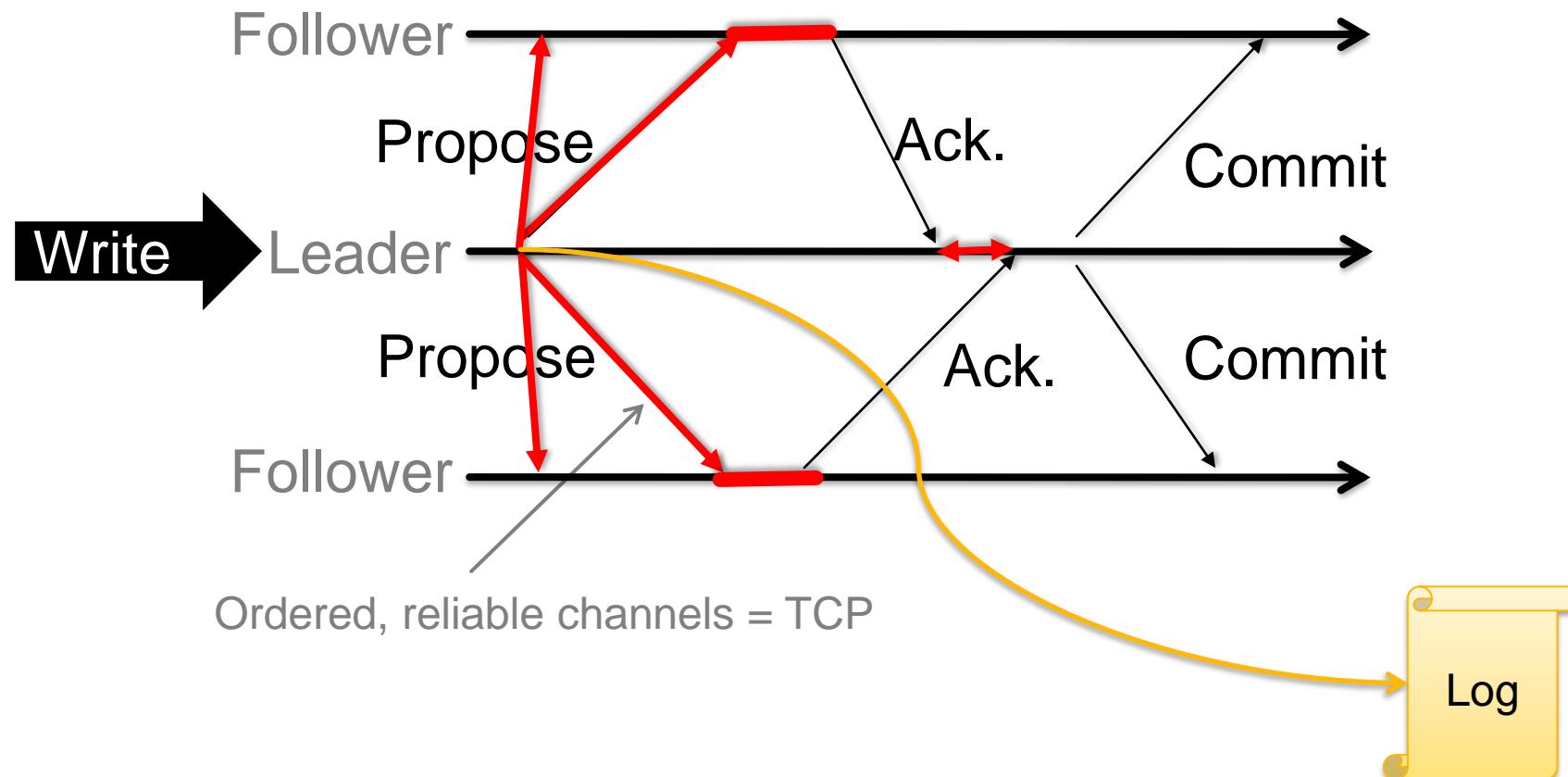
Network bottlenecks in datacenter and cloud

- Distributed systems are everywhere (Spark, ML, databases...)
 - Data movement bottlenecks
 - Get data across at HIGH BANDWIDTH
 - Coordination bottlenecks
 - Get to agreement with LOW LATENCY
- Focus on different ways to provide efficient coordination

Modern networks + NICs
mostly solve this



Zookeeper's Atomic Broadcast Protocol



Protocol described in: F. P. Junqueira, B. C. Reed, et al. Zab: High-performance broadcast for primary-backup systems. In DSN'11.

TCP/IP Refresher

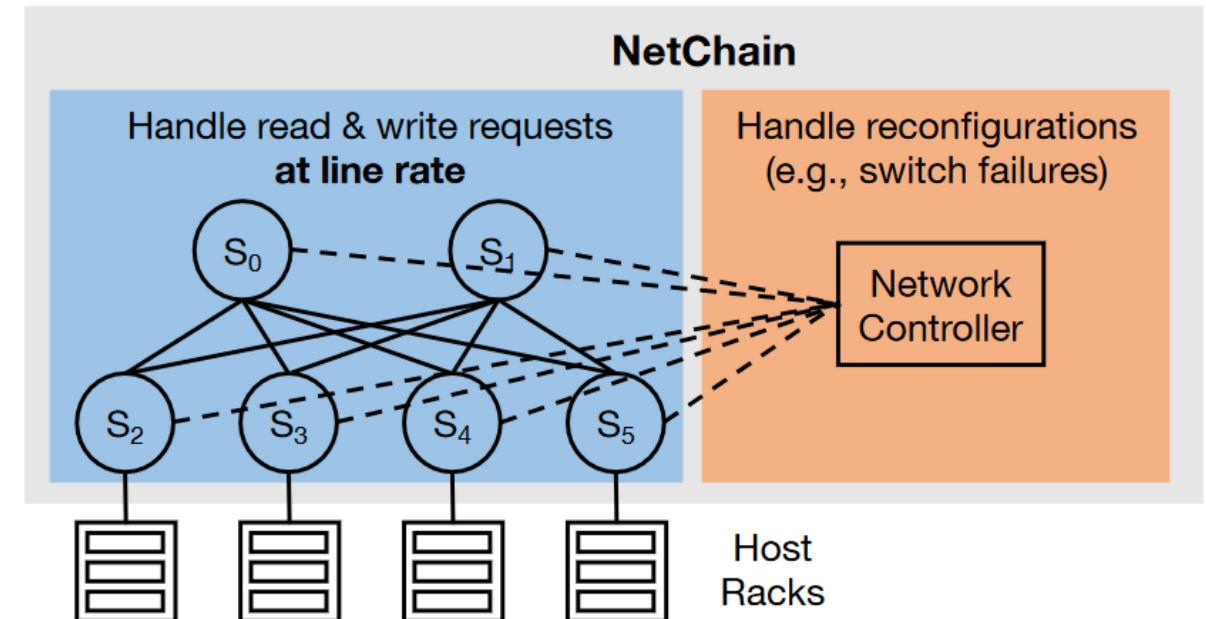
- Provide “stream” abstraction on top of packet-switched network
- TCP protocol ensures
 1. Reliability (keep track of what has been received, ACKs)
 2. Flow control (manage and monitor buffers on both ends, Windows)
- Typically implemented in OS kernel
 - TCP Sockets exposed to applications
- Lots of CPU cycles for each packet...

How to speed up consensus rounds?

- Can we rely on multi-core parallelism?
 - Atomic Broadcast needs to execute operations sequentially – maps to single-threaded execution
 - Can pipeline!
- In traditional solutions, network access is often an overhead
 - Traversing several software and hardware layers
- *One idea: perform operation directly on network level, specialize fully?*

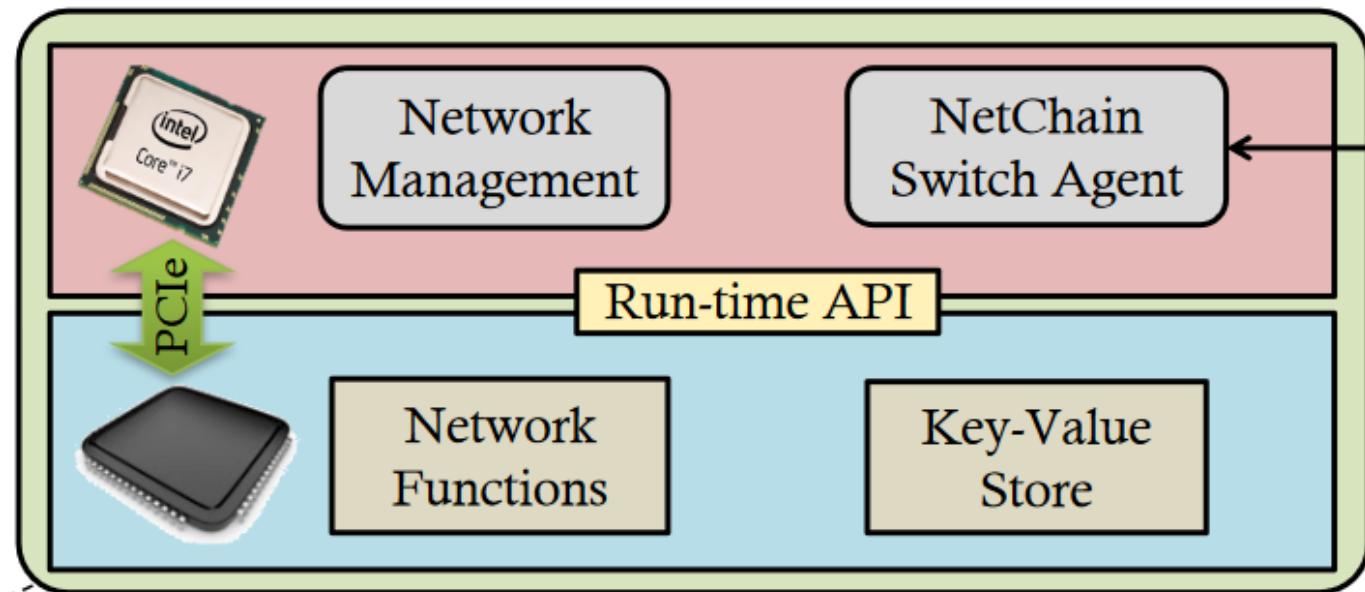
Programmable Switches and Consensus

- NetChain
 - Small dataset residing inside the switch
 - Replication achieved by manipulating Ethernet packets

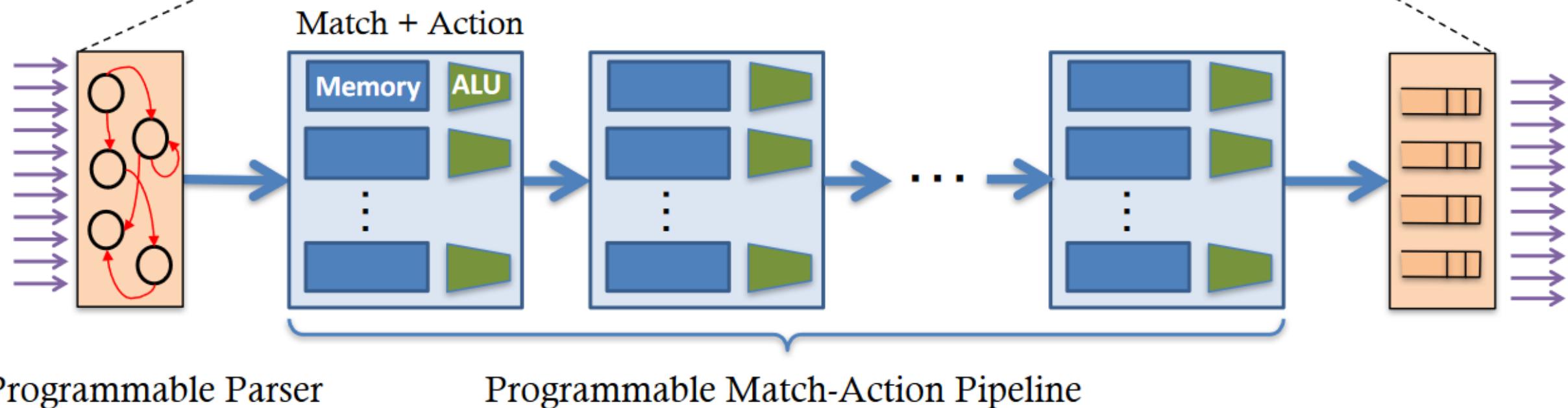


Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, Ion Stoica: **NetChain: Scale-Free Sub-RTT Coordination**. NSDI 2018: 35-49

Control plane (CPU)

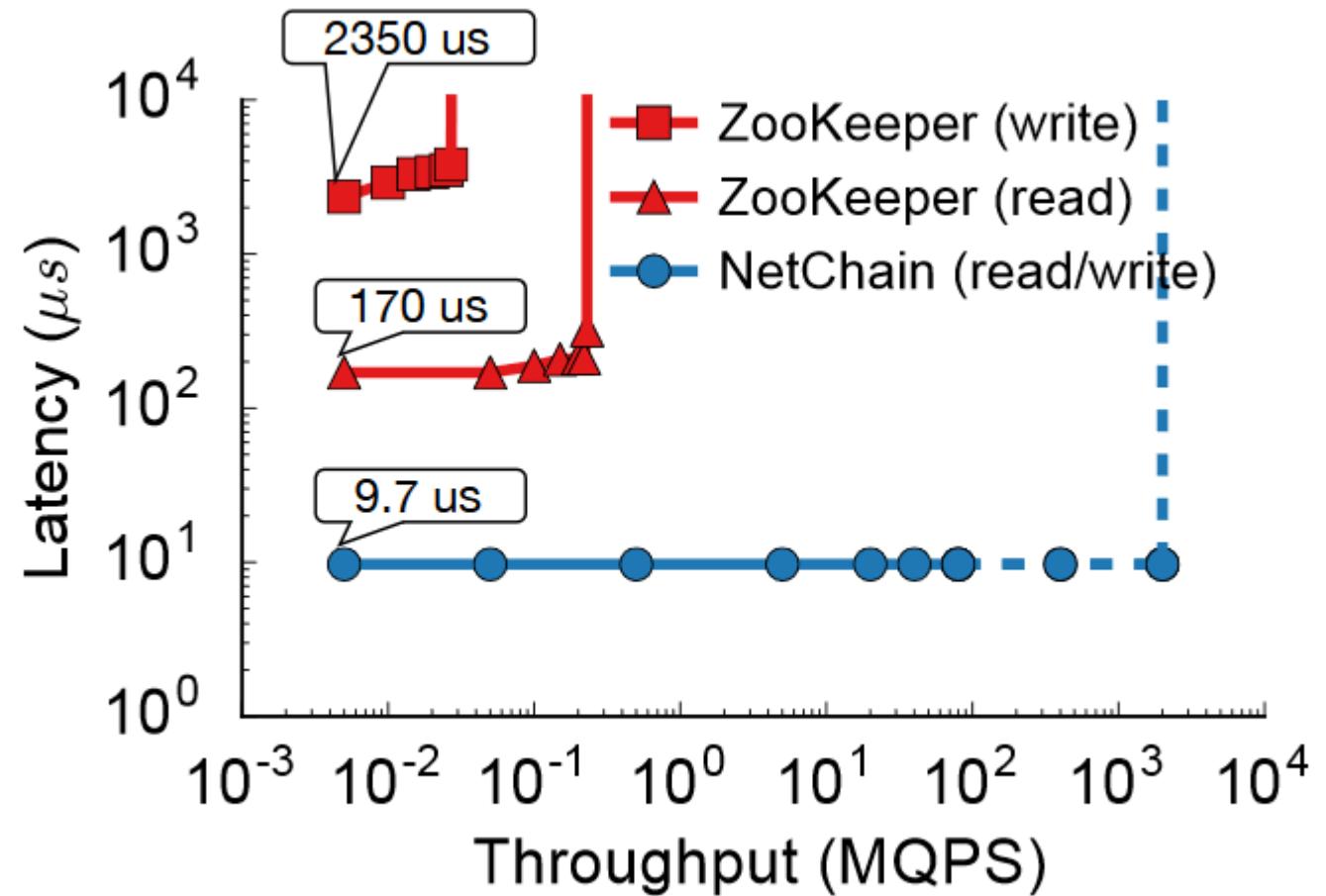


Data plane (ASIC)



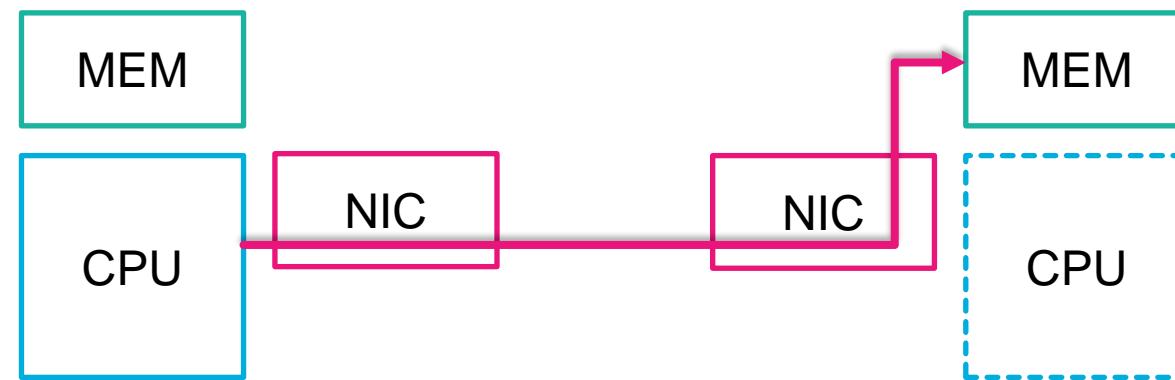
NetChain Performance

- Not end-to-end
- The switch becomes the node
- Small memory capacity
- Hard to program (P4)
- Hard to integrate with other software

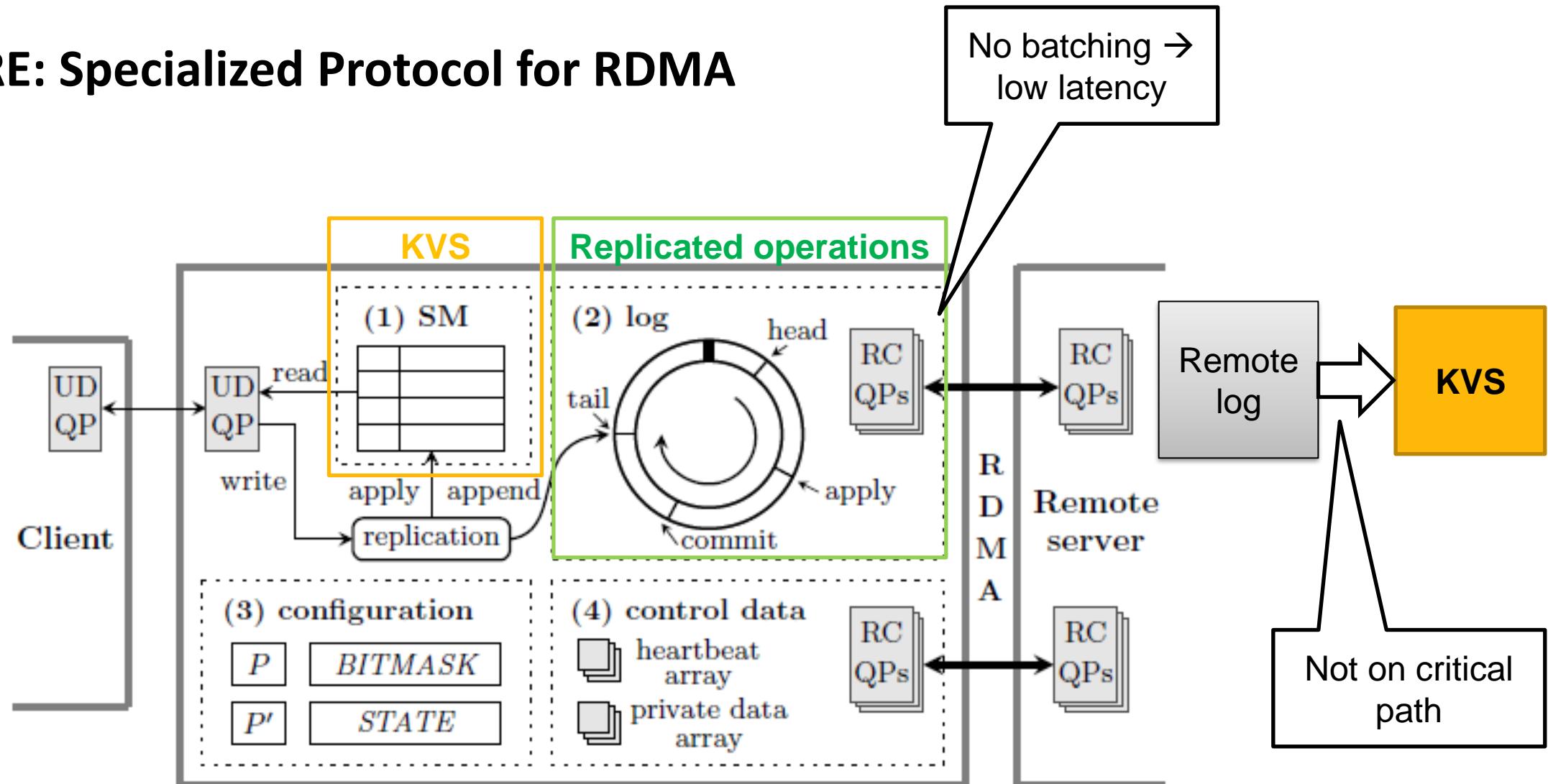


How to speed up consensus rounds?

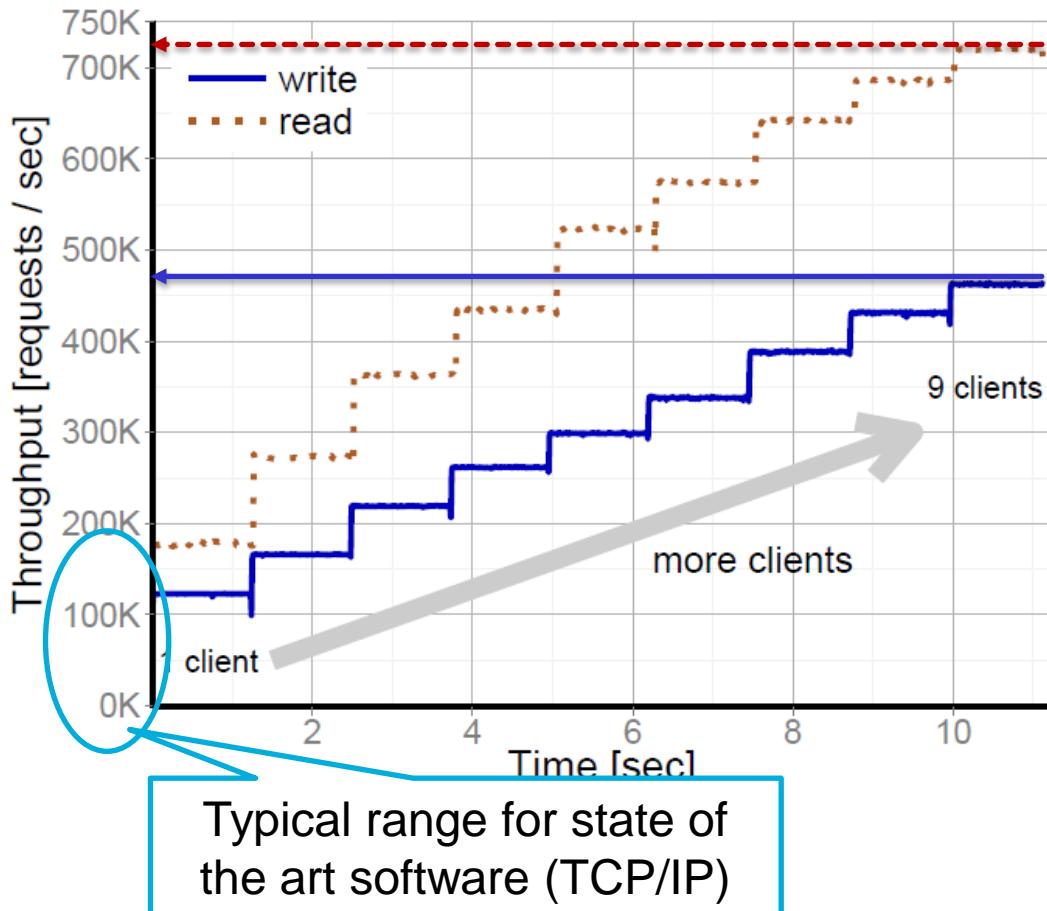
- In traditional solutions, network access is often an overhead
 - Traversing several software and hardware layers
- *Idea: Can we remove the CPU from the critical path of performance?*
- Remote Direct Memory Access (RDMA)
 - Can read/write data structures on remote machines without involving the remote CPU



DARE: Specialized Protocol for RDMA

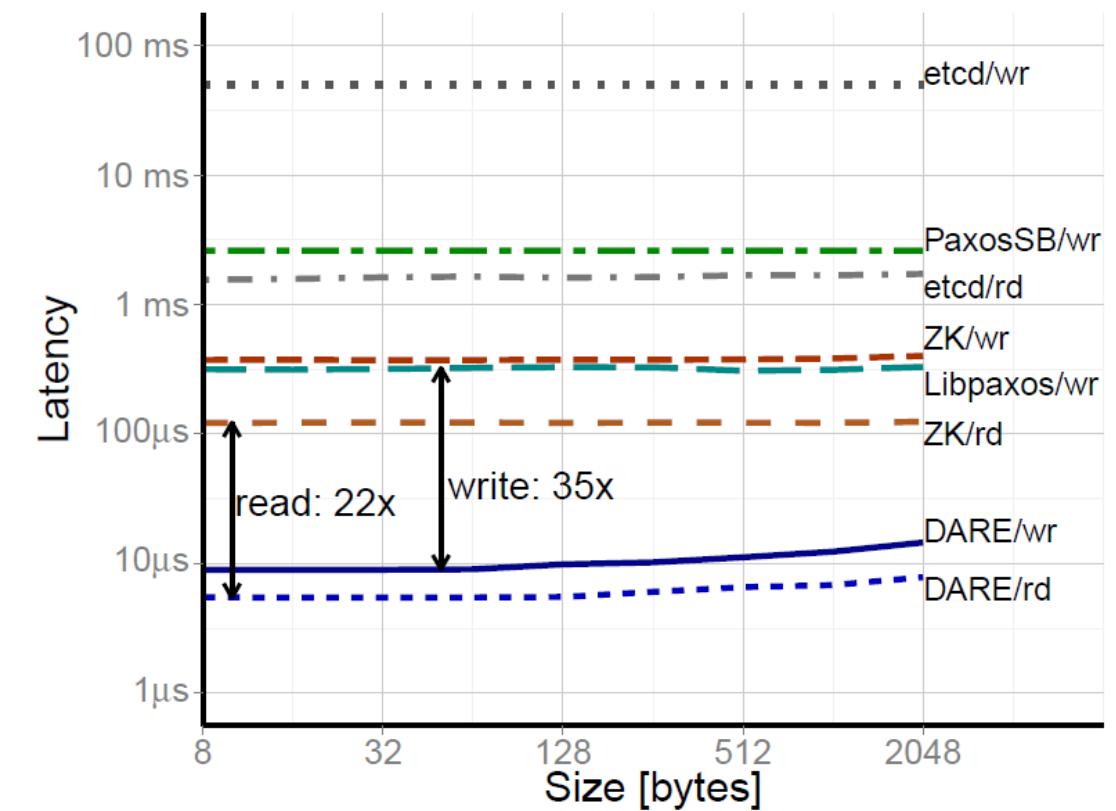


DARE Performance



Excellent performance but

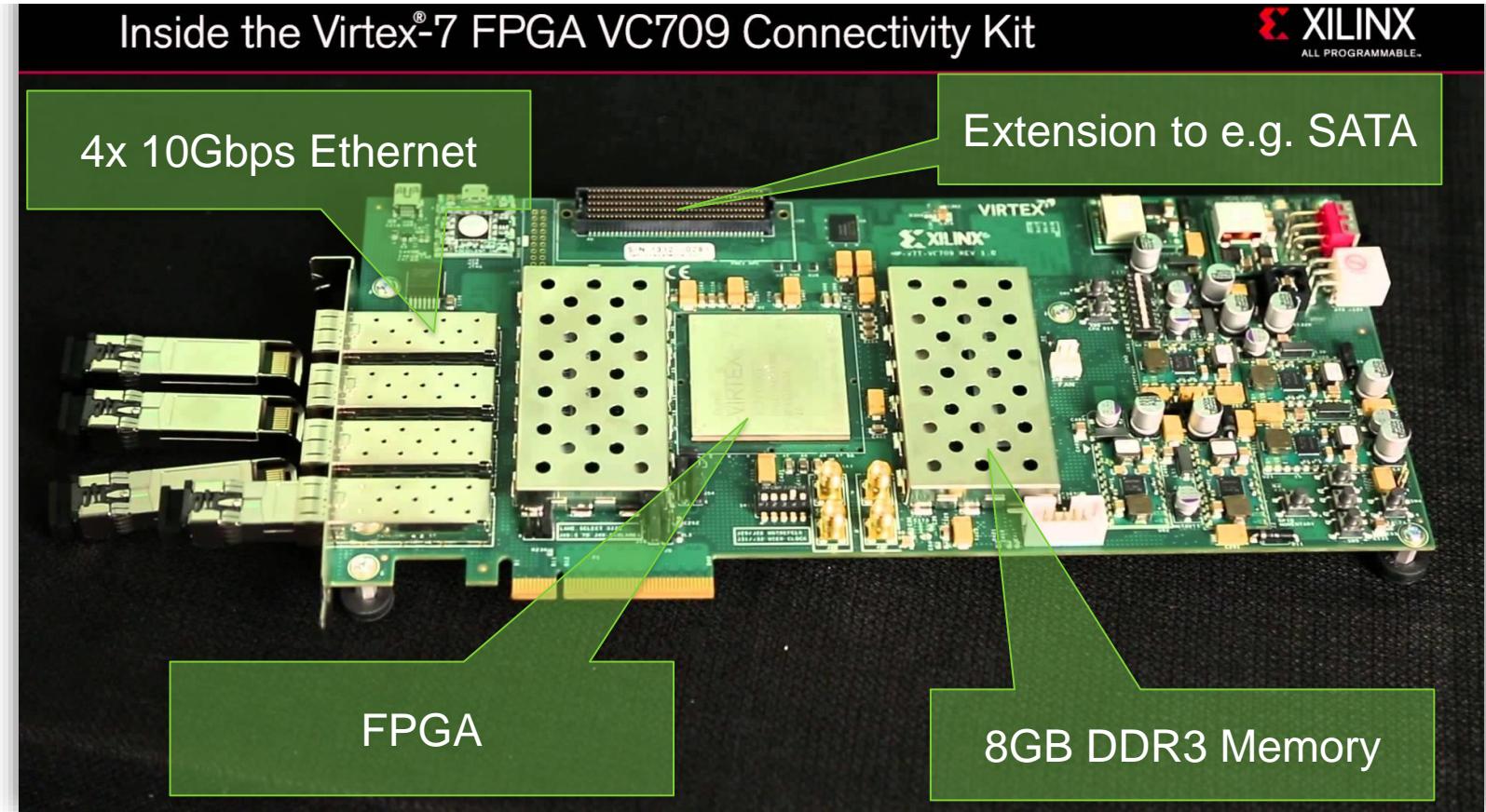
- Complex logic, many failure options
- Hard to integrate in cloud/datacenter, larger applications



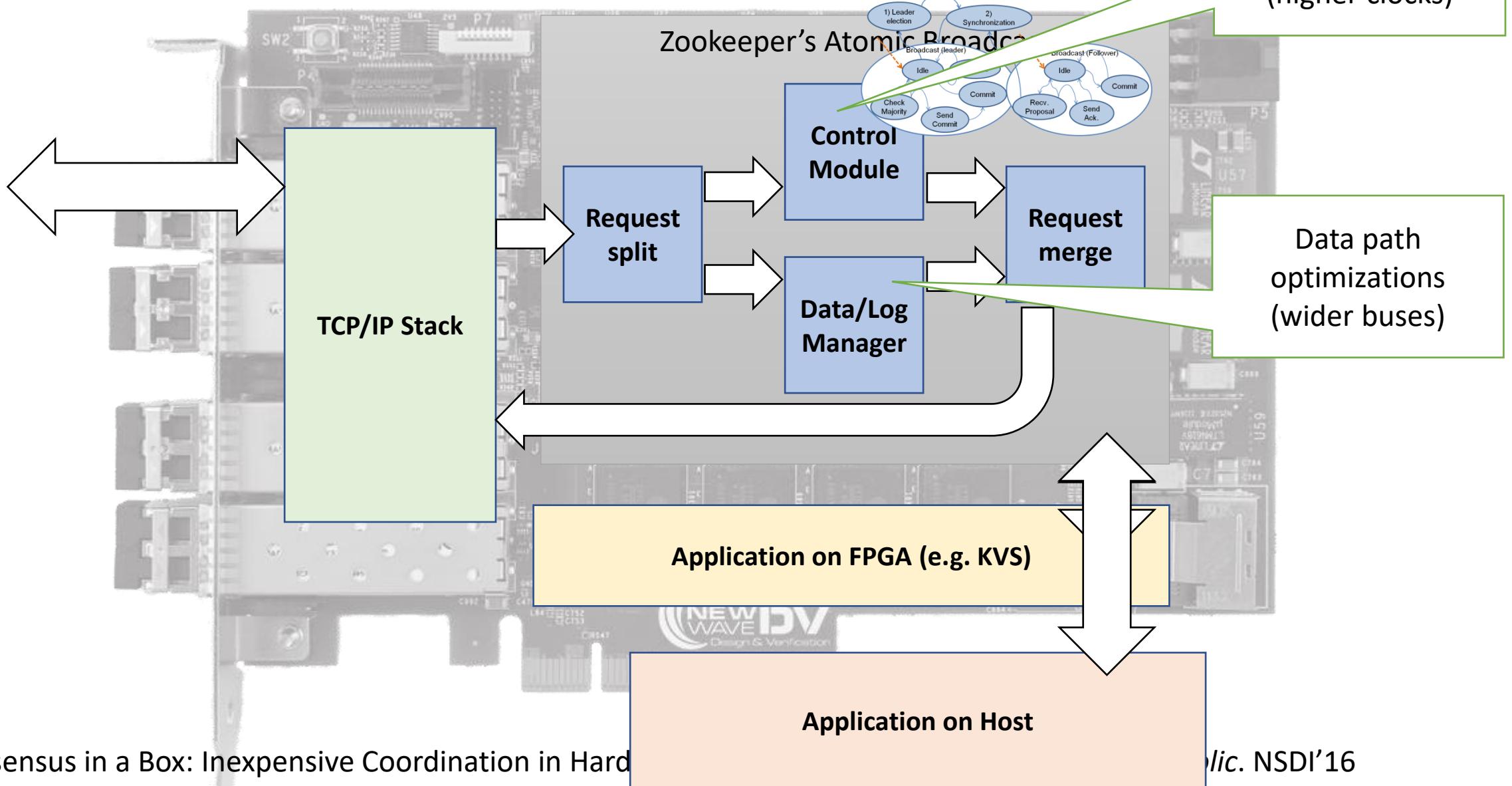
How to speed up consensus rounds?

- In traditional solutions, network access is often an overhead
 - Traversing several software and hardware layers
- Idea: specialize (parts of) the server, but keep
 - Commodity network – no change in infrastructure
 - TCP/IP sockets – no change in client logic
 - Widely-deployed consensus algorithm – no risk of accidental data corruption

FPGA Prototype device



Consensus in a Box



What makes it fast...

Latency:

- **Networking optimizations**
 - Low-latency on-chip buffers for RX path
 - Datacenter-specific optimizations to TCP/IP
- **Predictable behavior**
 - Tailored local data structures for common case behavior

Throughput:

- **Pipelined execution**



Challenges of Implementing TCP on FPGAs

- Complex control flow – challenging to code in low level language
- Many parallel connections, state for each – only limited space in on-chip memory (MBs)
- Retransmission needs large buffers – high latency access to DDR memory, operating at different timeframes than normal processing (100s of us instead of 100s of ns)

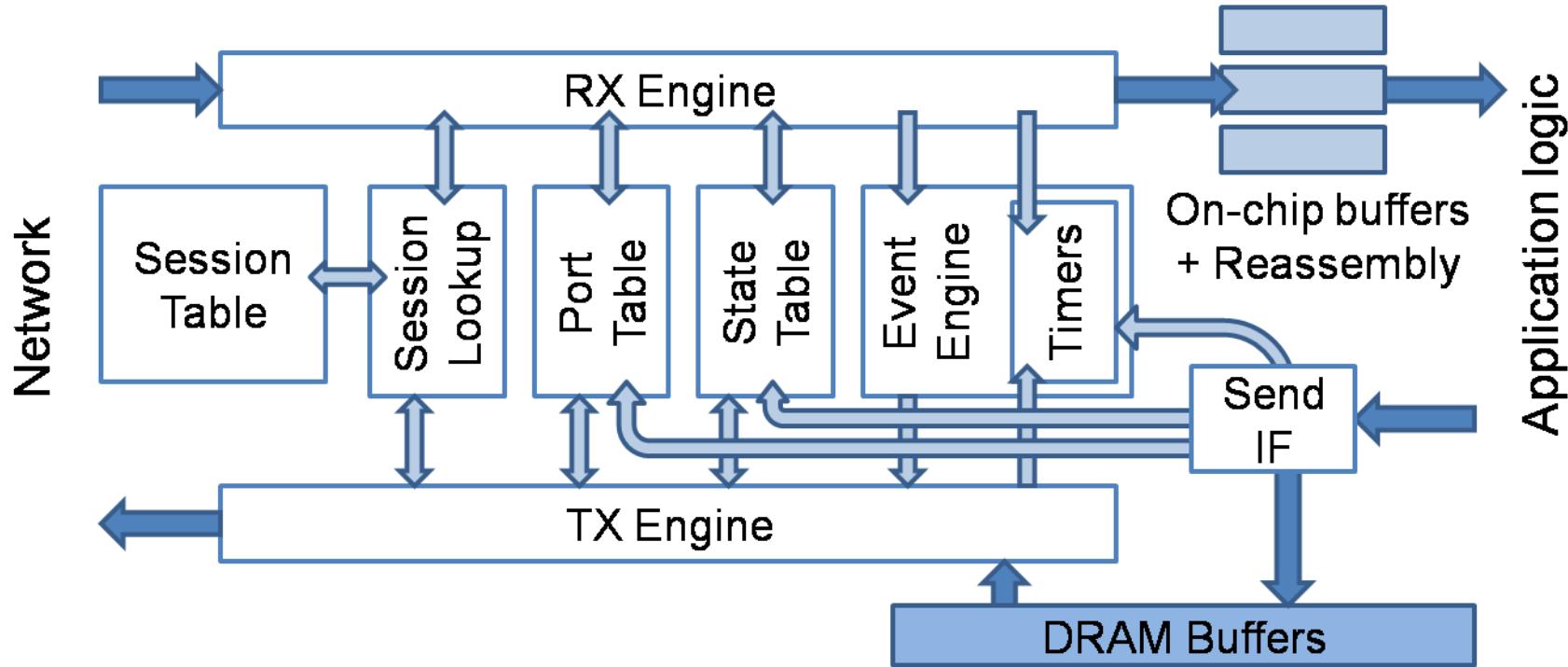
How we implemented TCP on FPGAs

Tailored it to datacenter use-case: client-server on fast networks

- Number of concurrent clients limited (e.g. 10k) – less on-chip buffer space
- If FPGA implements application logic, RX reassembly can be done by app – no RX buffer in off-chip memory
- Can assume lower latencies – faster and less frequent retransmissions from TX buffers off-chip
- Low latency matters but not nanosecond level – could use High Level Synthesis (HLS) – simplified implementation

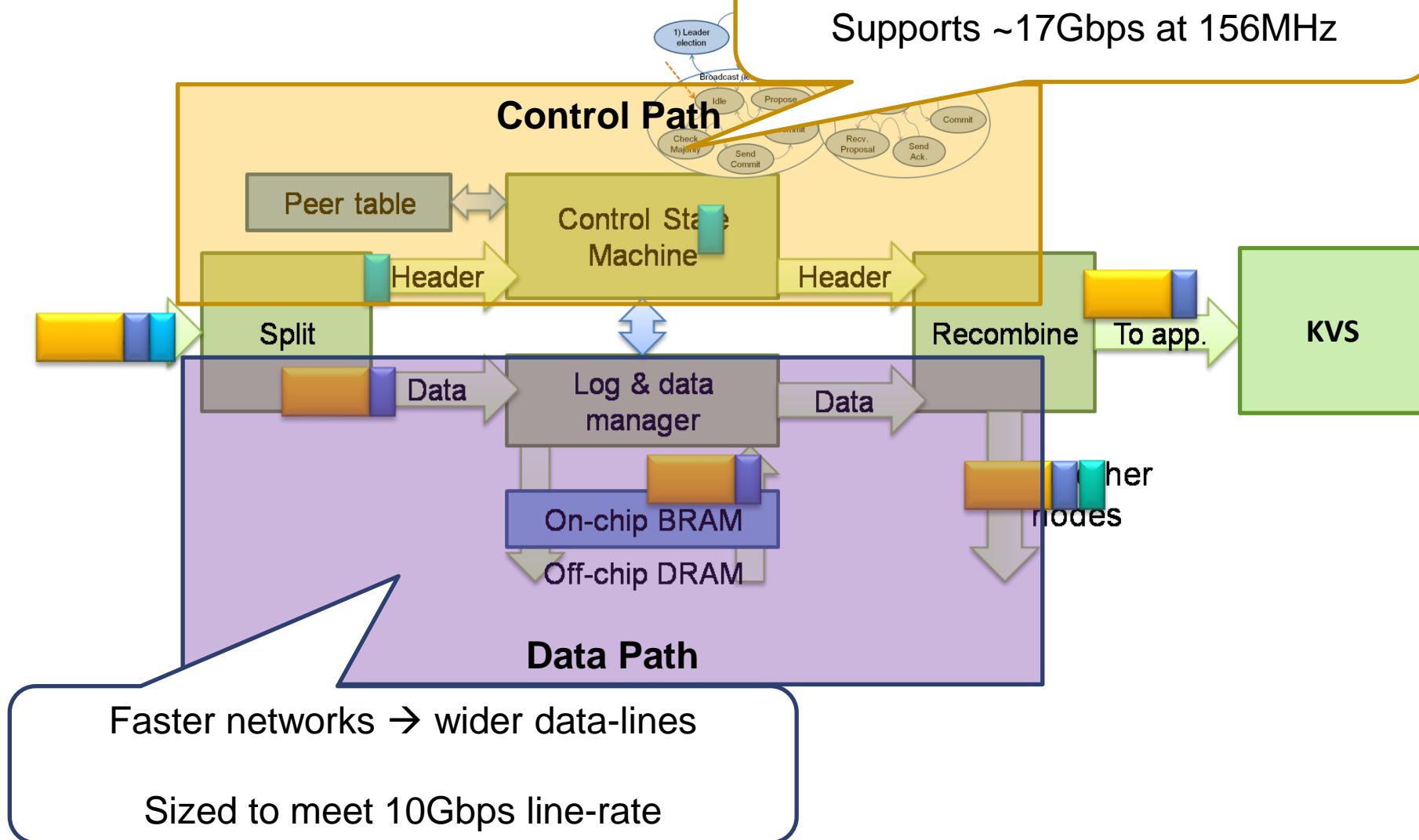


Networking pipeline

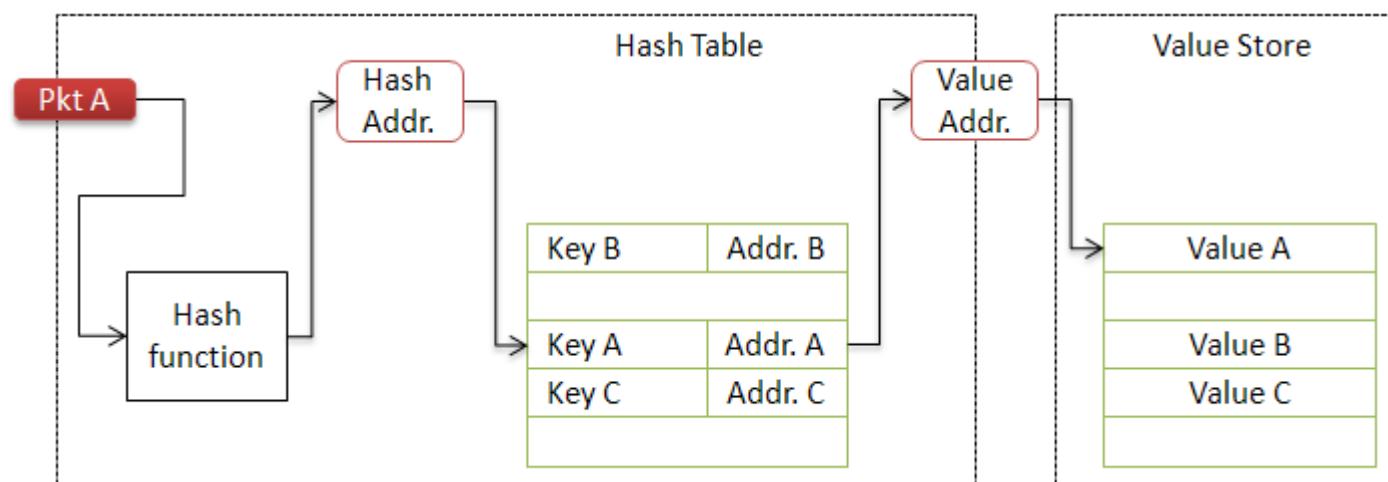
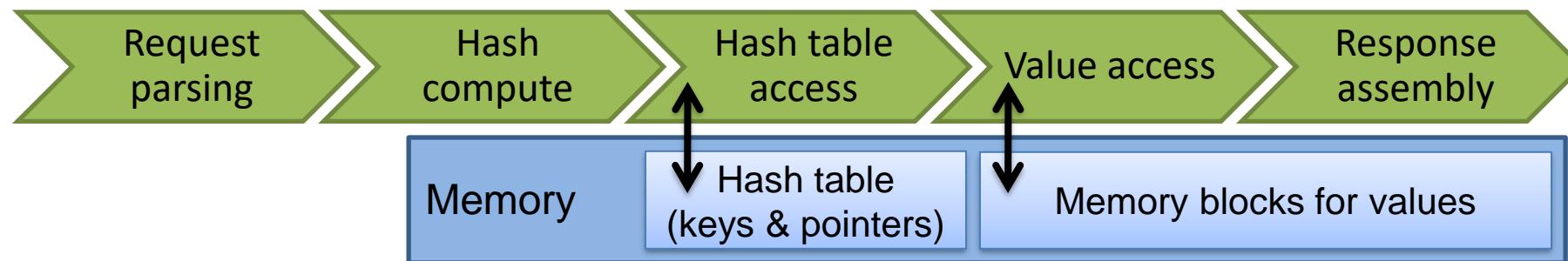


- Up to 10k connections
- Supports DHCP for IP assignment

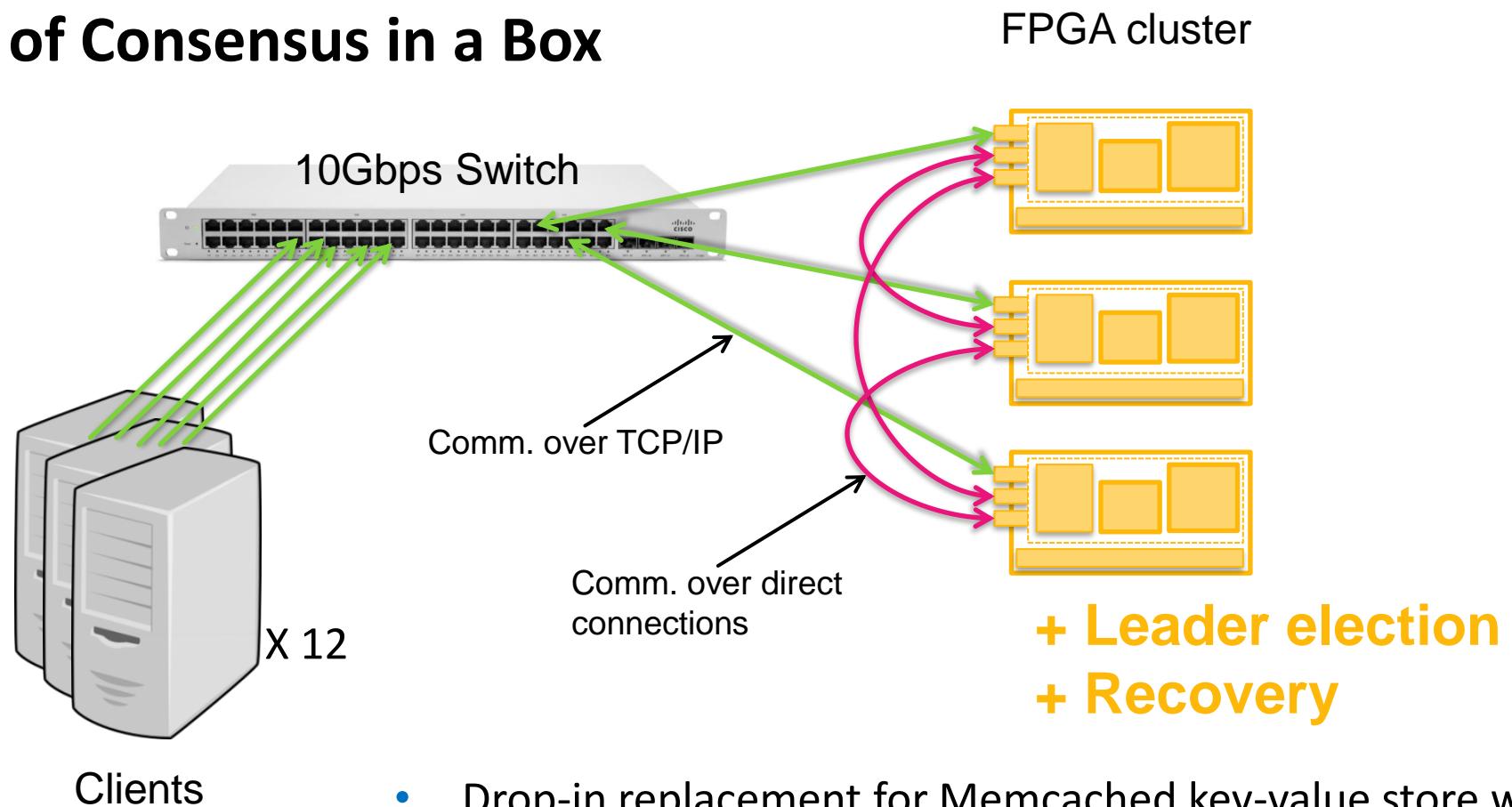
Consensus Logic as Pipeline



Key-value store

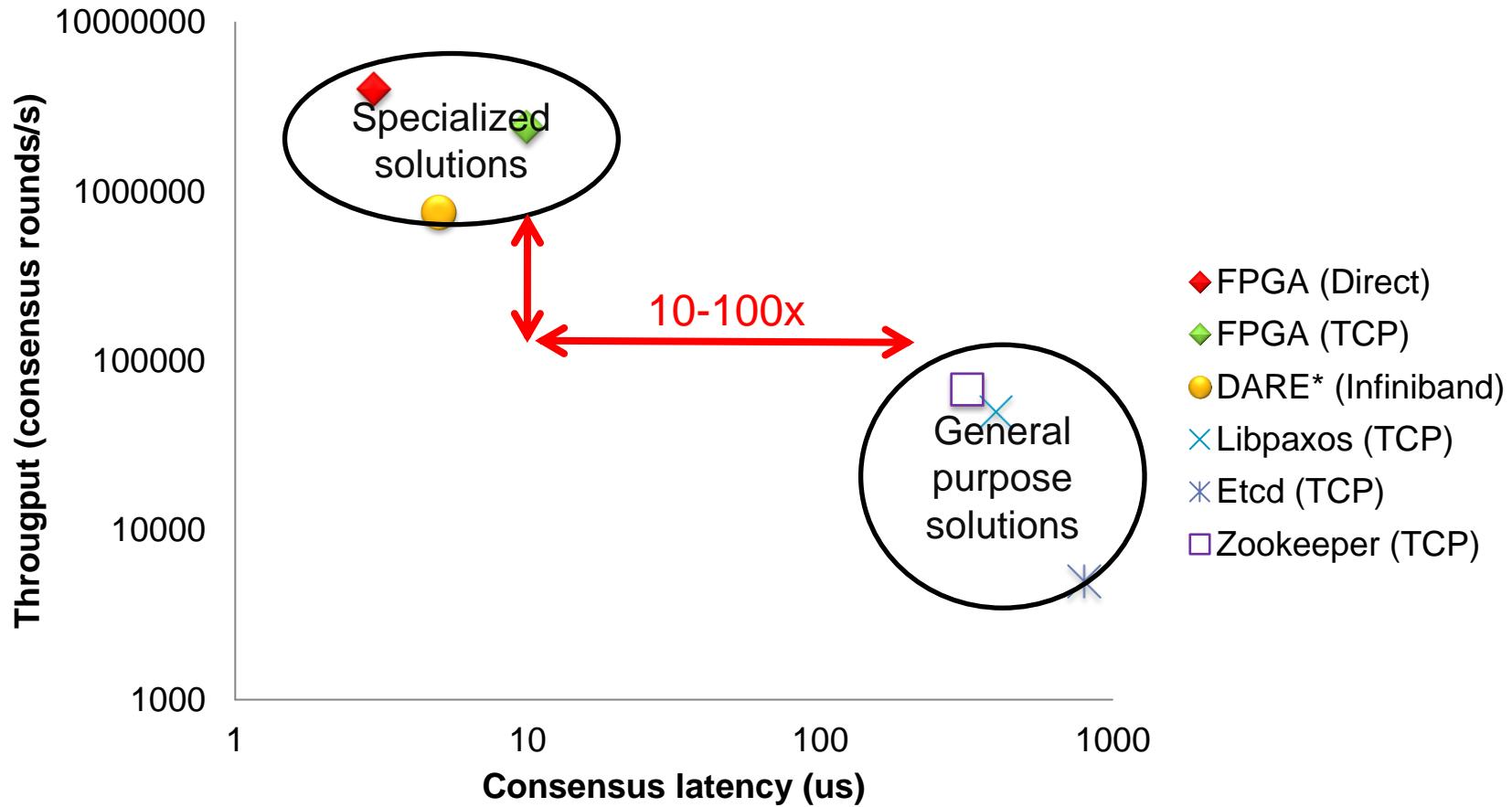


Evaluation setup of Consensus in a Box



- Drop-in replacement for Memcached key-value store with Zookeeper's replication
 - Standard tools for benchmarking (libmemcached)
 - Simulating 100s of clients
- Custom binary protocol
 - Go-based client library

The benefit of specialization...



[1] Dragojevic et al. FaRM: Fast Remote Memory. In NSDI'14.

[2] Poke et al. DARE: High-Performance State Machine Replication on RDMA Networks. In HPDC'15.

*=We extrapolated from the 5 node setup for a 3 node setup.

Platform for starting new research projects

- The work presented before is now in Multes KVS with 100Gbps networking
https://github.com/zistvan/Multes_for_Vitis_with_100Gbps_TCP-IP

Specialize in Moderation – Building
FPGAs into Smart Storage

Lucas Kuhring
IMDEA Software Institute, Madrid, Spain

IMDEA Software Institute

Abstract

In order to keep up with big data workloads, distributed storage needs to offer low latency, high bandwidth and efficient access to data. To achieve these properties, most state-of-the-art solutions focus either exclusively on software or hardware-based implementation. FPGAs are an example of the latter and a promising platform for building storage nodes, but they are more cumbersome to program and less flexible than software, which limits their adoption.

We make the case that, in order to be feasible in the cloud, solutions designed around programmable hardware, such as FPGAs, have to follow a service provider-centric methodology: the hardware should only provide functionality that is useful across all tenants and rarely changes. Consequently, application-specific functionality should be delivered thro-

The Case for Adding Privacy-Related Operators to Network-attached Smart Storage

Claudiu Mihali
Aalto University, Finland

Anca Hangan
UTCN, Romania

Gheorghe Sebestyen
UTCN, Romania

ABSTRACT

It is important to ensure that personally identifiable information (PII) is protected within large distributed systems and is used only for intended purposes. Achieving this is challenging and several techniques have been proposed for privacy-preserving analytics, but they typically focus on the end hosts only. We argue that future storage solutions should include, in addition to emerging compute offload, also privacy-related operators. Since many privacy operators, such as perturbation and anonymization, take place as the very first step before other computations, query offload to a Smart Storage device might be only feasible in the future if privacy-related operators can also be offloaded.

In this work we demonstrate that privacy-preserving operators can be implemented in hardware without reducing read bandwidths. We focus on perturbations and extend an FPGA-based network-attached Smart Storage solution to show that it is possible to provide these operations at 10Gbps line-rate while using only a small amount of additional FPGA real estate. We also discuss how future faster smart storage nodes

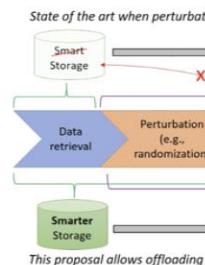


Figure 1: Some perturbation is being processed by query. This proposal allows offloading to storage.

1 INTRODUCTION

Big Data has revolutionized

In-Storage Computation of Histograms with Differential Privacy

Andrei Tosa, Anca Hangan, Gheorghe Sebestyen
Technical University of Cluj-Napoca, Romania
{firstname.lastname}@cs.utcluj.ro

Zsolt István
TU Darmstadt, Germany
zsolt.istvan@cs.tu-darmstadt.de

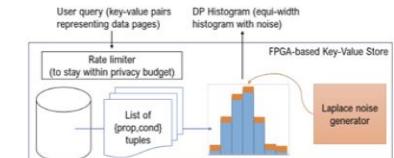


Fig. 1. To compute differentially private histograms, regular histograms are computed on the data and noise is added to the counts. The frequency at which such histograms can be created on a dataset has to be controlled, to disallow statistical attacks on the results.

Motivated by the above, we explore how one can offload differentially private histogram computation to an FPGA in a Smart Storage node. Our prototype builds on an open-source key-value store and exposes two interfaces: a traditional

Summary

- Distributed Systems are everywhere – but facing scalability bottlenecks
 - Increasing link speed not enough due to CPU bottleneck
 - Idea: Revisit protocols and processing paradigms -> Specialization
- Specialized Hardware becoming common in Clouds and Datacenters
- FPGAs: middle ground between performance and programmability
 - Can be used well for packet-oriented, network processing
- Challenges when implementing with FPGAs
 - Adopting algorithms and parallelism to the device (pipelining!)
- For all specialized solutions: Integration is often *more* important than raw performance!