



Overview

Introduction to Parallel Computing

CIS 410/510

Department of Computer and Information Science



UNIVERSITY OF OREGON

Outline

□ Course Overview

- What is CIS 410/510?
- What is expected of you?
- What will you learn in CIS 410/510?

□ Parallel Computing

- What is it?
- What motivates it?
- Trends that shape the field
- Large-scale problems and high-performance
- Parallel architecture types
- Scalable parallel computing and performance

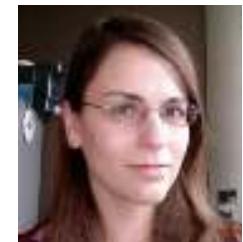
How did the idea for CIS 410/510 originate?

- There has never been an undergraduate course in parallel computing in the CIS Department at UO
- Only 1 course taught at the graduate level (CIS 631)
- Goal is to bring parallel computing education in CIS undergraduate curriculum, start at senior level
 - CIS 410/510 (Spring 2014, “experimental” course)
 - CIS 431/531 (Spring 2015, “new” course)
- CIS 607 – Parallel Computing Course Development
 - Winter 2014 seminar to plan undergraduate course
 - Develop 410/510 materials, exercises, labs, ...
- Intel gave a generous donation (\$100K) to the effort
- NSF and IEEE are spearheading a curriculum initiative for undergraduate education in parallel processing

<http://www.cs.gsu.edu/~tcpp/curriculum/>

Who's involved?

- Instructor
 - Allen D. Malony
 - ◆ scalable parallel computing
 - ◆ parallel performance analysis
 - ◆ taught CIS 631 for the last 10 years
- Faculty colleagues and course co-designers
 - Boyana Norris
 - ◆ Automated software analysis and transformation
 - ◆ Performance analysis and optimization
 - Hank Childs
 - ◆ Large-scale, parallel scientific visualization
 - ◆ Visualization of large data sets
- Intel scientists
 - Michael McCool, James Reinders, Bob MacKay
- Graduate students doing research in parallel computing



Intel Partners

□ James Reinders

- Director, Software Products
- Multi-core Evangelist



□ Michael McCool

- Software architect
- Former Chief scientist, RapidMind
- Adjunct Assoc. Professor, University of Waterloo



□ Arch Robison

- Architect of Threading Building Blocks
- Former lead developers of KAI C++



□ David MacKay

- Manager of software product consulting team



CIS 410/510 Graduate Assistants

□ Daniel Ellsworth

- 3rd year Ph.D. student
- Research advisor (Prof. Malony)
- Large-scale online system introspection



□ David Poliakoff

- 2nd year Ph.D. student
- Research advisor (Prof. Malony)
- Compiler-based performance analysis



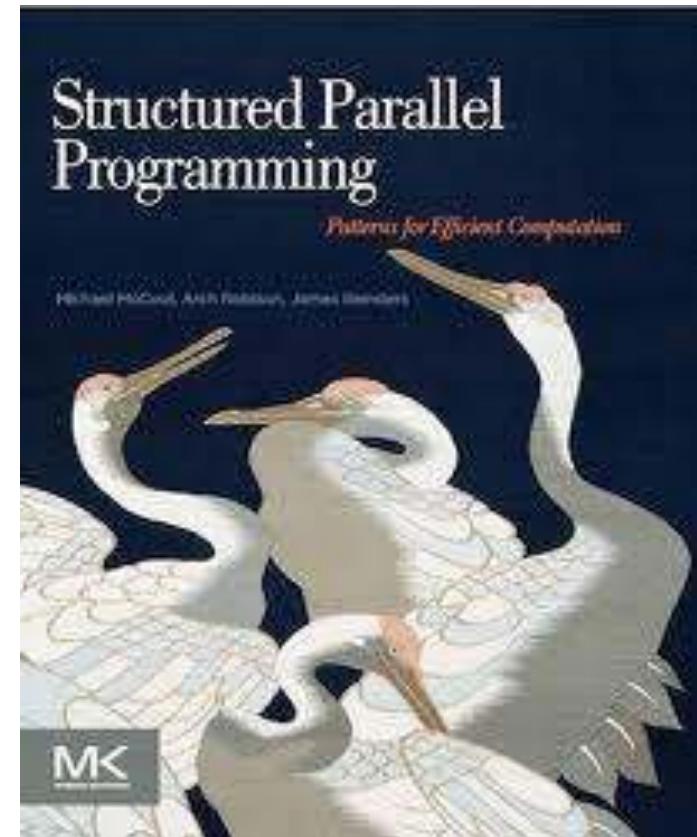
□ Brandon Hildreth

- 1st year Ph.D. student
- Research advisor (Prof. Malony)
- Automated performance experimentation



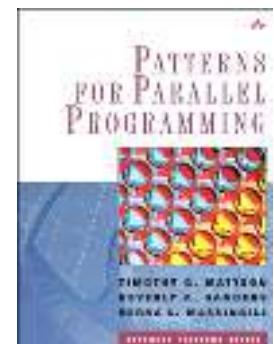
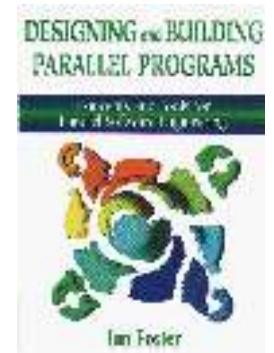
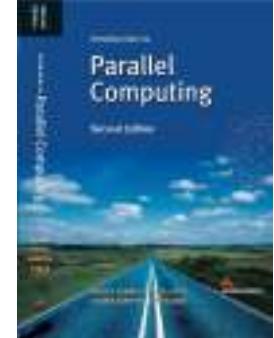
Required Course Book

- “Structured Parallel Programming: Patterns for Efficient Computation,” Michael McCool, Arch Robinson, James Reinders, 1st edition, Morgan Kaufmann, ISBN: 978-0-12-415993-8, 2012
<http://parallelbook.com/>
- Presents parallel programming from a point of view of patterns relevant to parallel computation
 - Map, Collectives, Data reorganization, Stencil and recurrence, Fork-Join, Pipeline
- Focuses on the use of shared memory parallel programming languages and environments
 - Intel Thread Building Blocks (TBB)
 - Intel Cilk Plus



Reference Textbooks

- *Introduction to Parallel Computing*, A. Grama, A. Gupta, G. Karypis, V. Kumar, Addison Wesley, 2nd Ed., 2003
 - Lecture slides from authors online
 - Excellent reference list at end
 - Used for CIS 631 before
 - Getting old for latest hardware
- *Designing and Building Parallel Programs*, Ian Foster, Addison Wesley, 1995.
 - Entire book is online!!!
 - Historical book, but very informative
- *Patterns for Parallel Programming* T. Mattson, B. Sanders, B. Massingill, Addison Wesley, 2005.
 - Targets parallel programming
 - Pattern language approach to parallel program design and development
 - Excellent references



What do you mean by experimental course?

- Given that this is the first offering of parallel computing in the undergraduate curriculum, we want to evaluate how well it worked
- We would like to receive feedback from students throughout the course
 - Lecture content and understanding
 - Parallel programming learning experience
 - Book and other materials
- Your experiences will help to update the course for its debut offering in (hopefully) Spring 2015

Course Plan

- Organize the course so that cover main areas of parallel computing in the lectures
 - Architecture (1 week)
 - Performance models and analysis (1 week)
 - Programming patterns (paradigms) (3 weeks)
 - Algorithms (2 weeks)
 - Tools (1 week)
 - Applications (1 week)
 - Special topics (1 week)
- Augment lecture with a programming lab
 - Students will take the lab with the course
 - ◆ graded assignments and term project will be posted
 - Targeted specifically to shared memory parallelism

Lectures

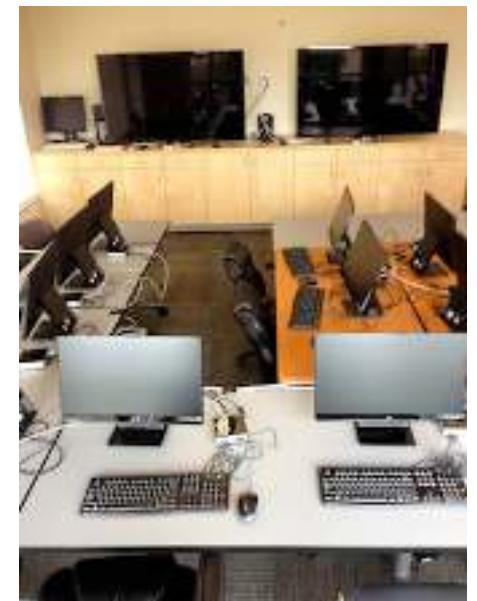
- Book and online materials are your main sources for broader and deeper background in parallel computing
- Lectures should be more interactive
 - Supplement other sources of information
 - Covers topics of more priority
 - Intended to give you some of my perspective
 - Will provide online access to lecture slides
- Lectures will complement programming component, but intended to cover other parallel computing aspects
- Try to arrange a guest lecture or 2 during quarter

Parallel Programming Lab

- Set up in the IPCC classroom
 - Daniel Ellsworth and David Poliakoff leading the lab
- Shared memory parallel programming (everyone)
 - Cilk Plus (<http://www.cilkplus.org/>)
 - ◆ extension to the C and C++ languages to support data and task parallelism
 - Thread Building Blocks (TBB) (
<https://www.threadingbuildingblocks.org/>)
 - ◆ C++ template library for task parallelism
 - OpenMP (<http://openmp.org/wp/>)
 - ◆ C/C++ and Fortran directive-based parallelism
- Distributed memory message passing (graduate)
 - MPI (http://en.wikipedia.org/wiki/Message_Passing_Interface)
 - ◆ library for message communication on scalable parallel systems

WOPR (What Operational Parallel Resource)

- WOPR was built from whole cloth
 - Constructed by UO graduate students
- Built Next Unit of Computing (NUC) cluster with Intel funds
 - 16x Intel NUC
 - ◆ Haswell i5 CPU (2 cores, hyperthreading)
 - ◆ Intel HD 4000 GPU (OpenCL programmable)
 - ◆ 1 GigE, 16 GB memory, 240 GB mSATA
 - ◆ 16x Logitech keyboard and mouse
 - 16x ViewSonic 22" monitor
 - Dell Edge GigE switch
 - Dell head node



Other Parallel Resources – Mist Cluster

- Distributed memory cluster
- 16 8-core nodes
 - 2x quad-core Pentium Xeon (2.33 GHz)
 - 16 Gbyte memory
 - 160 Gbyte disk
- Dual Gigabit ethernet adaptors
- Master node (same specs)
- Gigabit ethernet switch
- *mist.nic.uoregon.edu*



Other Parallel Resources – ACISS Cluster

- *Applied Computational Instrument for Scientific Synthesis*
 - NSF MRI R² award (2010)
- *Basic nodes* (1,536 total cores)
 - 128 ProLiant SL390 G7
 - Two Intel X5650 2.66 GHz 6-core CPUs per node
 - 72GB DDR3 RAM per basic node
- *Fat nodes* (512 total cores)
 - 16 ProLiant DL 580 G7
 - Four Intel X7560 2.266 GHz 8-core CPUs per node
 - 384GB DDR3 per fat node
- *GPU nodes* (624 total cores, 156 GPUs)
 - 52 ProLiant SL390 G7 nodes, 3 NVidia M2070 GPUs (156 total GPUs)
 - Two Intel X5650 2.66 GHz 6-core CPUs per node (624 total cores)
 - 72GB DDR3 per GPU node
- ACISS has 2672 total cores
- ACISS is located in the UO Computing Center



Course Assignments

- Homework
 - Exercises primarily to prepare for midterm
- Parallel programming lab
 - Exercises for parallel programming patterns
 - Program using Cilk Plus, Thread Building Blocks, OpenMP
 - Graduate students will also do assignments with MPI
- Team term project
 - Programming, presentation, paper
 - Graduate students distributed across teams
- Research summary paper (graduate students)
- Midterm exam later in the 7th week of the quarter
- No final exam
 - Team project presentations during final period

Parallel Programming Term Project

- Major programming project for the course
 - Non-trivial parallel application
 - Include performance analysis
 - Use NUC cluster and possibly Mist and ACIIS clusters
- Project teams
 - 5 person teams, 6 teams (depending on enrollment)
 - Will try our best to balance skills
 - Have 1 graduate student per team
- Project dates
 - Proposal due end of 4th week)
 - Project talk during last class
 - Project due at the end of the term
- Need to get system accounts!!!

Term Paper (for graduate students)

- Investigate parallel computing topic of interest
 - More in depth review
 - Individual choice
 - Summary of major points
- Requires minimum of ten references
 - Book and other references has a large bibliography
 - Google Scholar, Keywords: parallel computing
 - NEC CiteSeer Scientific Literature Digital Library
- Paper abstract and references due by 3rd week
- Final term paper due at the end of the term
- Individual work

Grading

□ Undergraduate

- 5% homework
- 10% pattern programming labs
- 20% programming assignments
- 30% midterm exam
- 35% project

□ Graduate

- 15% programming assignments
- 30% midterm exam
- 35% project
- 20% research paper

Overview

- Broad/Old field of computer science concerned with:
 - Architecture, HW/SW systems, languages, programming paradigms, algorithms, and theoretical models
 - Computing in parallel
- Performance is the *raison d'être* for parallelism
 - High-performance computing
 - Drives computational science revolution
- Topics of study
 - Parallel architectures
 - Parallel programming
 - Parallel algorithms
 - Parallel performance models and tools
 - Parallel applications

What will you get out of CIS 410/510?

- In-depth understanding of parallel computer design
- Knowledge of how to program parallel computer systems
- Understanding of pattern-based parallel programming
- Exposure to different forms parallel algorithms
- Practical experience using a parallel cluster
- Background on parallel performance modeling
- Techniques for empirical performance analysis
- Fun and new friends

Parallel Processing – What is it?

- A *parallel computer* is a computer system that uses multiple processing elements simultaneously in a cooperative manner to solve a computational problem
- *Parallel processing* includes techniques and technologies that make it possible to compute in parallel
 - Hardware, networks, operating systems, parallel libraries, languages, compilers, algorithms, tools, ...
- Parallel computing is an evolution of serial computing
 - Parallelism is natural
 - Computing problems differ in level / type of parallelism
- Parallelism is all about performance! Really?

Concurrency

- Consider multiple tasks to be executed in a computer
- Tasks are concurrent with respect to each if
 - They *can* execute at the same time (*concurrent execution*)
 - Implies that there are no dependencies between the tasks
- Dependencies
 - If a task requires results produced by other tasks in order to execute correctly, the task's execution is *dependent*
 - If two tasks are dependent, they are not concurrent
 - Some form of synchronization must be used to enforce (satisfy) dependencies
- Concurrency is fundamental to computer science
 - Operating systems, databases, networking, ...

Concurrency and Parallelism

- Concurrent is not the same as parallel! Why?
- Parallel execution
 - Concurrent tasks *actually* execute at the same time
 - Multiple (processing) resources have to be available
- **Parallelism = concurrency + “parallel” hardware**
 - Both are required
 - Find concurrent execution opportunities
 - Develop application to execute in parallel
 - Run application on parallel hardware
- Is a parallel application a concurrent application?
- Is a parallel application run with one processor parallel? Why or why not?

Parallelism

- There are granularities of parallelism (parallel execution) in programs
 - Processes, threads, routines, statements, instructions, ...
 - Think about what are the software elements that execute concurrently
- These must be supported by hardware resources
 - Processors, cores, ... (execution of instructions)
 - Memory, DMA, networks, ... (other associated operations)
 - All aspects of computer architecture offer opportunities for parallel hardware execution
- Concurrency is a necessary condition for parallelism
 - Where can you find concurrency?
 - How is concurrency expressed to exploit parallel systems?

Why use parallel processing?

- Two primary reasons (both performance related)
 - Faster time to solution (response time)
 - Solve bigger computing problems (in same time)
- Other factors motivate parallel processing
 - Effective use of machine resources
 - Cost efficiencies
 - Overcoming memory constraints
- Serial machines have inherent limitations
 - Processor speed, memory bottlenecks, ...
- Parallelism has become the future of computing
- Performance is still the driving concern
- **Parallelism = concurrency + parallel HW + performance**

Perspectives on Parallel Processing

- Parallel computer architecture
 - Hardware needed for parallel execution?
 - Computer system design
- (Parallel) Operating system
 - How to manage systems aspects in a parallel computer
- Parallel programming
 - Libraries (low-level, high-level)
 - Languages
 - Software development environments
- Parallel algorithms
- Parallel performance evaluation
- Parallel tools
 - Performance, analytics, visualization, ...

Why study parallel computing today?

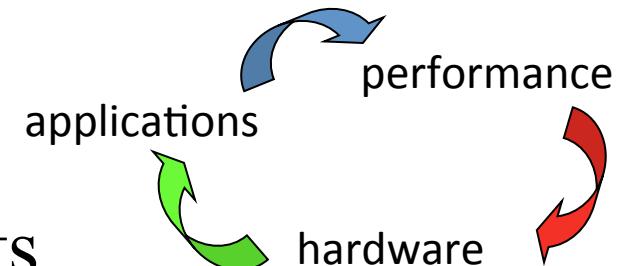
- Computing architecture
 - Innovations often drive to novel programming models
- Technological convergence
 - The “killer micro” is ubiquitous
 - Laptops and supercomputers are fundamentally similar!
 - Trends cause diverse approaches to converge
- Technological trends make parallel computing inevitable
 - Multi-core processors are here to stay!
 - Practically every computing system is operating in parallel
- Understand fundamental principles and design tradeoffs
 - Programming, systems support, communication, memory, ...
 - Performance
- Parallelism is the future of computing

Inevitability of Parallel Computing

- Application demands
 - Insatiable need for computing cycles
- Technology trends
 - Processor and memory
- Architecture trends
- Economics
- Current trends:
 - Today's microprocessors have multiprocessor support
 - Servers and workstations available as multiprocessors
 - Tomorrow's microprocessors are multiprocessors
 - Multi-core is here to stay and #cores/processor is growing
 - Accelerators (GPUs, gaming systems)

Application Characteristics

- Application performance demands hardware advances
- Hardware advances generate new applications
- New applications have greater performance demands
 - Exponential increase in microprocessor performance
 - Innovations in parallel architecture and integration
- Range of performance requirements
 - System performance must also improve as a whole
 - Performance requirements require computer engineering
 - Costs addressed through technology advancements



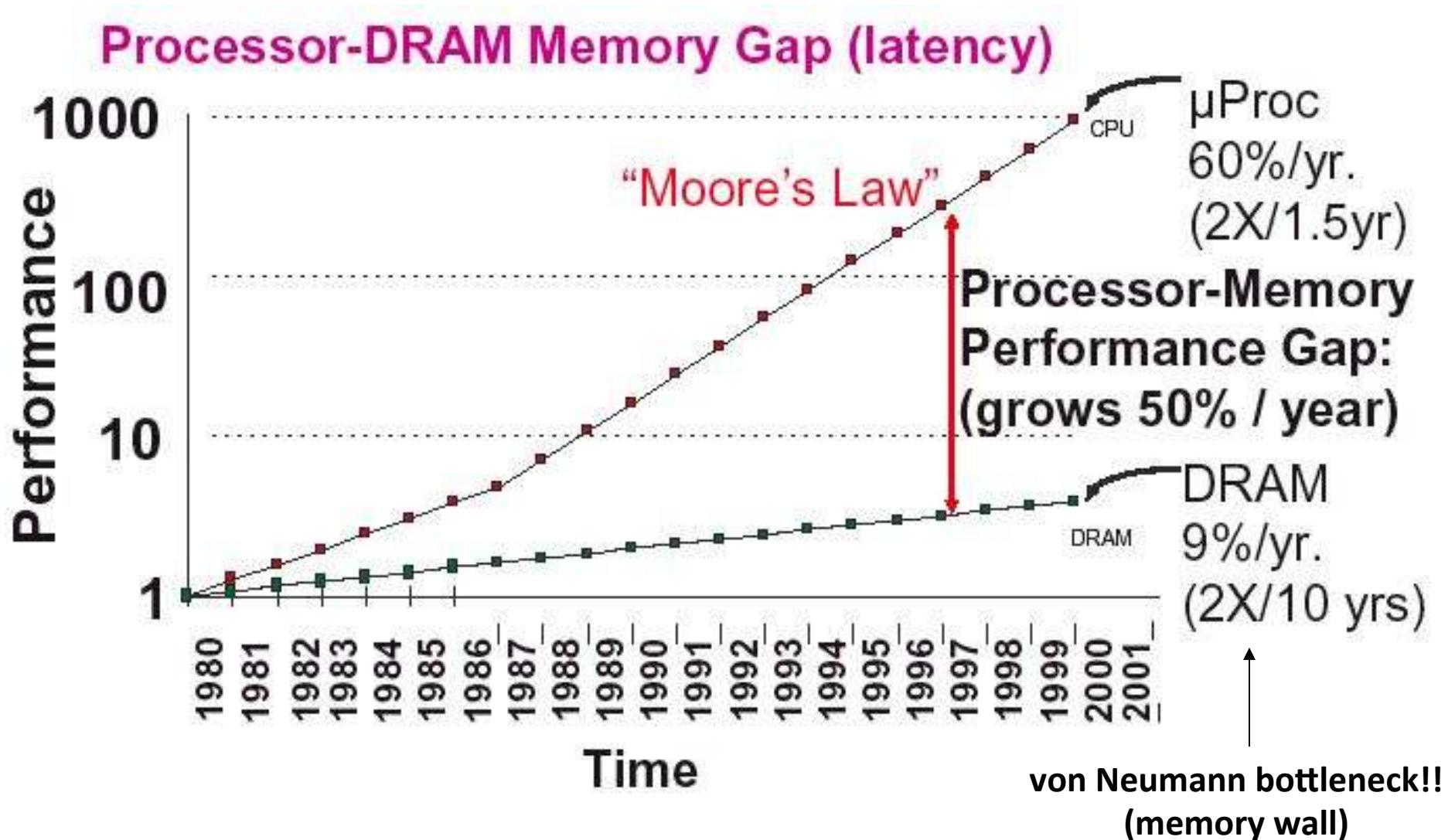
Broad Parallel Architecture Issues

- Resource allocation
 - How many processing elements?
 - How powerful are the elements?
 - How much memory?
- Data access, communication, and synchronization
 - How do the elements cooperate and communicate?
 - How are data transmitted between processors?
 - What are the abstractions and primitives for cooperation?
- Performance and scalability
 - How does it all translate into performance?
 - How does it scale?

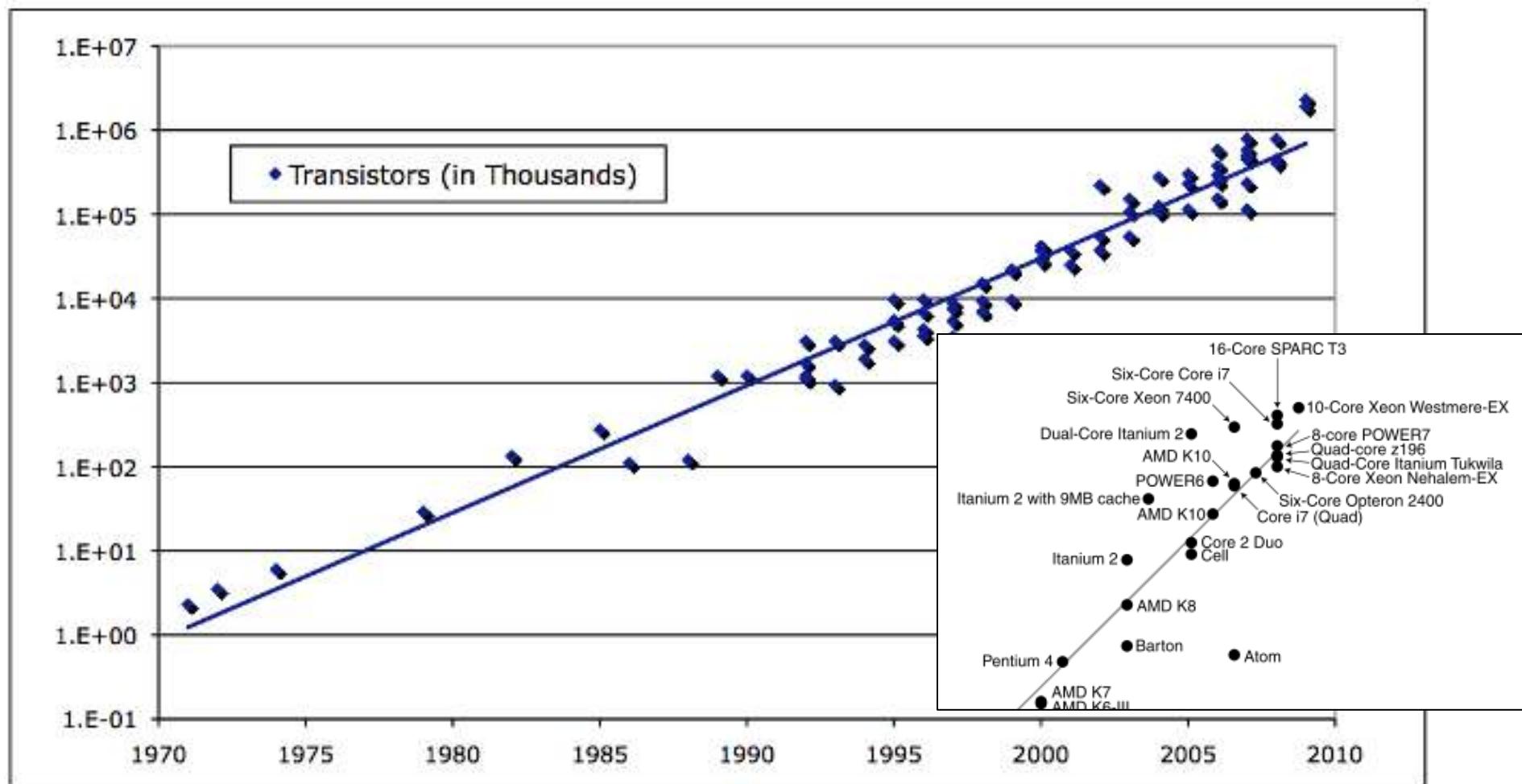
Leveraging Moore's Law

- More transistors = more parallelism opportunities
- Microprocessors
 - Implicit parallelism
 - ◆ pipelining
 - ◆ multiple functional units
 - ◆ superscalar
 - Explicit parallelism
 - ◆ SIMD instructions
 - ◆ long instruction words

What's Driving Parallel Computing Architecture?



Microprocessor Transistor Counts (1971-2011)



Data from Kunle Olukotun, Lance Hammond, Herb Sutter,
Burton Smith, Chris Batten, and Krste Asanović
Slide from Kathy Yellick

What has happened in the last several years?

- Processing chip manufacturers increased processor performance by increasing CPU clock frequency
 - Riding Moore's law
- Until the chips got too hot!
 - Greater clock frequency \Rightarrow greater electrical power
 - Pentium 4 heat sink ○ Frying an egg on a Pentium 4



- Add multiple cores to add performance
 - Keep clock frequency same or reduced
 - Keep lid on power requirements

Power Density Growth

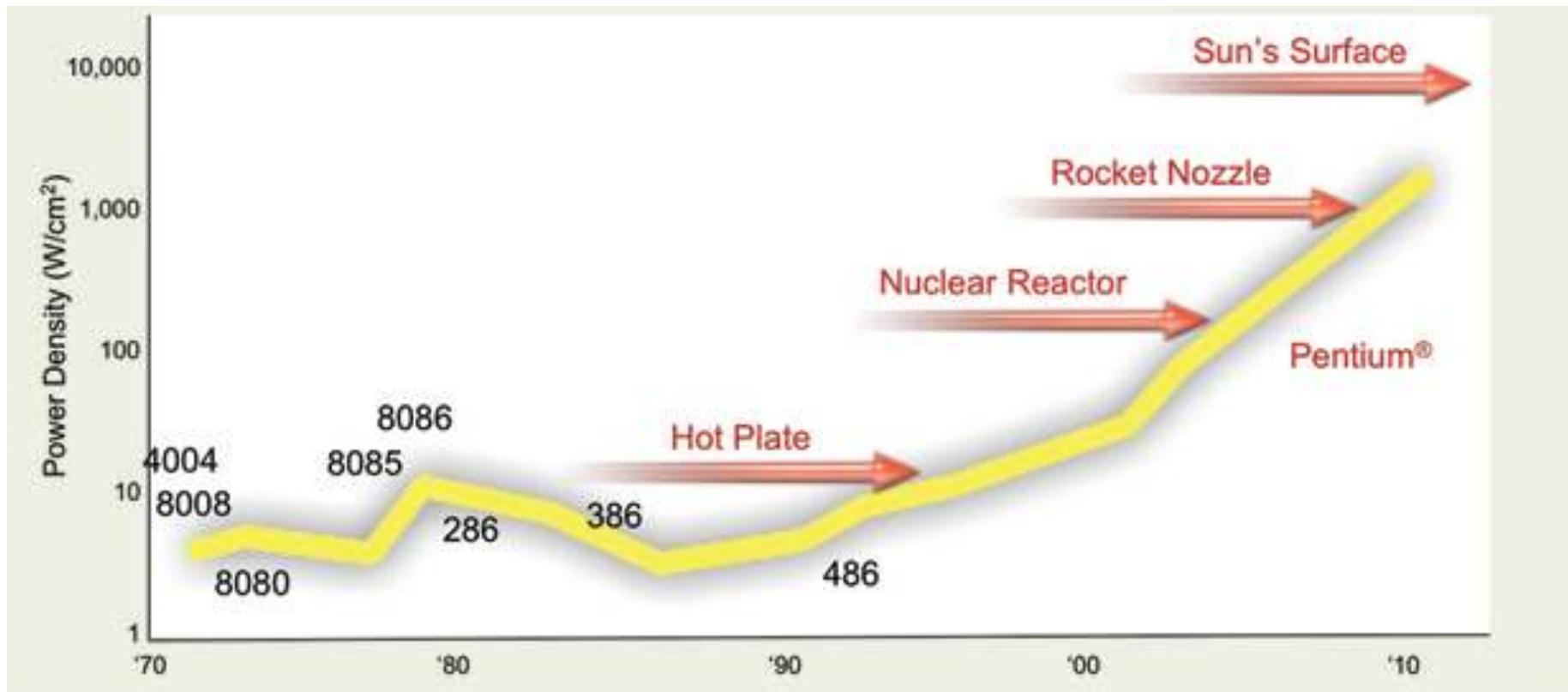
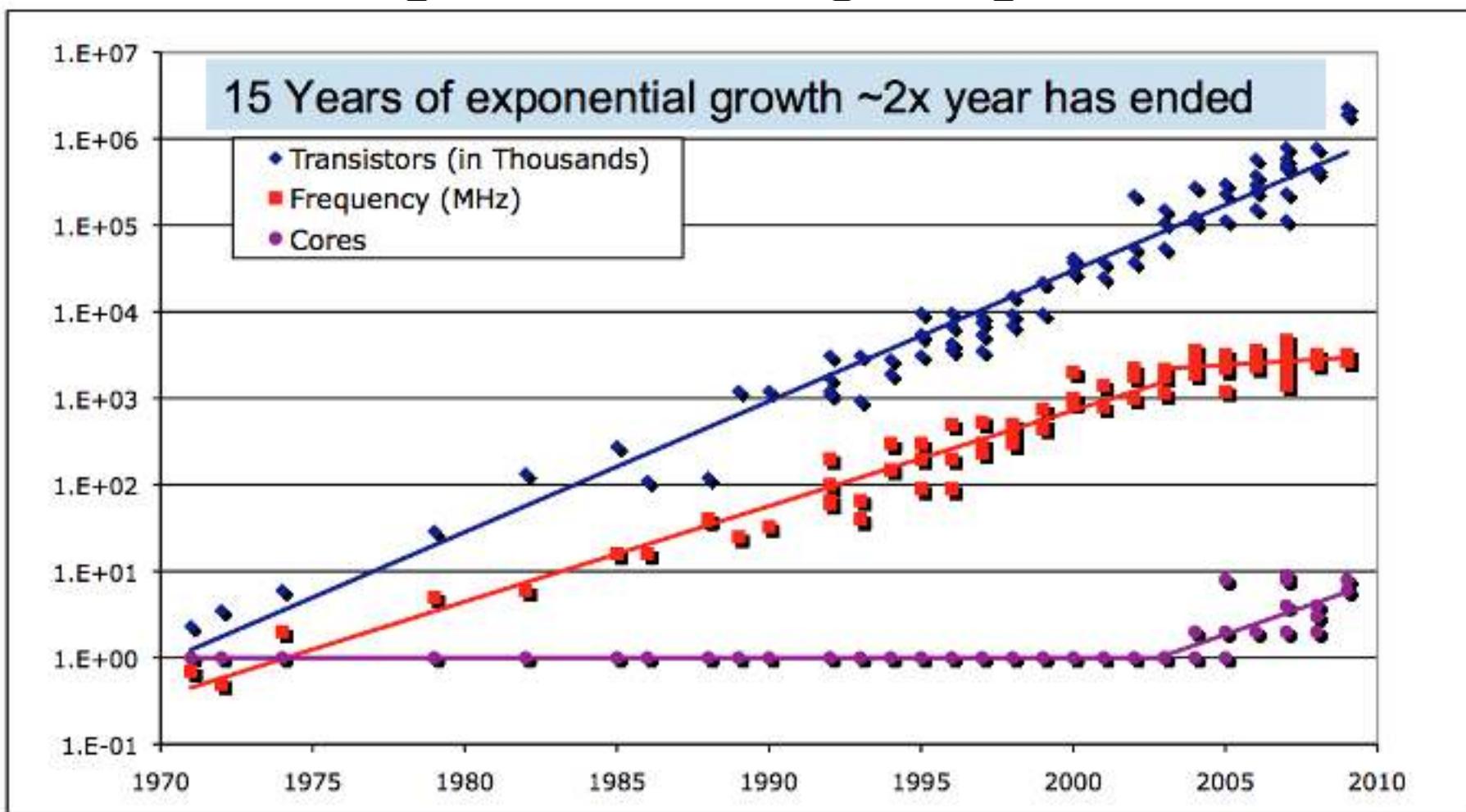


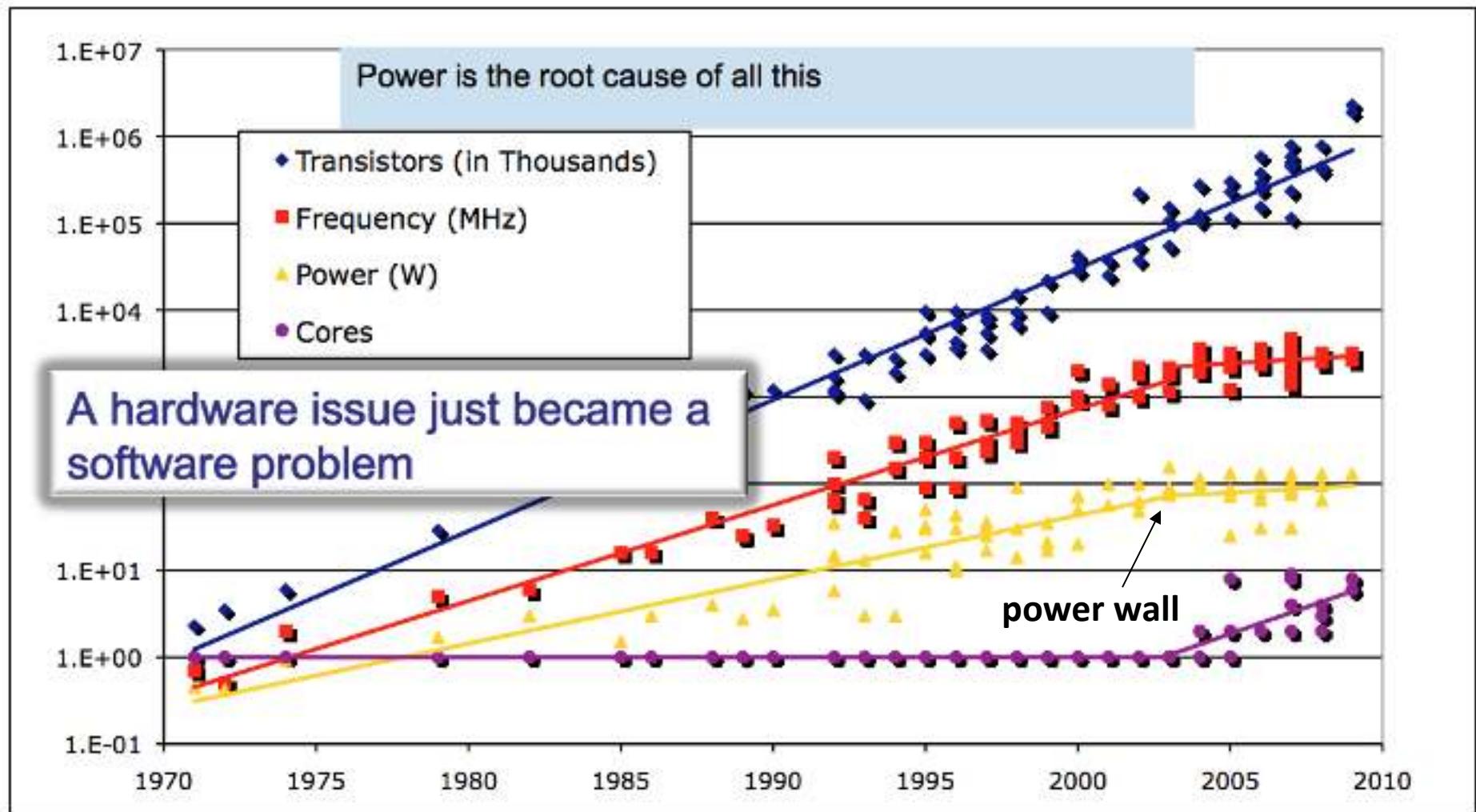
Figure courtesy of Pat Gelsinger, Intel Developer Forum, Spring 2004

What's Driving Parallel Computing Architecture?



Data from Kunle Olukotun, Lance Hammond, Herb Sutter,
Burton Smith, Chris Batten, and Krste Asanović
Slide from Kathy Yellick

What's Driving Parallel Computing Architecture?



Classifying Parallel Systems – Flynn’s Taxonomy

- Distinguishes multi-processor computer architectures along the two independent dimensions
 - *Instruction* and *Data*
 - Each dimension can have one state: *Single* or *Multiple*
- SISD: Single Instruction, Single Data
 - Serial (non-parallel) machine
- SIMD: Single Instruction, Multiple Data
 - Processor arrays and vector machines
- MISD: Multiple Instruction, Single Data (weird)
- MIMD: Multiple Instruction, Multiple Data
 - Most common parallel computer systems

Parallel Architecture Types

- Instruction-Level Parallelism
 - Parallelism captured in instruction processing
- Vector processors
 - Operations on multiple data stored in vector registers
- Shared-memory Multiprocessor (SMP)
 - Multiple processors sharing memory
 - Symmetric Multiprocessor (SMP)
- Multicomputer
 - Multiple computer connect via network
 - Distributed-memory cluster
- Massively Parallel Processor (MPP)

Phases of Supercomputing (Parallel) Architecture

- Phase 1 (1950s): sequential instruction execution
- Phase 2 (1960s): sequential instruction issue
 - Pipeline execution, reservations stations
 - Instruction Level Parallelism (ILP)
- Phase 3 (1970s): vector processors
 - Pipelined arithmetic units
 - Registers, multi-bank (parallel) memory systems
- Phase 4 (1980s): SIMD and SMPs
- Phase 5 (1990s): MPPs and clusters
 - Communicating sequential processors
- Phase 6 (>2000): many cores, accelerators, scale, ...

Performance Expectations

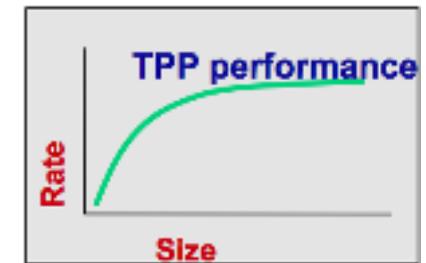
- If each processor is rated at k MFLOPS and there are p processors, we should expect to see $k*p$ MFLOPS performance? Correct?
- If it takes 100 seconds on 1 processor, it should take 10 seconds on 10 processors? Correct?
- Several causes affect performance
 - Each must be understood separately
 - But they interact with each other in complex ways
 - ◆ solution to one problem may create another
 - ◆ one problem may mask another
- Scaling (system, problem size) can change conditions
- Need to understand performance space

Scalability

- A program can scale up to use many processors
 - What does that mean?
- How do you evaluate scalability?
- How do you evaluate scalability goodness?
- Comparative evaluation
 - If double the number of processors, what to expect?
 - Is scalability linear?
- Use parallel efficiency measure
 - Is efficiency retained as problem size increases?
- Apply performance metrics

Top 500 Benchmarking Methodology

- Listing of the world's 500 most powerful computers
- Yardstick for high-performance computing (HPC)
 - Rmax : maximal performance Linpack benchmark
 - ◆ dense linear system of equations ($Ax = b$)
- Data listed
 - Rpeak : theoretical peak performance
 - Nmax : problem size needed to achieve Rmax
 - N^{1/2} : problem size needed to achieve 1/2 of Rmax
 - Manufacturer and computer type
 - Installation site, location, and year
- Updated twice a year at SC and ISC conferences

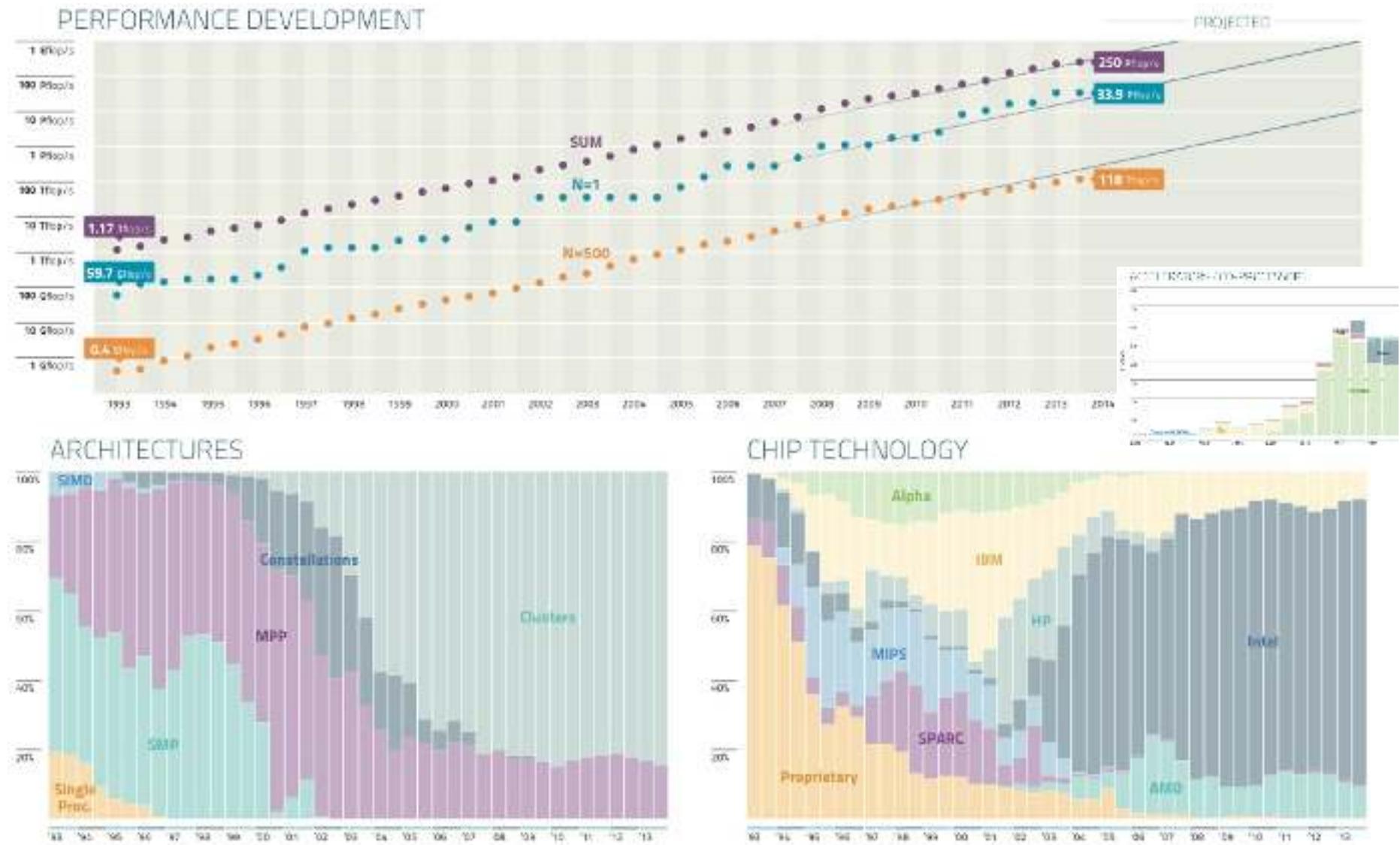


Top 10 (November 2013)

Different architectures

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Super Computer Center in Guangzhou China	Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325
7	Texas Advanced Computing Center/Univ. of Texas United States	Stampede - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462,462	5,168.1	8,520.1	4,510
8	Forschungszentrum Juelich (FZJ) Germany	JUQUEEN - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	458,752	5,008.9	5,872.0	2,301
9	DOE/NNSA/LLNL United States	Vulcan - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	393,216	4,293.3	5,033.2	1,972
10	Leibniz Rechenzentrum Germany	SuperMUC - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR IBM	147,456	2,897.0	3,185.1	3,423

Top 500 – Performance (November 2013)



#1: NUDT Tiahne-2 (*Milkyway-2*)

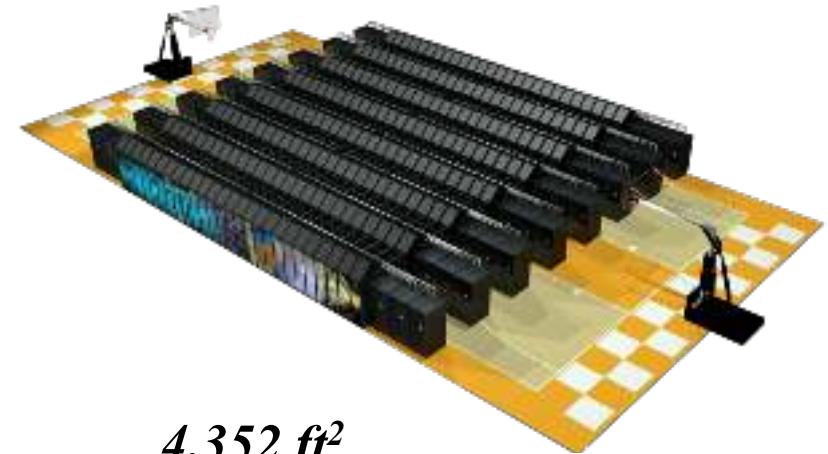
- Compute Nodes have 3.432 Tflop/s per node
 - 16,000 nodes
 - 32000 Intel Xeon CPU
 - 48000 Intel Xeon Phi
- Operations Nodes
 - 4096 FT CPUs
- Proprietary interconnect
 - TH2 express
- 1PB memory
 - Host memory only
- Global shared parallel storage is 12.4 PB
- Cabinets: $125+13+24 = 162$
 - Compute, communication, storage
 - $\sim 750 \text{ m}^2$



#2: ORNL Titan Hybrid System (Cray XK7)



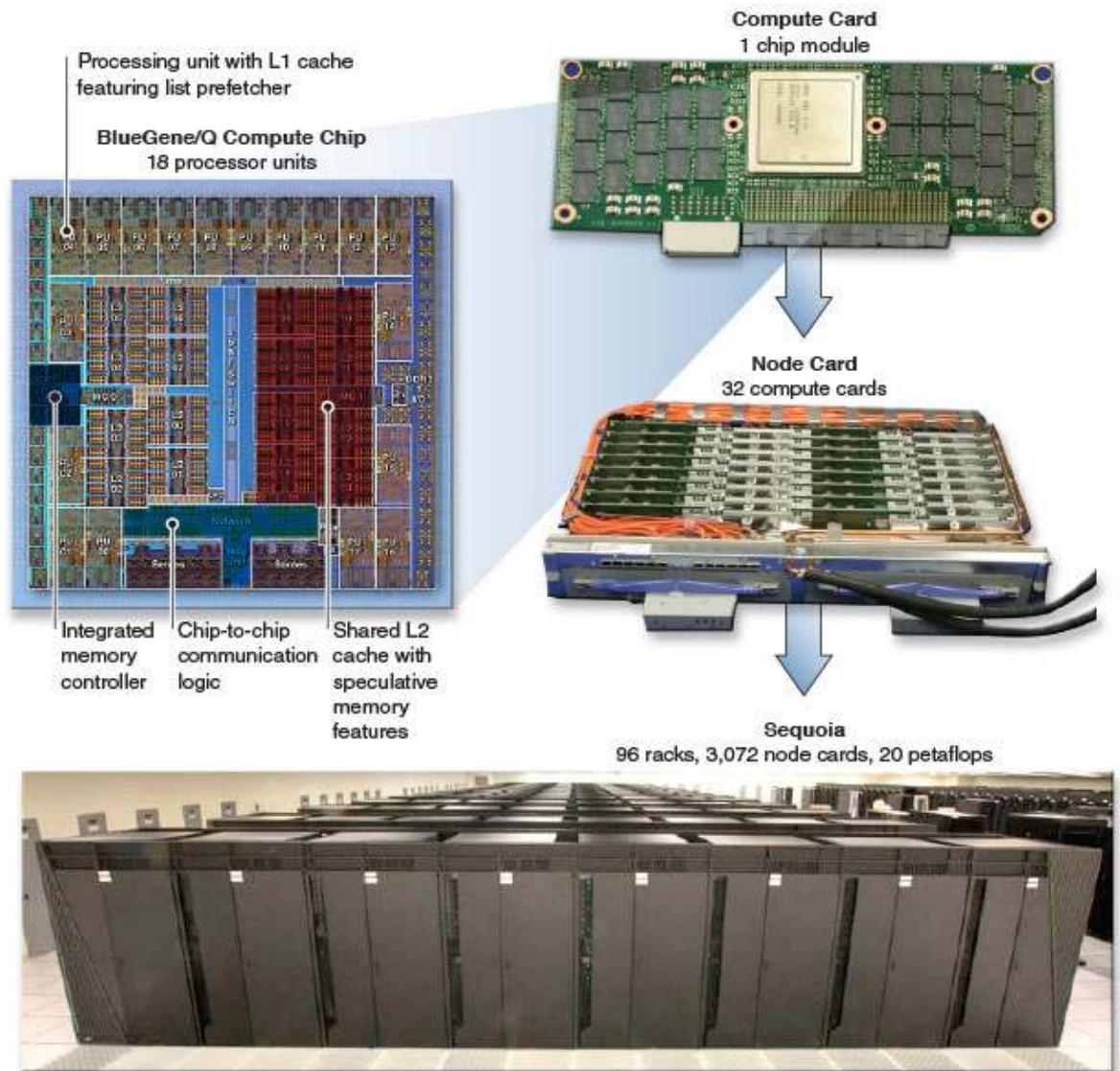
- Peak performance of 27.1 PF
 - 24.5 GPU + 2.6 CPU
- 18,688 Compute Nodes each with:
 - 16-Core AMD Opteron CPU
 - NVIDIA Tesla “K20x” GPU
 - 32 + 6 GB memory
- 512 Service and I/O nodes
- 200 Cabinets
- 710 TB total system memory
- Cray Gemini 3D Torus Interconnect
- 8.9 MW peak power



$4,352 \text{ ft}^2$

#3: LLNL Sequoia (IBM BG/Q)

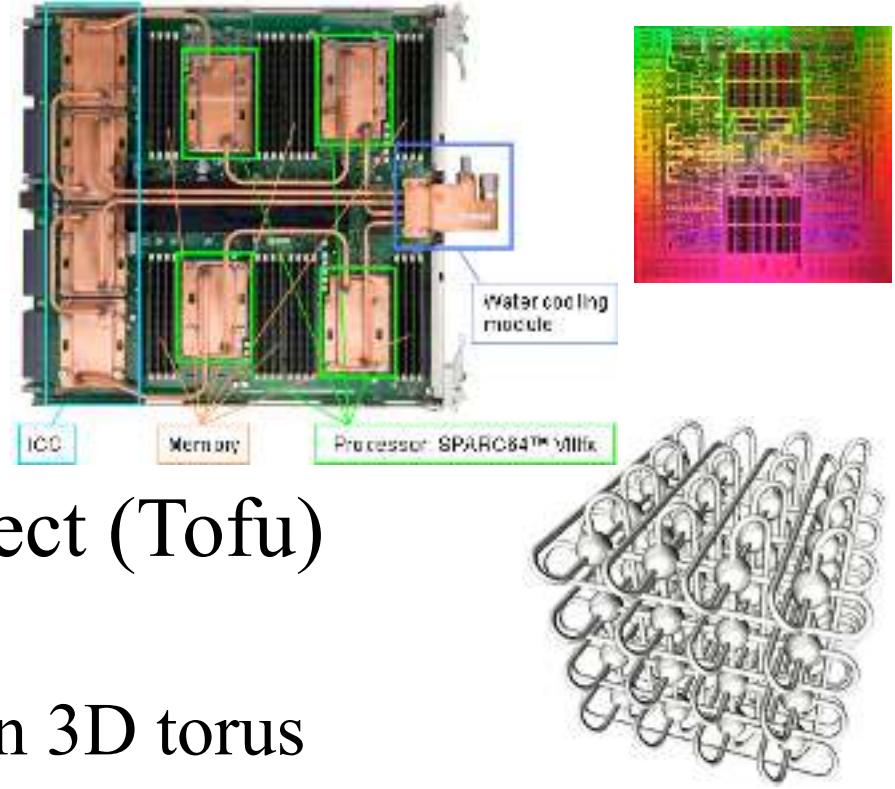
- Compute card
 - 16-core PowerPC A2 processor
 - 16 GB DDR3
- Compute node has 98,304 cards
- Total system size:
 - 1,572,864 processing cores
 - 1.5 PB memory
- 5-dimensional torus interconnection network
- Area of 3,000 ft²



#4: RIKEN K Computer



- 80,000 CPUs
 - SPARC64 VIIIfx
 - 640,000 cores
- 800 water-cooled racks
- 5D mesh/torus interconnect (Tofu)
 - 12 links between node
 - 12x higher scalability than 3D torus



Contemporary HPC Architectures

Date	System	Location	Comp	Comm	Peak (PF)	Power (MW)
2009	Jaguar; Cray XT5	ORNL	AMD 6c	Seastar2	2.3	7.0
2010	Tianhe-1A	NSC Tianjin	Intel + NVIDIA	Proprietary	4.7	4.0
2010	Nebulae	NSCS Shenzhen	Intel + NVIDIA	IB	2.9	2.6
2010	Tsubame 2	TiTech	Intel + NVIDIA	IB	2.4	1.4
2011	K Computer	RIKEN/Kobe	SPARC64 VIIIfx	Tofu	10.5	12.7
2012	Titan; Cray XK6	ORNL	AMD + NVIDIA	Gemini	27	9
2012	Mira; BlueGeneQ	ANL	SoC	Proprietary	10	3.9
2012	Sequoia; BlueGeneQ	LLNL	SoC	Proprietary	20	7.9
2012	Blue Waters; Cray	NCSA/UIUC	AMD + (partial) NVIDIA	Gemini	11.6	
2013	Stampede	TACC	Intel + MIC	IB	9.5	5
2013	Tianhe-2	NSCC-GZ (Guangzhou)	Intel + MIC	Proprietary	54	~20

Top 10 (Top500 List, June 2011)

Rank	Site	Computer	Country	Cores	Rmax [Pflops]	% of Peak
1	RIKEN Advanced Inst for Comp Sci	K Computer Fujitsu SPARC64 VIIIfx + custom	Japan	548,352	8.16	93
2	Nat. SuperComputer Center in Tianjin	Tianhe-1A, NUDT Intel + Nvidia GPU + custom	China	186,368	2.57	55
3	DOE / OS Oak Ridge Nat Lab	Jaguar, Cray AMD + custom	USA	224,162	1.76	75
4	Nat. Supercomputer Center in Shenzhen	Nebulae, Dawning Intel + Nvidia GPU + IB	China	120,640	1.27	43
5	GSIC Center, Tokyo Institute of Technology	Tsubame 2.0, HP Intel + Nvidia GPU + IB	Japan	73,278	1.19	52
6	DOE / NNSA LANL & SNL	Cielo, Cray AMD + custom	USA	142,272	1.11	81
7	NASA Ames Research Center/NAS	Pleiades SGI Altix ICE 8200EX/8400EX + IB	USA	111,104	1.09	83
8	DOE / OS Lawrence Berkeley Nat Lab	Hopper, Cray AMD + custom	USA	153,408	1.054	82
9	Commissariat à l'Energie Atomique (CEA)	Tera-10, Bull Intel + IB	France	138,368	1.050	84
10	DOE / NNSA Los Alamos Nat Lab	Roadrunner, IBM AMD + Cell GPU + IB	USA	122,400	1.04	76

Figure credit: <http://www.netlib.org/utk/people/JackDongarra/SLIDES/korea-2011.pdf>

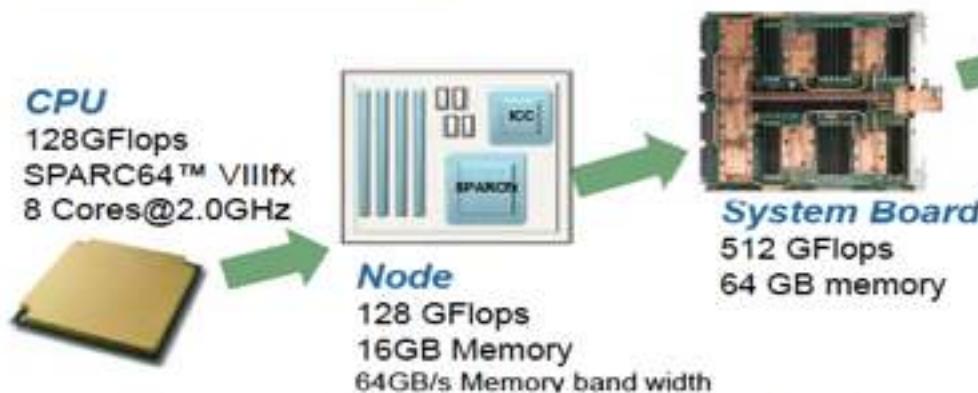
Japanese K Computer (#1 in June 2011)

K computer Specifications



CPU (SPARC64 VIIIfx)	Cores/Node	8 cores (@2GHz)
	Performance	128GFlops
	Architecture	SPARC V9 + HPC extension
	Cache	L1(I/D) Cache : 32KB/32KB L2 Cache : 6MB
	Power	58W (typ: 30 C)
	Mem. bandwidth	64GB/s.
Node	Configuration	1 CPU / Node
	Memory capacity	16GB (2GB/core)
System board(SB)	No. of nodes	4 nodes /SB
Rack	No. of SB	24 SBs/rack
System	Nodes/system	> 80,000

Inter-connect	Topology	6D Mesh/Torus
	Performance	5GB/s. for each link
	No. of link	10 links/ node
	Additional feature	H/W barrier, reduction
	Architecture	Routing chip structure (no outside switch box)
Cooling	CPU, ICC*	Direct water cooling
	Other parts	Air cooling



* ICC : Interconnect Chip

New Linpack run with 705,024 cores at 10.51 Pflop/s (88,128 CPUs)

Top 500 Top 10 (2006)

	Manufacturer	Computer	Rmax [TF/s]	Installation Site	Country	Year	#Proc
1	IBM	BlueGene/L eServer Blue Gene	280.6	DOE/NNSA/LLNL	USA	2005	131,072
2	Sandia/Cray	Red Storm Cray XT3	101.4	NNSA/Sandia	USA	2006	26,544
3	IBM	BGW eServer Blue Gene	91.29	IBM Thomas Watson	USA	2005	40,960
4	IBM	ASC Purple eServer pSeries p575	75.76	DOE/NNSA/LLNL	USA	2005	12,208
5	IBM	MareNostrum JS21 Cluster, Myrinet	62.63	Barcelona Supercomputing Center	Spain	2006	12,240
6	Dell	Thunderbird PowerEdge 1850, IB	53.00	NNSA/Sandia	USA	2005	9,024
7	Bull	Tera-10 NovaScale 5160, Quadrics	52.84	CEA	France	2006	9,968
8	SGI	Columbia Altix, Infiniband	51.87	NASA Ames	USA	2004	10,160
9	NEC/Sun	Tsubame Fire x4600, ClearSpeed, IB	47.38	GSIC / Tokyo Institute of Technology	Japan	2006	11,088
10	Cray	Jaguar Cray XT3	43.48	ORNL	USA	2006	10,424

Top 500 Linpack Benchmark List (June 2002)

Computer (Full Precision)		Number of Processors	R_{max} Gflop/s	N_{max} order	$N_{1/2}$ order	R_{peak} Gflop/s
★Earth Simulator, NEC processors****	esc	5104	35610	1041216	265408	40832
ASCI White-Pacific, IBM SP Power 3(375 MHz)	llnl	8000	7226	518096	179000	12000
★Compaq AlphaServer SC ES45/EV68 1GHz	psc	3016	4463	280000	85000	6032
Compaq AlphaServer SC ES45/EV68 1GHz	psc	3024	4059	525000	105000	6048
★Compaq AlphaServer SC ES45/EV68 1GHz	cea	2560	3980	360000	85000	5120
IBM SP Power3 208 nodes 375 MHz	llnl	3328	3052.	371712		4992
★Compaq Alphaserver SC ES45/EV68 1GHz	lanl	2048	2916	272000		4096
★IBM SP Power3 158 nodes 375 MHz	llnl	2528	2526.	371712	102400	3792
ASCI Red Intel Pentium II Xeon core 333MHz	snl	9632	2379.6	362880	75400	3207
ASCI Blue-Pacific SST, IBM SP 604E(332 MHz)	llnl	5808	2144.	431344	432344	3868
ASCI Red Intel Pentium II Xeon core 333MHz	snl	9472	2121.3	251904	66000	3154
Compaq Alphaserver SC ES45/EV68 1GHz	lanl	1520	2096	390000	71000	3040
★IBM SP 112 nodes (375 MHz POWER3 High)	ibm	1792	1791	275000	275000	2688
HITACHI SR8000/MPP/1152(450MHz)	u toyko	1152	1709.1	141000	16000	2074
★HITACHI SR8000-F1/168(375MHz)	leibniz	168	1653.	160000	19560	2016
ASCI Red Intel Pentium II Xeon core 333Mhz	snl	6720	1633.3	306720	52500	2238
SGI ASCI Blue Mountain	lanl	5040	1608.	374400	138000	2520
IBM SP 328 nodes (375 MHz POWER3 Thin)	noo	1312	1417.	374000	374000	1968
Intel ASCI Option Red (200 MHz Pentium Pro)	snl	9152	1338.	235000	63000	1830
NEC SX-5/128M8(3.2ns)	osaka	128	1192.0	129536	10240	1280
CRAY T3E-1200 (600 MHz)	us government	1488	1127.	148800	28272	1786
HITACHI SR8000-F1/112(375MHz)	leibniz	112	1035.0	120000	15160	1344

Japanese Earth Simulator

- World's fastest supercomputer!!! (2002)
 - 640 NEC SX-6 nodes
 - ◆ 8 vector processors
 - 5104 total processors
 - Single stage crossbar
 - ◆ ~2900 meters of cables
 - 10 TB memory
 - 700 TB disk space
 - 1.6 PB mass storage
 - 40 Tflops peak performance
 - 35.6 Tflops Linpack performance



Prof. Malony and colleagues at Japanese ES



Performance Development in Top 500

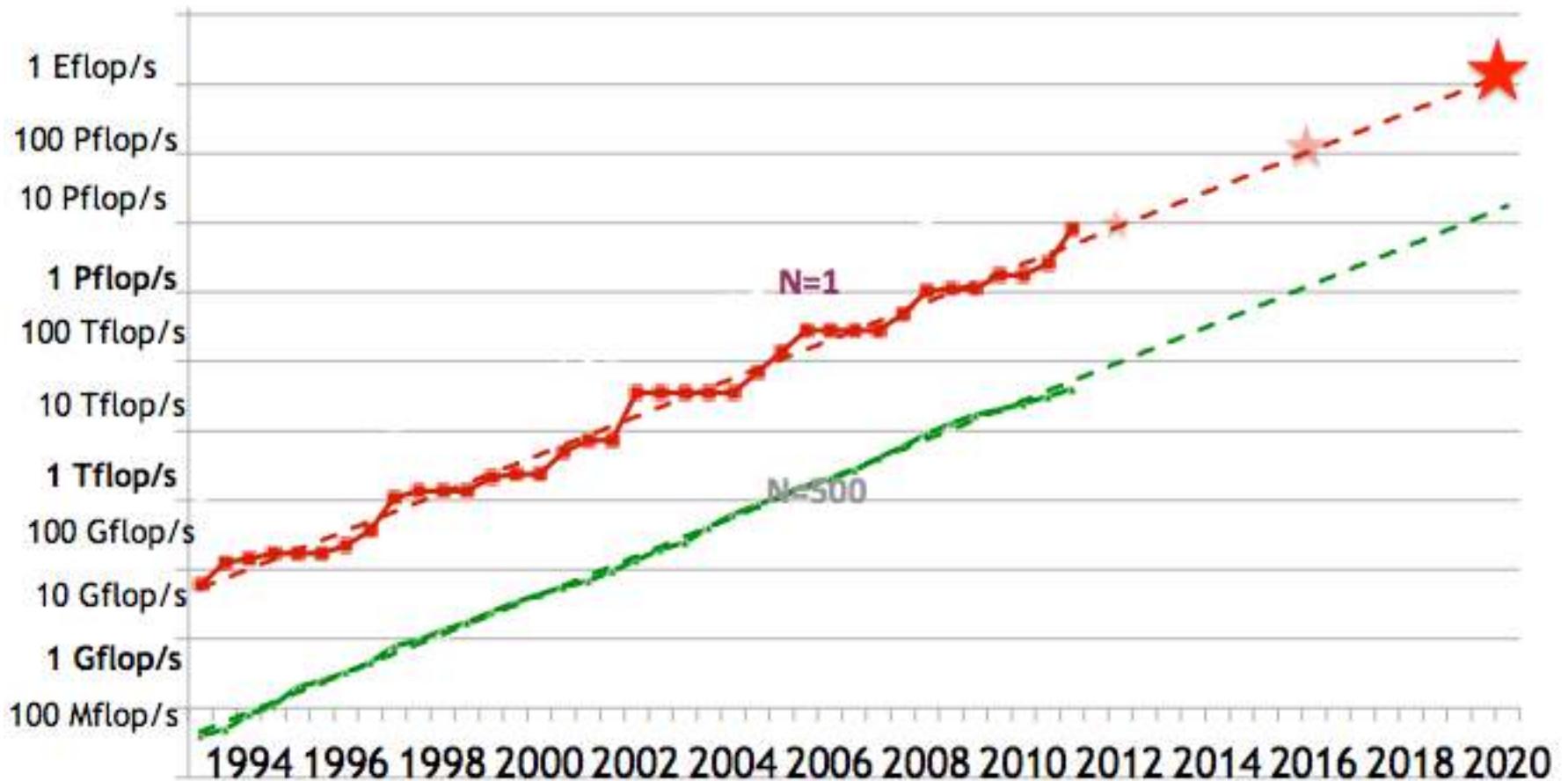


Figure credit: <http://www.netlib.org/utk/people/JackDongarra/SLIDES/korea-2011.pdf>

Exascale Initiative

- Exascale machines are targeted for 2019
- What are the potential differences and problems?

Systems	2011 K Computer	2019	Difference Today & 2019
System peak	8.7 Pflop/s	1 Eflop/s	$O(100)$
Power	10 MW	~20 MW	???
System memory	1.6 PB	32 - 64 PB	$O(10)$
Node performance	128 GF	1,2 or 15TF	$O(10) - O(100)$
Node memory BW	64 GB/s	2 - 4TB/s	$O(100)$
Node concurrency	8	$O(1k)$ or 10k	$O(100) - O(1000)$
Total Node Interconnect BW	20 GB/s	200-400GB/s	$O(10)$
System size (nodes)	68,544	$O(100,000)$ or $O(1M)$	$O(10) - O(100)$
Total concurrency	548,352	$O(billion)$	$O(1,000)$
MTTI	days	$O(1 day)$	- $O(10)$

Major Changes to Software and Algorithms

- What were we concerned about before and now?
- Must rethink the design for exascale
 - Data movement is expensive (Why?)
 - Flops per second are cheap (Why?)
- Need to reduce communication and synchronization
- Need to develop fault-resilient algorithms
- How do we deal with massive parallelism?
- Software must adapt to the hardware (autotuning)

Supercomputing and Computational Science

- By definition, a supercomputer is of a class of computer systems that are the most powerful computing platforms at that time
- Computational science has always lived at the leading (and bleeding) edge of supercomputing technology
- “Most powerful” depends on performance criteria
 - Performance metrics related to computational algorithms
 - Benchmark “real” application codes
- Where does the performance come from?
 - More powerful processors
 - More processors (cores)
 - Better algorithms

Computational Science

- Traditional scientific methodology
 - Theoretical science
 - ◆ Formal systems and theoretical models
 - ◆ Insight through abstraction, reasoning through proofs
 - Experimental science
 - ◆ Real system and empirical models
 - ◆ Insight from observation, reasoning from experiment design
- Computational science
 - Emerging as a principal means of scientific research
 - Use of computational methods to model scientific problems
 - ◆ Numerical analysis plus simulation methods
 - ◆ Computer science tools
 - Study and application of these solution techniques

Computational Challenges

- Computational science thrives on computer power
 - Faster solutions
 - Finer resolution
 - Bigger problems
 - Improved interaction
 - BETTER SCIENCE!!!
- How to get more computer power?
 - Scalable parallel computing
- Computational science also thrives better integration
 - Couple computational resources
 - Grid computing

Scalable Parallel Computing

- Scalability in parallel architecture
 - Processor numbers
 - Memory architecture
 - Interconnection network
 - Avoid critical architecture bottlenecks
- Scalability in computational problem
 - Problem size
 - Computational algorithms
 - ◆ Computation to memory access ratio
 - ◆ Computation to communication ration
- Parallel programming models and tools
- Performance scalability

Next Lectures

- Parallel computer architectures
- Parallel performance models



Parallel Computer Architecture

Introduction to Parallel Computing

CIS 410/510

Department of Computer and Information Science



UNIVERSITY OF OREGON

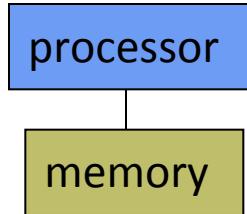
Outline

- Parallel architecture types
- Instruction-level parallelism
- Vector processing
- SIMD
- Shared memory
 - Memory organization: UMA, NUMA
 - Coherency: CC-UMA, CC-NUMA
- Interconnection networks
- Distributed memory
- Clusters
- Clusters of SMPs
- Heterogeneous clusters of SMPs

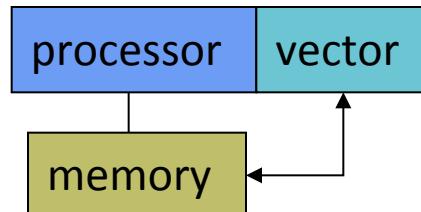
Parallel Architecture Types

- Uniprocessor

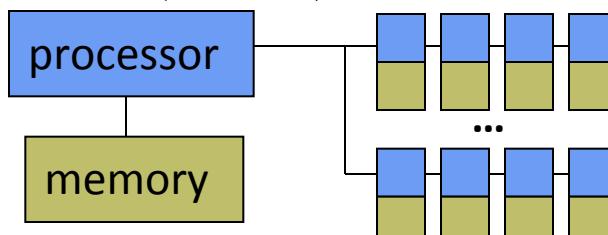
- Scalar processor



- Vector processor



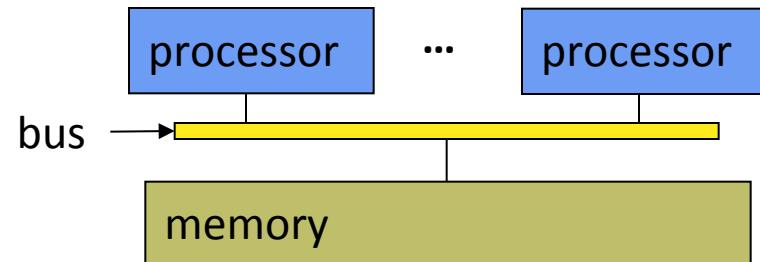
- Single Instruction Multiple Data (SIMD)



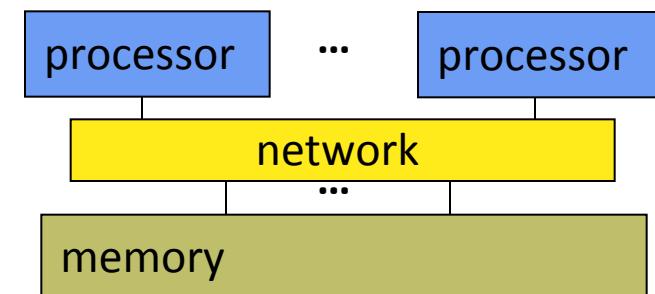
- Shared Memory Multiprocessor (SMP)

- Shared memory address space

- Bus-based memory system



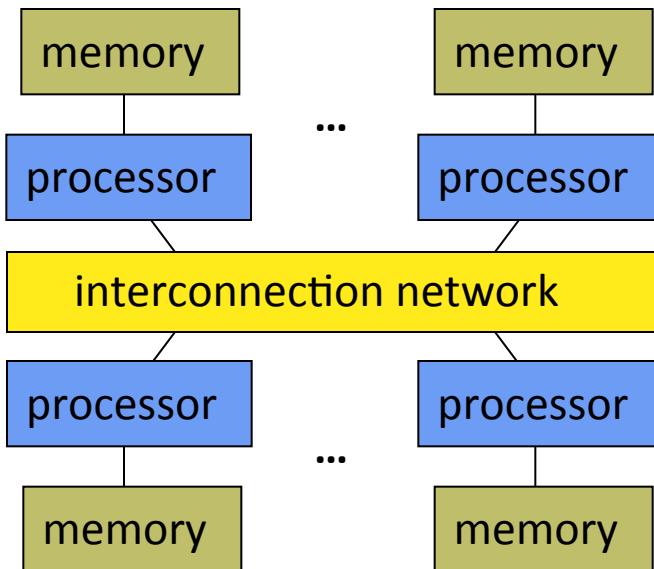
- Interconnection network



Parallel Architecture Types (2)

- Distributed Memory Multiprocessor

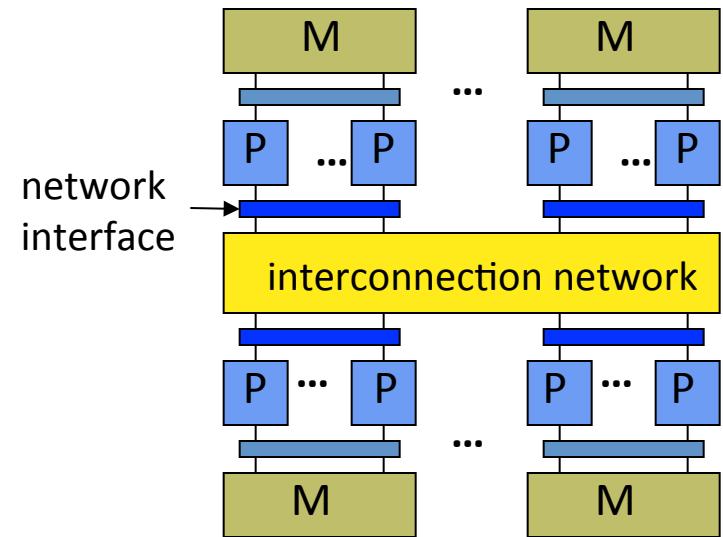
- Message passing between nodes



- Massively Parallel Processor (MPP)
 - Many, many processors

- Cluster of SMPs

- Shared memory addressing within SMP node
 - Message passing between SMP nodes

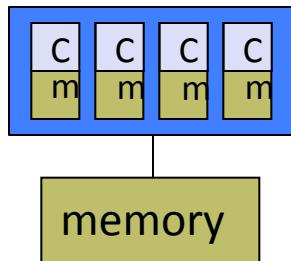


- Can also be regarded as MPP if processor number is large

Parallel Architecture Types (3)

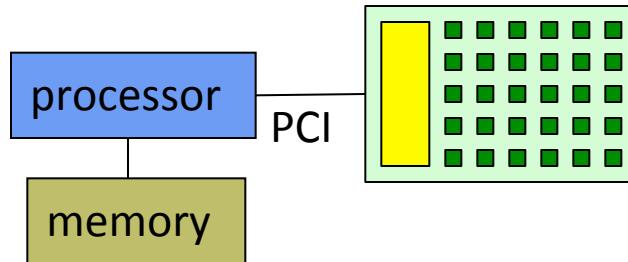
- Multicore

- Multicore processor

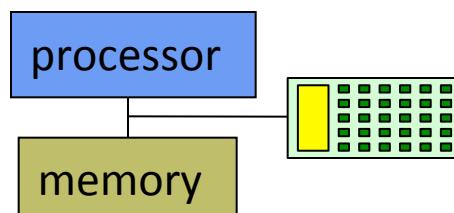


cores can be
hardware
multithreaded
(hyperthread)

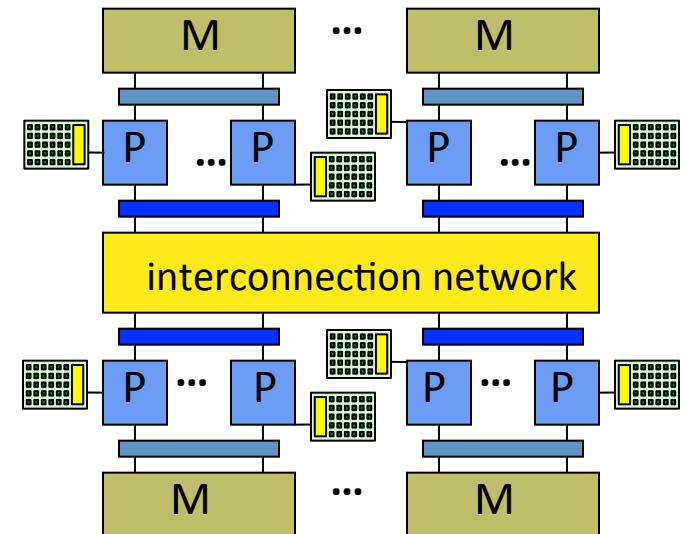
- GPU accelerator



- “Fused” processor accelerator



- Multicore SMP+GPU Cluster
 - Shared memory addressing within SMP node
 - Message passing between SMP nodes
 - GPU accelerators attached



How do you get parallelism in the hardware?

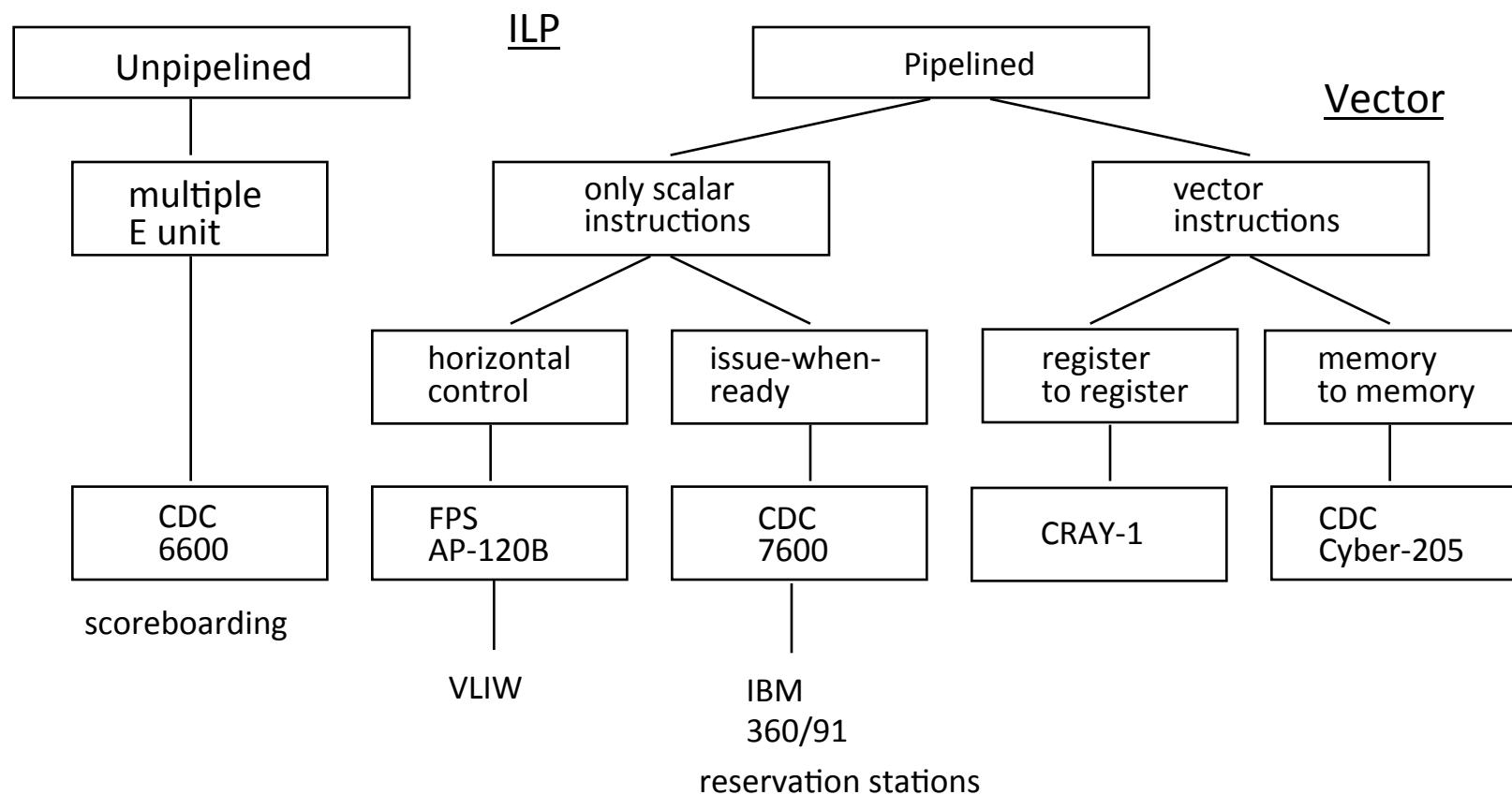
- Instruction-Level Parallelism (ILP)
- Data parallelism
 - Increase amount of data to be operated on at same time
- Processor parallelism
 - Increase number of processors
- Memory system parallelism
 - Increase number of memory units
 - Increase bandwidth to memory
- Communication parallelism
 - Increase amount of interconnection between elements
 - Increase communication bandwidth

Instruction-Level Parallelism

- Opportunities for splitting up instruction processing
- Pipelining within instruction
- Pipelining between instructions
- Overlapped execution
- Multiple functional units
- Out of order execution
- Multi-issue execution
- Superscalar processing
- Superpipelining
- Very Long Instruction Word (VLIW)
- Hardware multithreading (hyperthreading)

Parallelism in Single Processor Computers

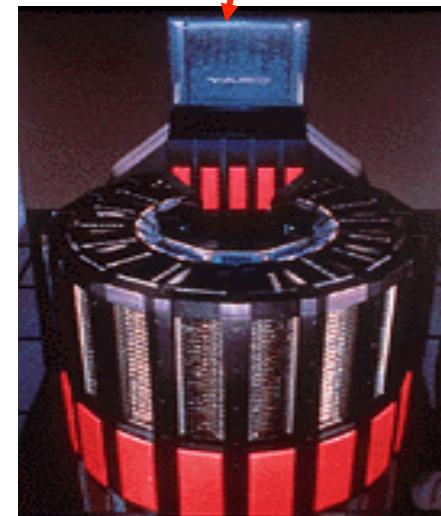
□ History of processor architecture innovation



Vector Processing

- Scalar processing
 - Processor instructions operate on scalar values
 - integer registers and floating point registers
- Vectors
 - Set of scalar data
 - Vector registers
 - ◆ integer, floating point (typically)
 - Vector instructions operate on vector registers (SIMD)
- Vector unit pipelining
- Multiple vector units
- Vector chaining

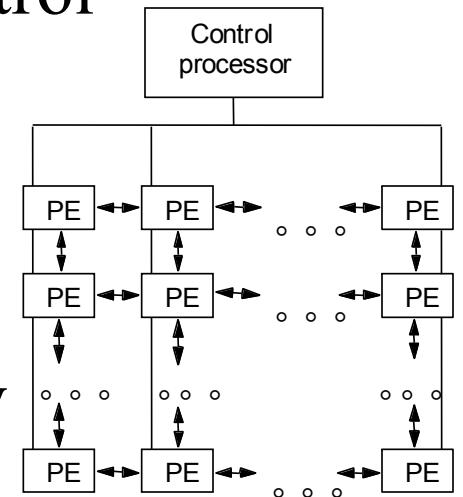
Liquid-cooled with inert fluorocarbon. (That's a waterfall fountain!!!)



Data Parallel Architectures

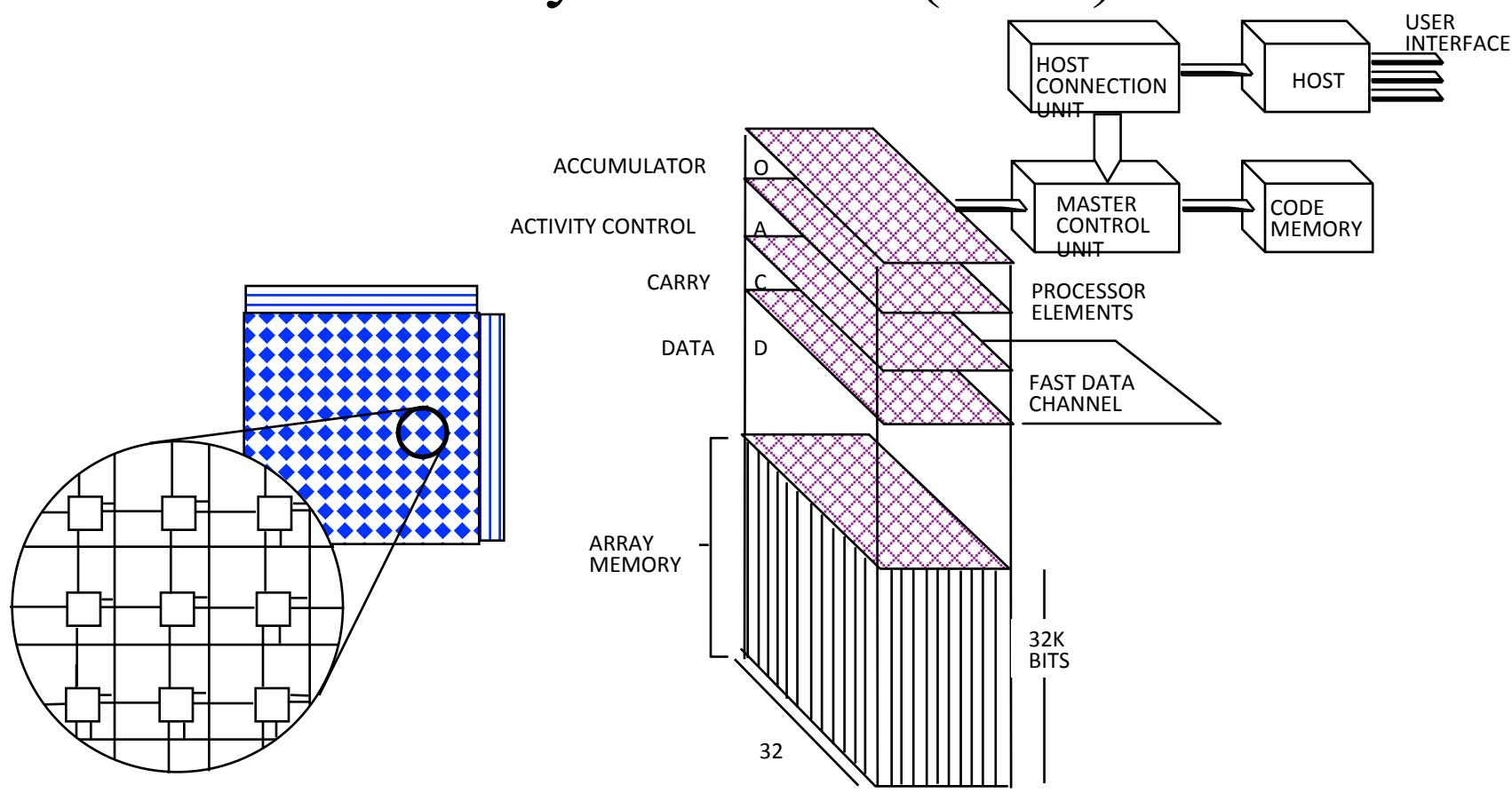
- SIMD (Single Instruction Multiple Data)
 - Logical single thread (instruction) of control
 - Processor associated with data elements

- Architecture
 - Array of simple processors with memory
 - Processors arranged in a regular topology
 - Control processor issues instructions
 - ◆ All processors execute same instruction (maybe disabled)
 - Specialized synchronization and communication
 - Specialized reduction operations
 - Array processing



AMT DAP 500

- Applied Memory Technology (AMT)
- Distributed Array Processor (DAP)

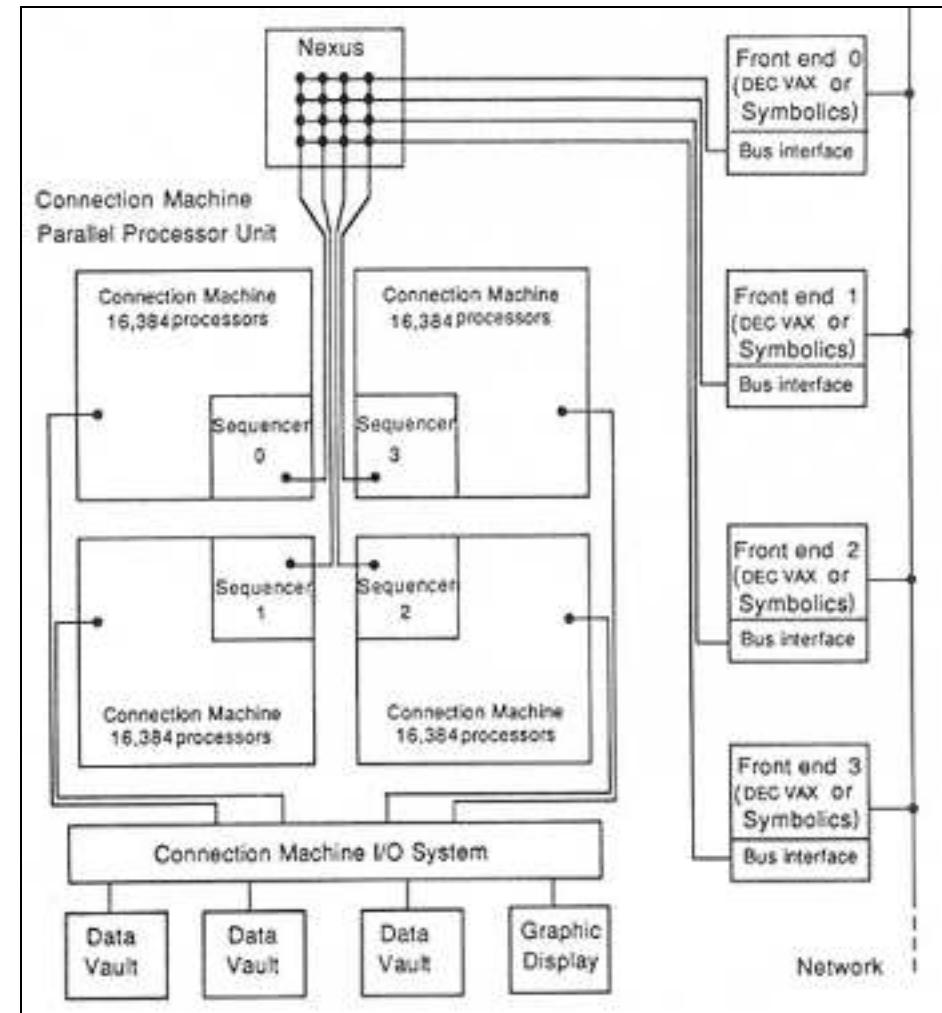
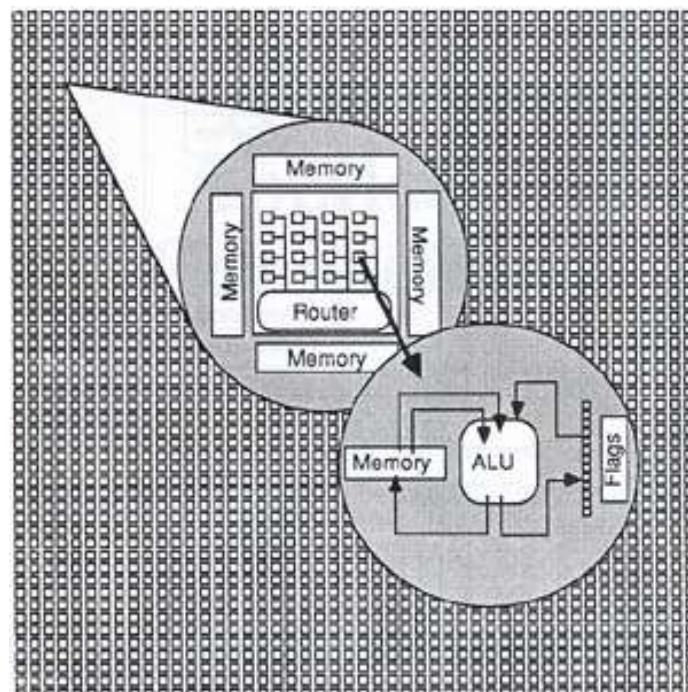


Thinking Machines Connection Machine

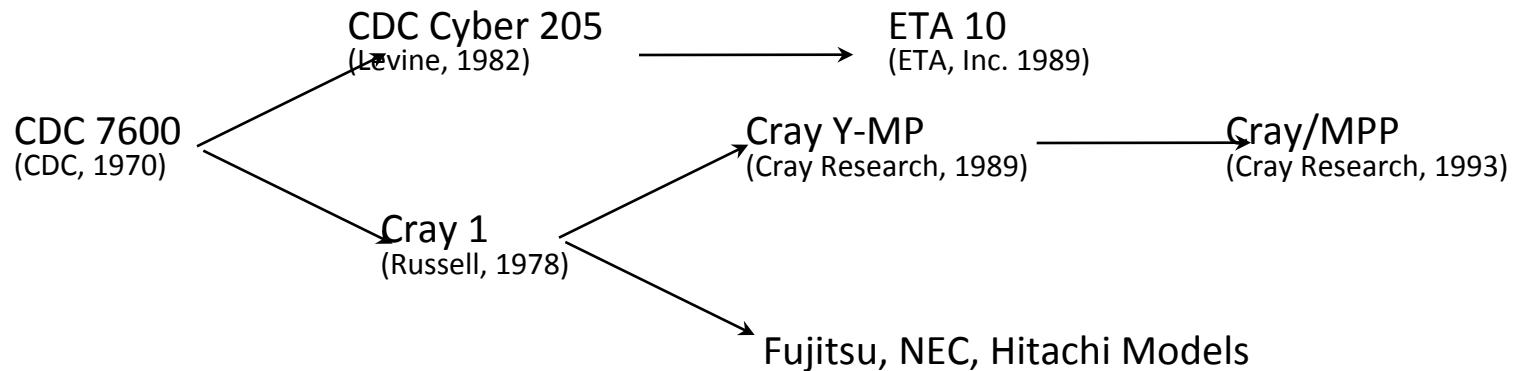


16,000 processors!!!

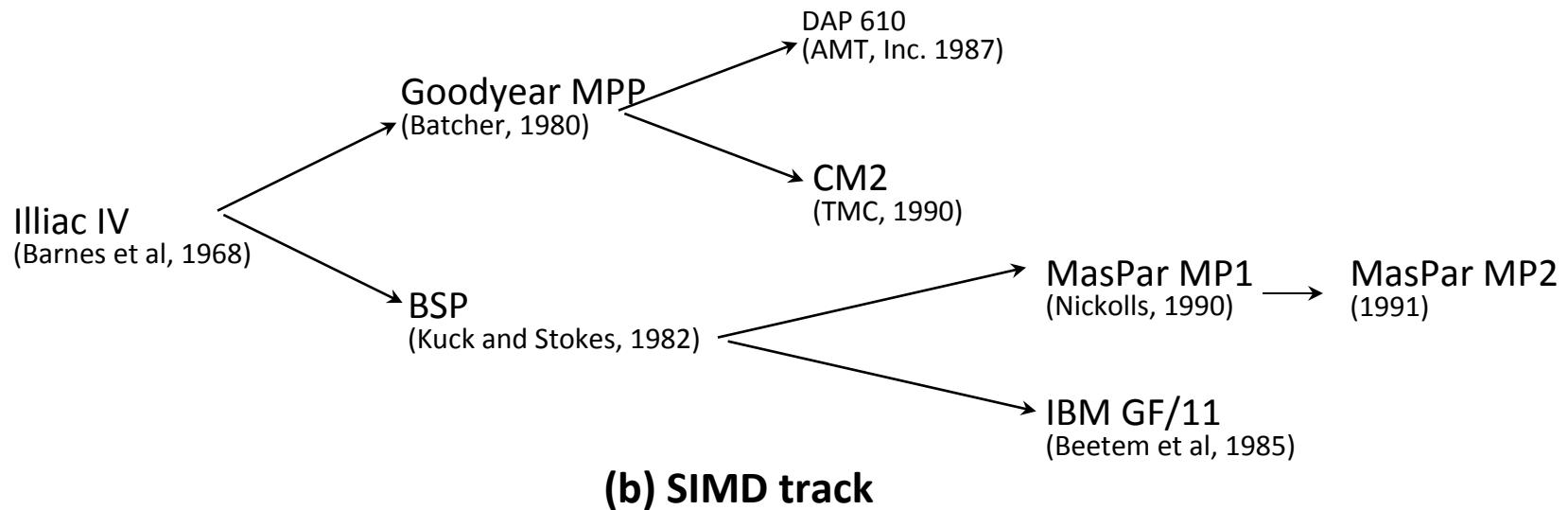
(Tucker, IEEE Computer, Aug. 1988)



Vector and SIMD Processing Timeline



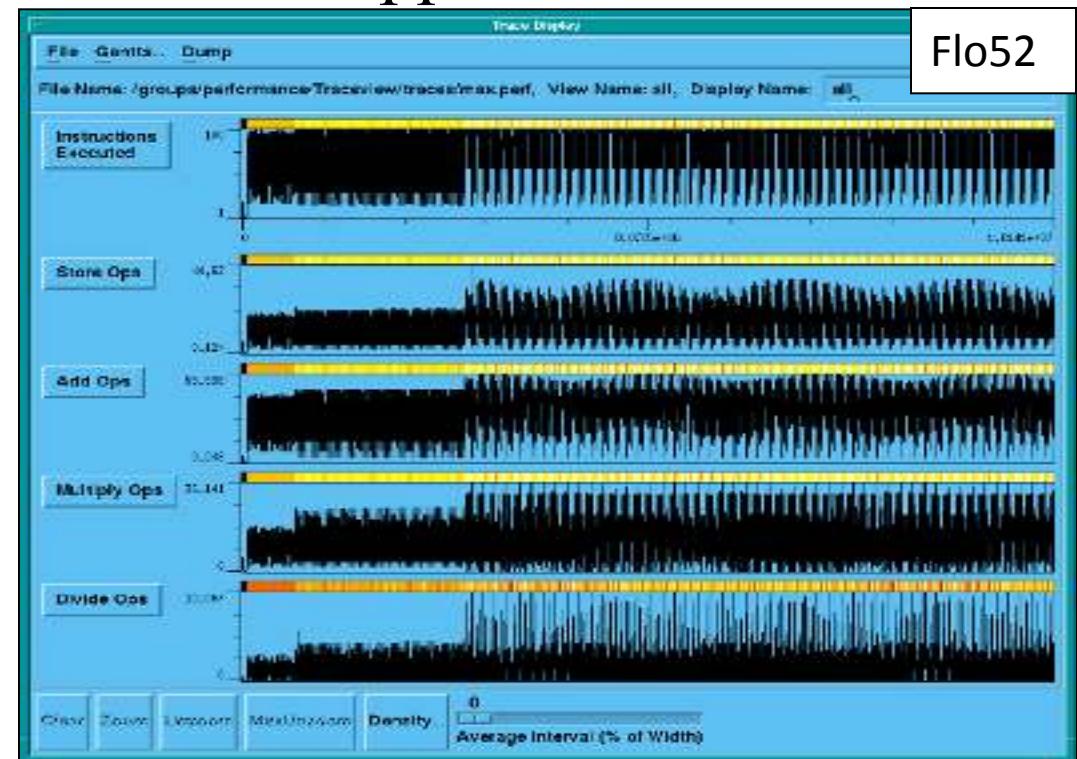
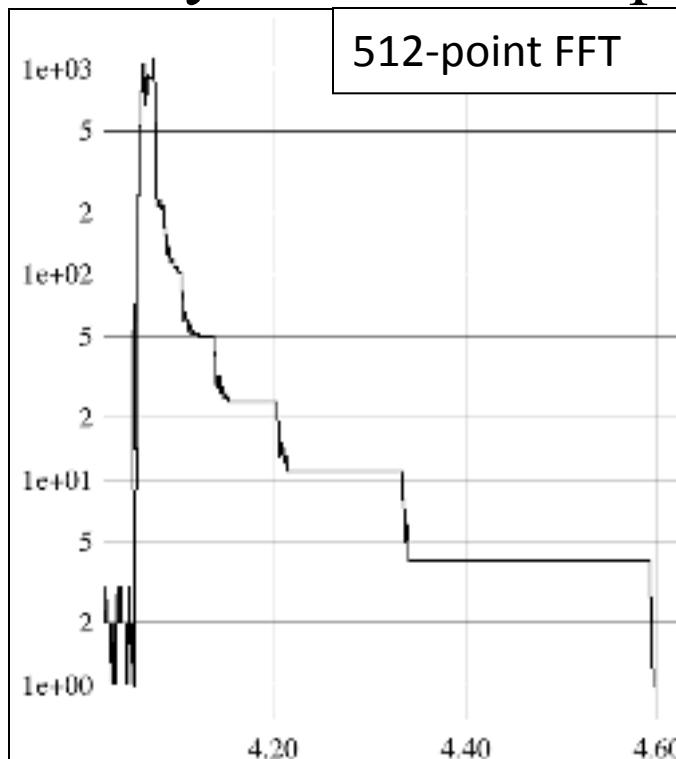
(a) Multivector track



(b) SIMD track

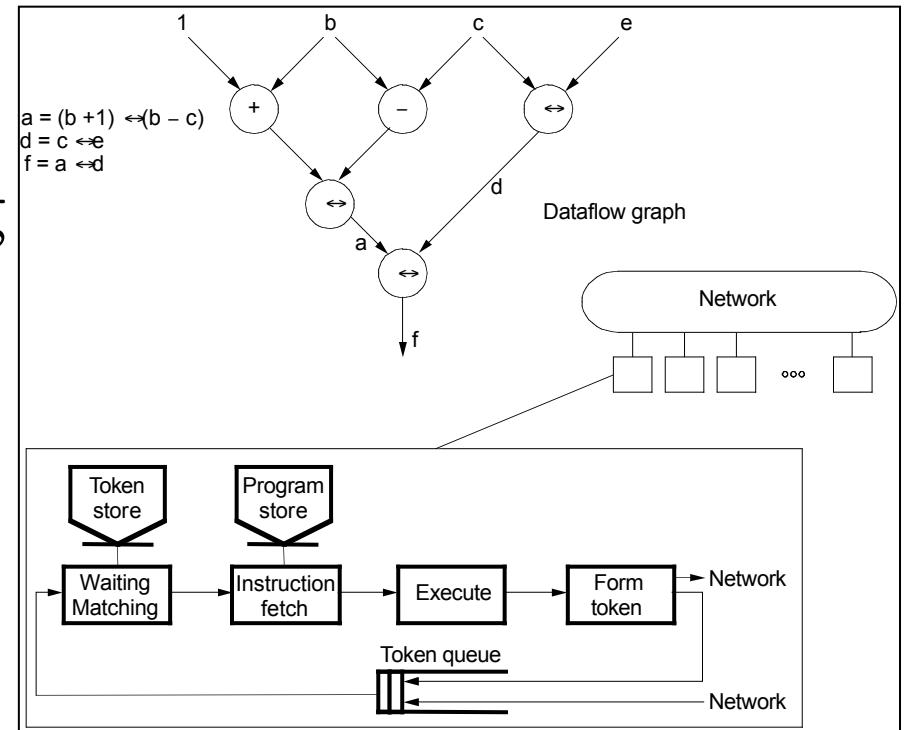
What's the maximum parallelism in a program?

- “MaxPar: An Execution Driven Simulator for Studying Parallel Systems,” Ding-Kai Chen, M.S. Thesis, University of Illinois, Urbana-Champaign, 1989.
- Analyze the data dependencies in application execution



Dataflow Architectures

- Represent computation as graph of dependencies
- Operations stored in memory until operands are ready
- Operations can be dispatched to processors
- Tokens carry tags of next instruction to processor
- Tag compared in matching store
- A match fires execution
- Machine does the hard parallelization work
- Hard to build correctly

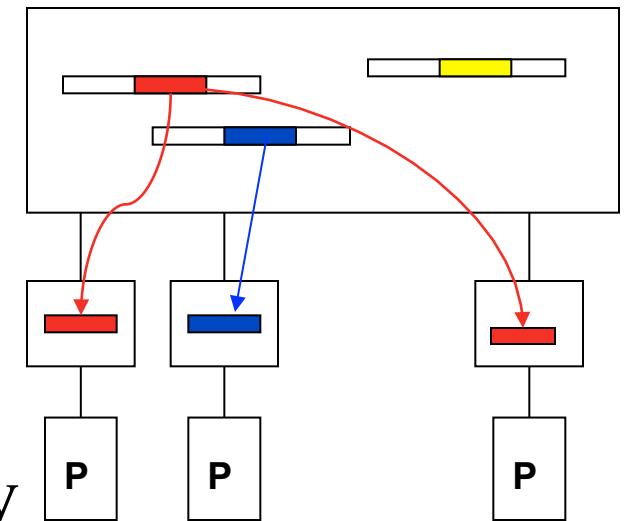


Shared Physical Memory

- Add processors to single processor computer system
- Processors *share* computer system resources
 - Memory, storage, ...
- Sharing physical memory
 - Any processor can reference any memory location
 - Any I/O controller can reference any memory address
 - Single physical memory address space
- Operating system runs on any processor, or all
 - OS see single memory address space
 - Uses shared memory to coordinate
- Communication occurs as a result of loads and stores

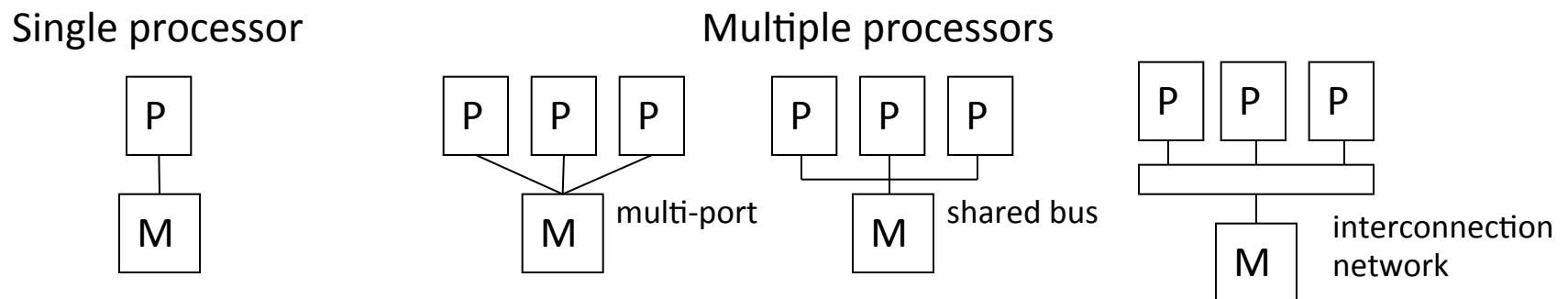
Caching in Shared Memory Systems

- Reduce average latency
 - automatic replication closer to processor
- Reduce average bandwidth
- Data is logically transferred from producer to consumer to memory
 - store reg → mem
 - load reg ← mem
- Processors can share data efficiently
- What happens when store and load are executed on different processors?
- Cache coherence problems

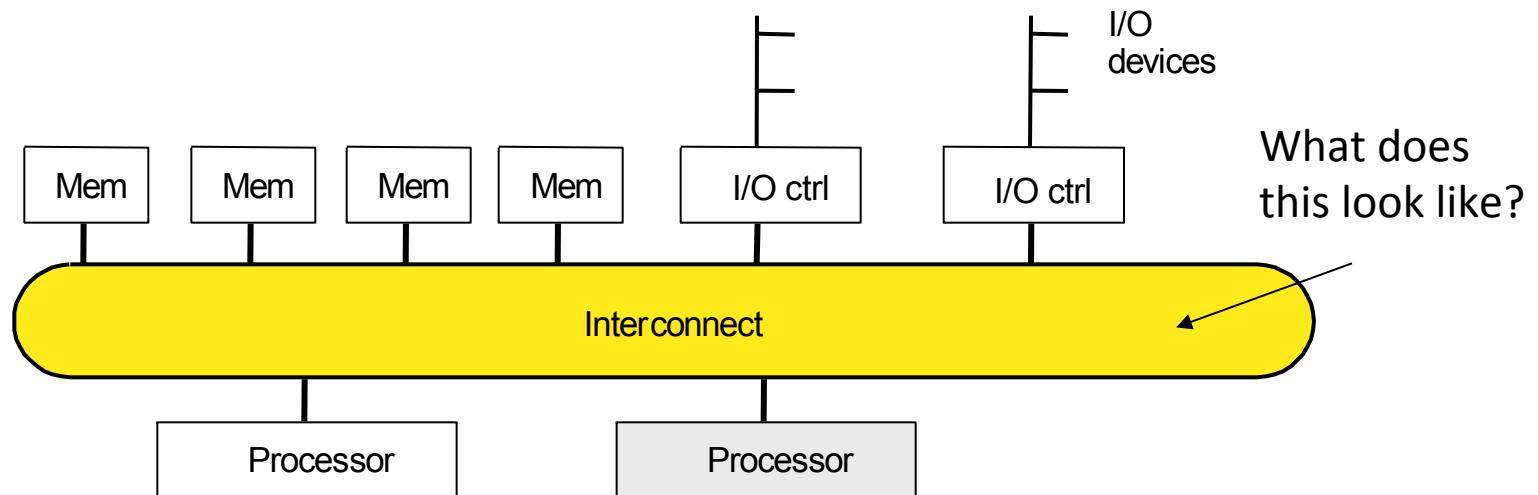


Shared Memory Multiprocessors (SMP)

□ Architecture types

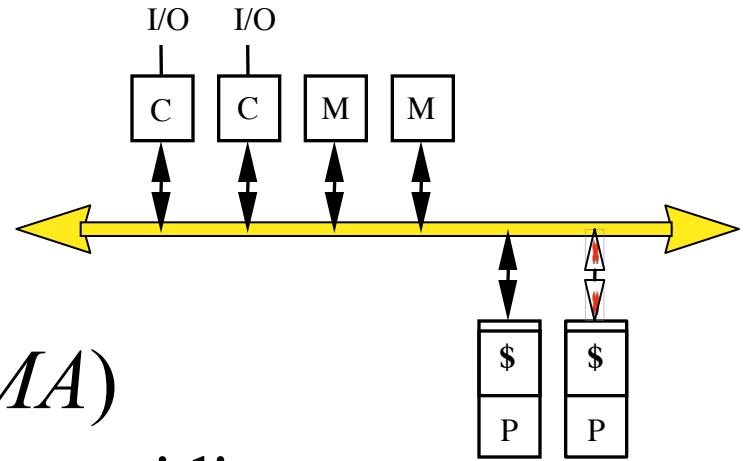


□ Differences lie in memory system interconnection



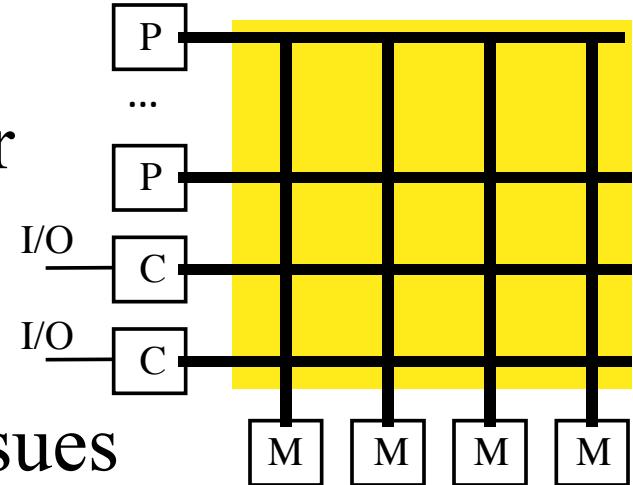
Bus-based SMP

- Memory bus handles all memory read/write traffic
- Processors share bus
- *Uniform Memory Access (UMA)*
 - Memory (not cache) uniformly equidistant
 - Take same amount of time (generally) to complete
- May have multiple memory modules
 - Interleaving of physical address space
- Caches introduce memory hierarchy
 - Lead to data consistency problems
 - Cache coherency hardware necessary (*CC-UMA*)



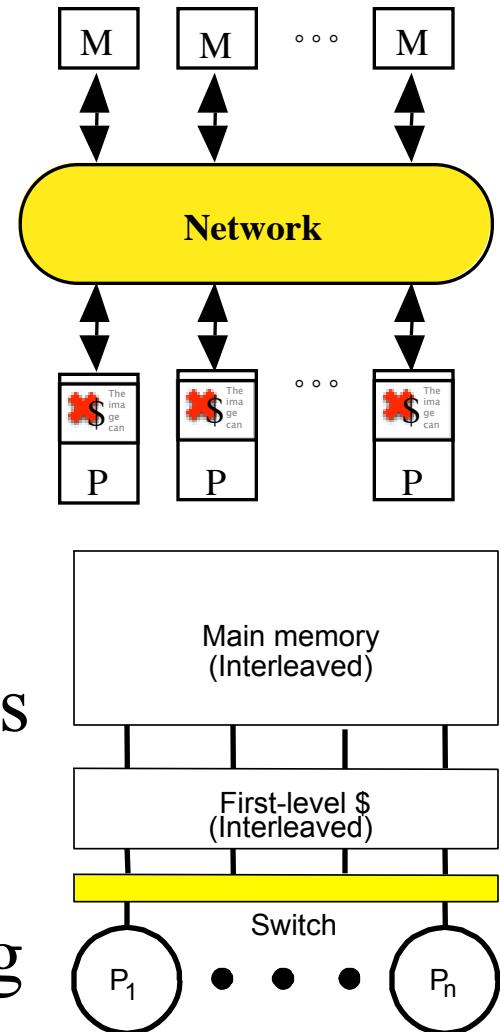
Crossbar SMP

- Replicates memory bus for every processor and I/O controller
 - Every processor has direct path
- UMA SMP architecture
- Can still have cache coherency issues
- Multi-bank memory or interleaved memory
- Advantages
 - Bandwidth scales linearly (no shared links)
- Problems
 - High incremental cost (cannot afford for many processors)
 - Use switched multi-stage interconnection network



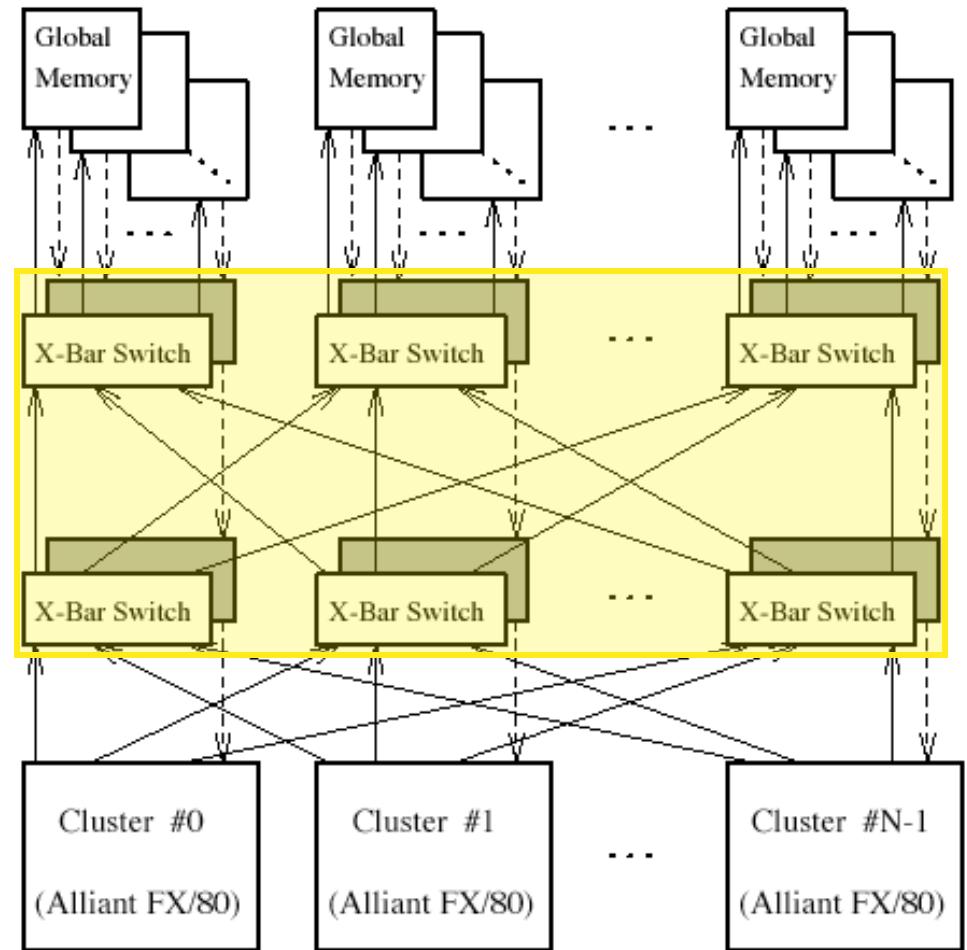
“Dance Hall” SMP and Shared Cache

- Interconnection network connects processors to memory
- Centralized memory (UMA)
- Network determines performance
 - Continuum from bus to crossbar
 - Scalable memory bandwidth
- Memory is physically separated from processors
- Could have cache coherence problems
- Shared cache reduces coherence problem and provides fine grained data sharing

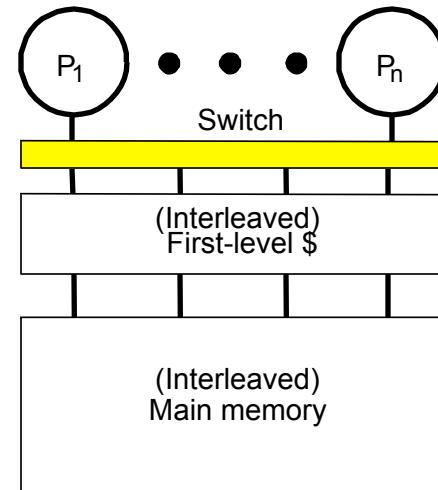


University of Illinois CSRD Cedar Machine

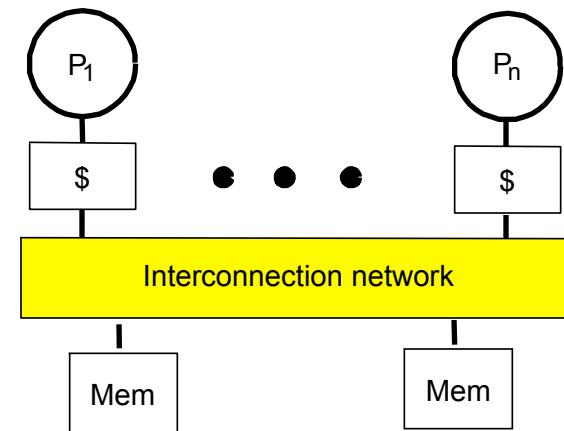
- ❑ Center for Supercomputing Research and Development
- ❑ Multi-cluster scalable parallel computer
- ❑ Alliant FX/80
 - 8 processors w/ vectors
 - Shared cache
 - HW synchronization
- ❑ Omega switching network
- ❑ Shared global memory
- ❑ SW-based global memory coherency



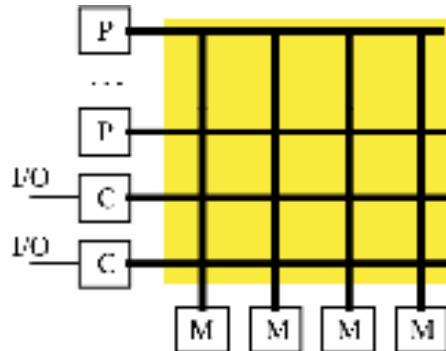
Natural Extensions of the Memory System



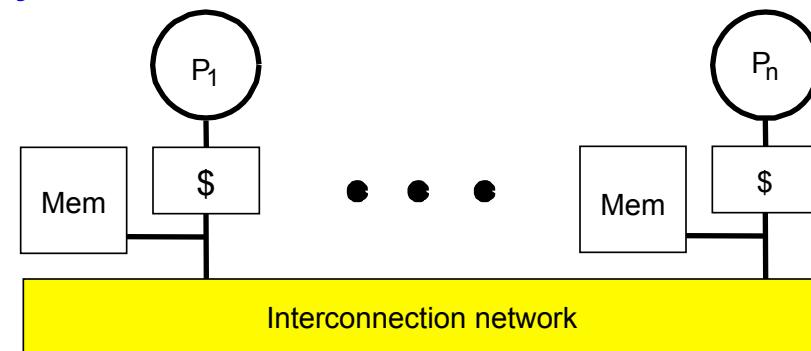
Shared Cache



Centralized Memory
Dance Hall, UMA



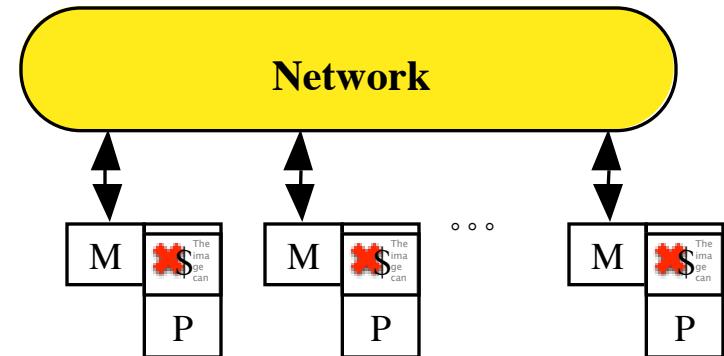
Crossbar, Interleaved



Distributed Memory (NUMA)

Non-Uniform Memory Access (NUMA) SMPs

- Distributed memory
- Memory is physically resident close to each processor
- Memory is still shared
- *Non-Uniform Memory Access (NUMA)*
 - Local memory and remote memory
 - Access to local memory is faster, remote memory slower
 - Access is non-uniform
 - Performance will depend on data locality
- Cache coherency is still an issue (more serious)
- Interconnection network architecture is more scalable



Cache Coherency and SMPs

- Caches play key role in SMP performance
 - Reduce average data access time
 - Reduce bandwidth demands placed on shared interconnect
- Private processor caches create a problem
 - Copies of a variable can be present in multiple caches
 - A write by one processor may not become visible to others
 - ◆ they'll keep accessing stale value in their caches

⇒ *Cache coherence* problem
- What do we do about it?
 - Organize the memory hierarchy to make it go away
 - Detect and take actions to eliminate the problem

Definitions

- Memory operation (load, store, read-modify-write, ...)
 - Memory issue is operation presented to memory system
 - Processor perspective
 - Write: subsequent reads return the value
 - Read: subsequent writes cannot affect the value
 - *Coherent memory system*
 - There exists a serial order of memory operations on each location such that
 - ◆ operations issued by a process appear in order issued
 - ◆ value returned by each read is that written by previous write
- ⇒ write propagation + write serialization

Motivation for Memory Consistency

- Coherence implies that writes to a location become visible to all processors in the same order
- But when does a write become visible?
- How do we establish orders between a write and a read by different processors?
 - Use event synchronization
- Implement hardware protocol for cache coherency
- Protocol will be based on model of memory consistency

P_1	P_2
<hr/>	
/* Assume initial value of A and flag is 0 */	
A = 1;	while (flag == 0); /* spin idly */
flag = 1;	print A;

Memory Consistency

- Specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to each other
 - What orders are preserved?
 - Given a load, constrains the possible values returned by it
- Implications for both programmer and system designer
 - Programmer uses to reason about correctness
 - System designer can use to constrain how much accesses can be reordered by compiler or hardware
- Contract between programmer and system

Sequential Consistency

- Total order achieved by interleaving accesses from different processes
 - Maintains *program order*
 - Memory operations (from all processes) appear to issue, execute, and complete atomically with respect to others
 - As if there was a single memory (no cache)

“A multiprocessor is sequentially consistent if 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 appear in this sequence in the order specified by its program.” [Lamport, 1979]

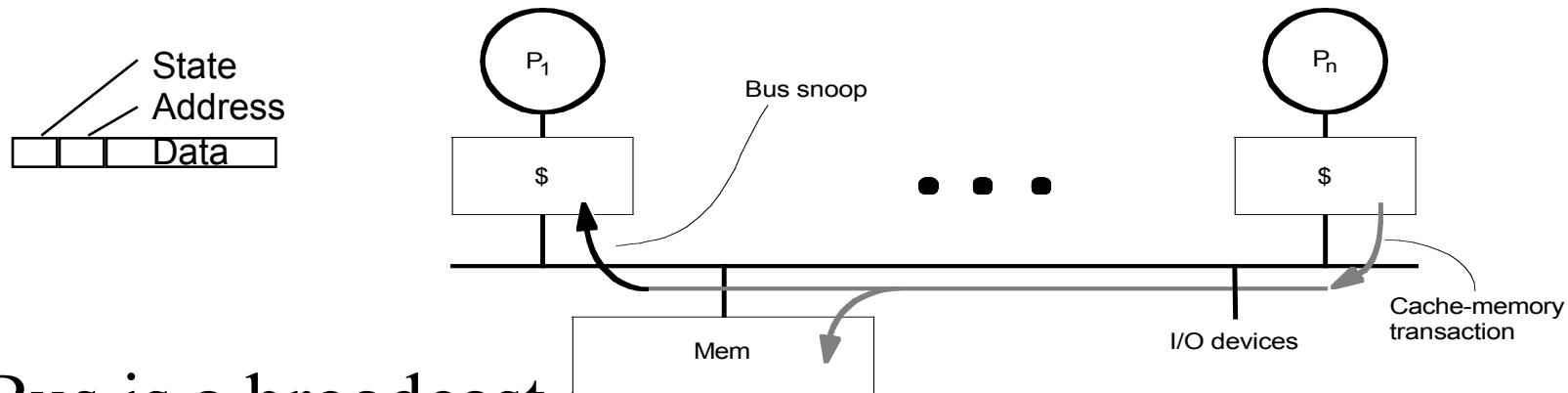
Sequential Consistency (Sufficient Conditions)

- There exist a total order consistent with the memory operations becoming visible in program order
- Sufficient Conditions
 - every process issues memory operations in program order
 - after write operation is issued, the issuing process waits for write to complete before issuing next memory operation (atomic writes)
 - after a read is issued, the issuing process waits for the read to complete and for the write whose value is being returned to complete (globally) before issuing its next memory operation
- Cache-coherent architectures implement consistency

Bus-based Cache-Coherent (CC) Architecture

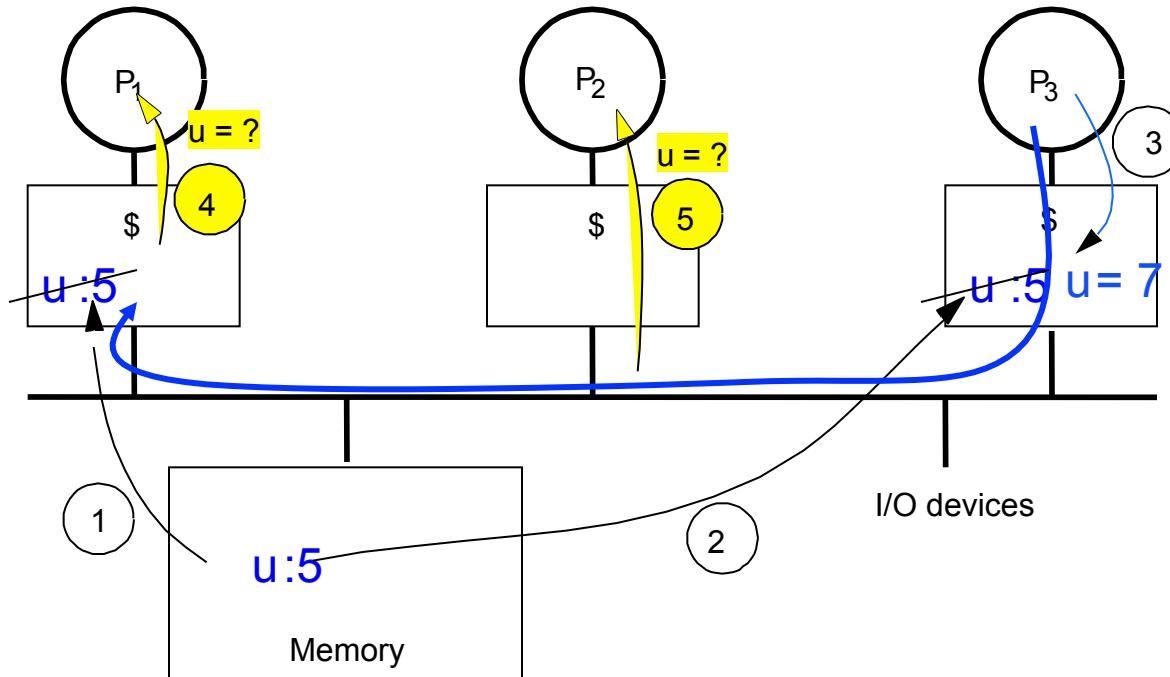
- Bus Transactions
 - Single set of wires connect several devices
 - Bus protocol: arbitration, command/addr, data
 - Every device observes every transaction
- Cache block state transition diagram
 - FSM specifying how disposition of block changes
 - ◆ invalid, valid, dirty
 - *Snoopy protocol*
- Basic Choices
 - Write-through vs Write-back
 - Invalidate vs. Update

Snoopy Cache-Coherency Protocols

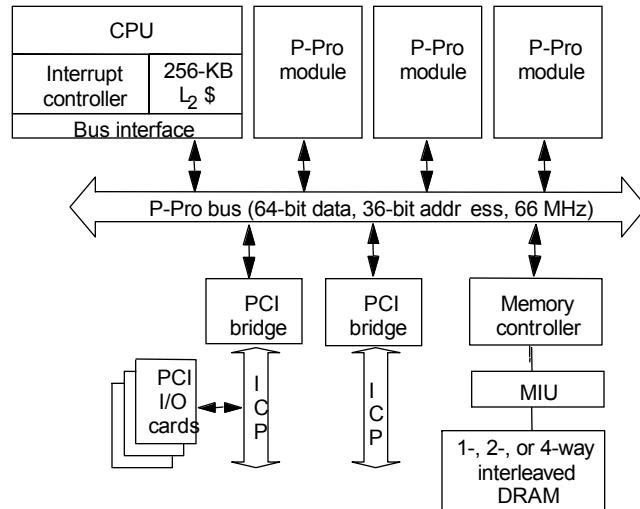
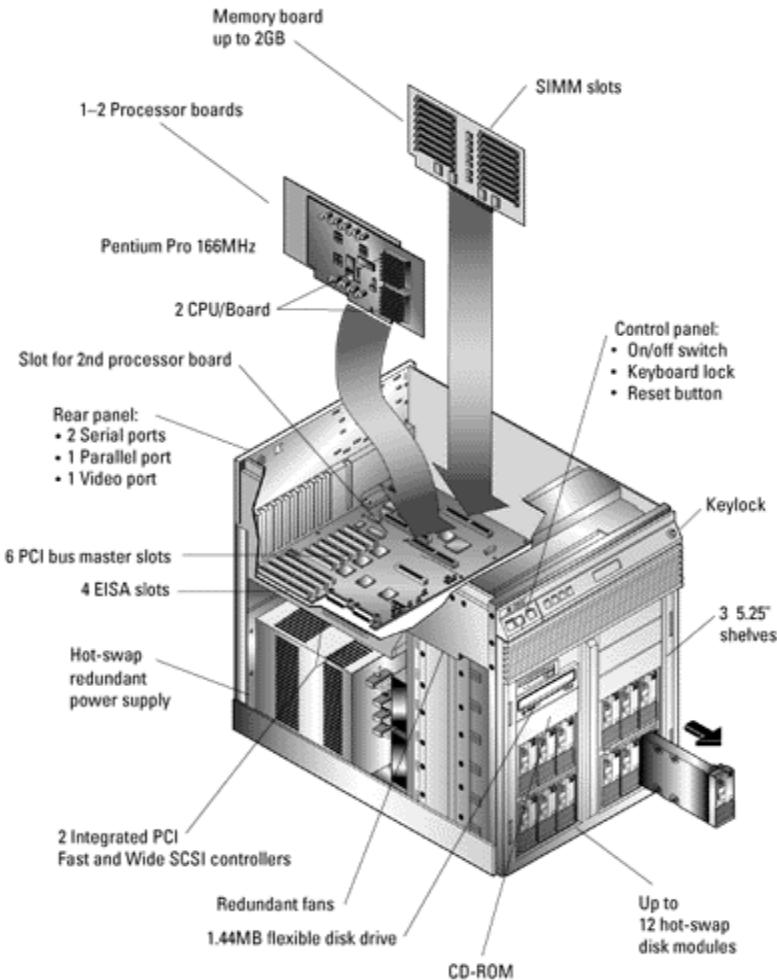


- Bus is a broadcast medium
- Caches know what they have
- Cache controller “snoops” all transactions on shared bus
 - relevant transaction if for a block its cache contains
 - take action to ensure coherence
 - ◆ invalidate, update, or supply value
 - depends on state of the block and the protocol

Example: Write-back Invalidate

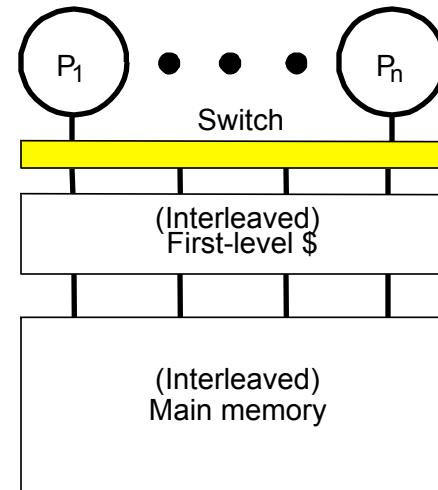


Intel Pentium Pro Quad Processor

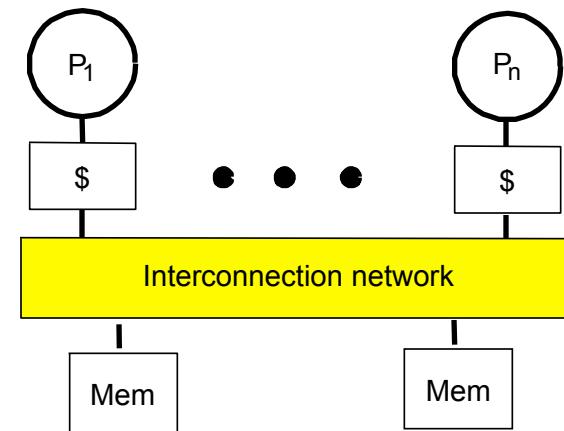


- All coherence and multiprocessing glue in processor module
- Highly integrated, targeted at high volume
- Low latency and bandwidth

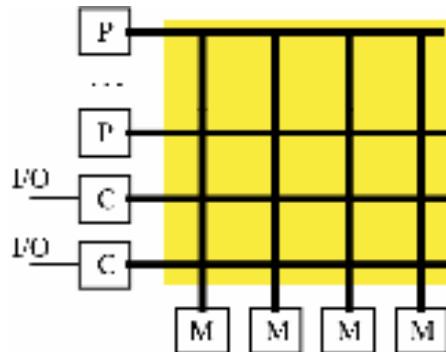
Natural Extensions of the Memory System



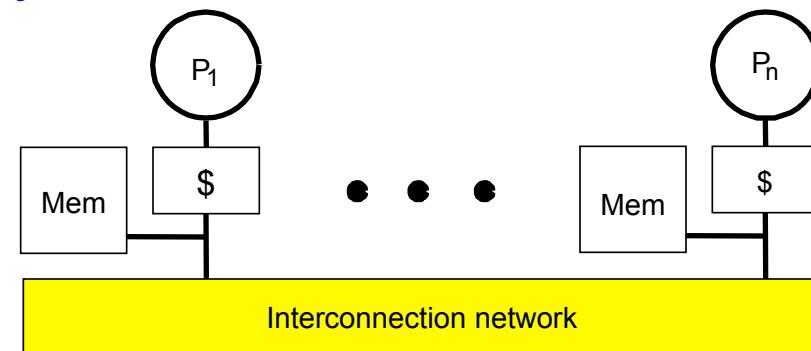
Shared Cache



Centralized Memory
Dance Hall, UMA



Crossbar, Interleaved



Distributed Shared Memory (NUMA)

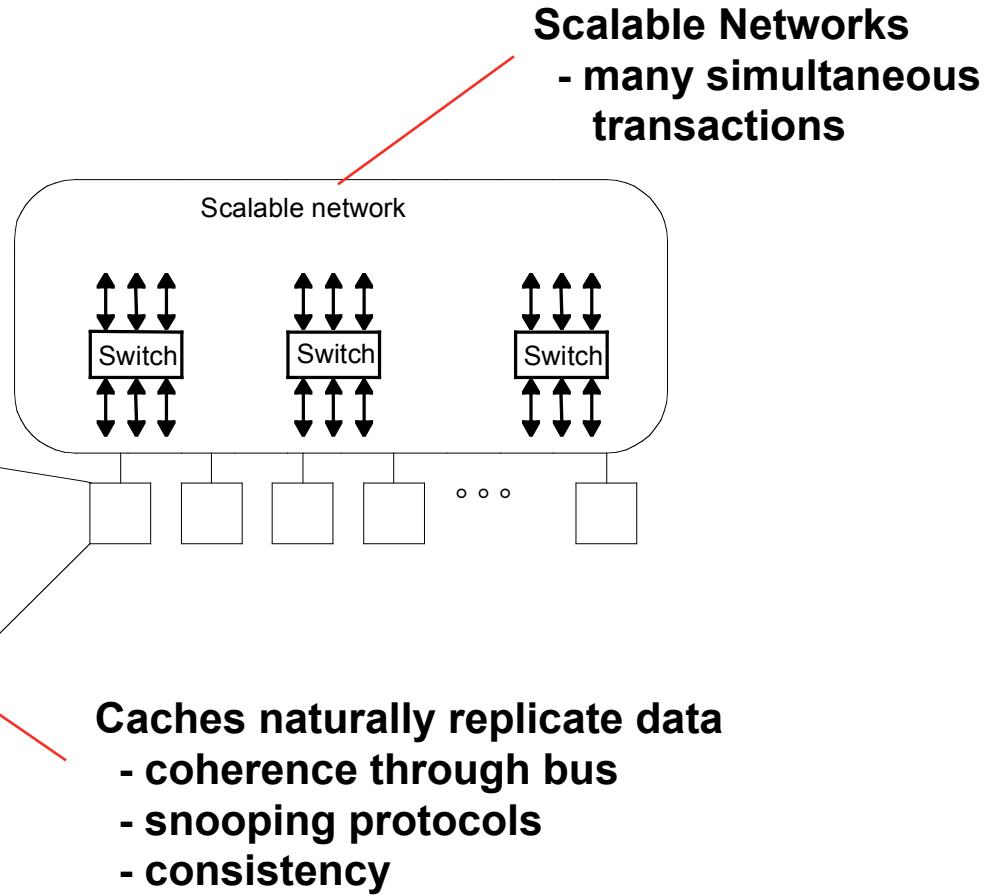
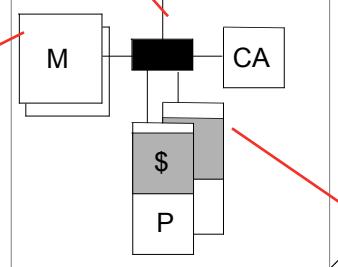
Memory Consistency

- Specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to each other
 - What orders are preserved?
 - Given a load, constrains the possible values returned by it
- Implications for both programmer and system designer
 - Programmer uses to reason about correctness
 - System designer can use to constrain how much accesses can be reordered by compiler or hardware
- Contract between programmer and system
- Need coherency systems to enforce memory consistency

Context for Scalable Cache Coherence

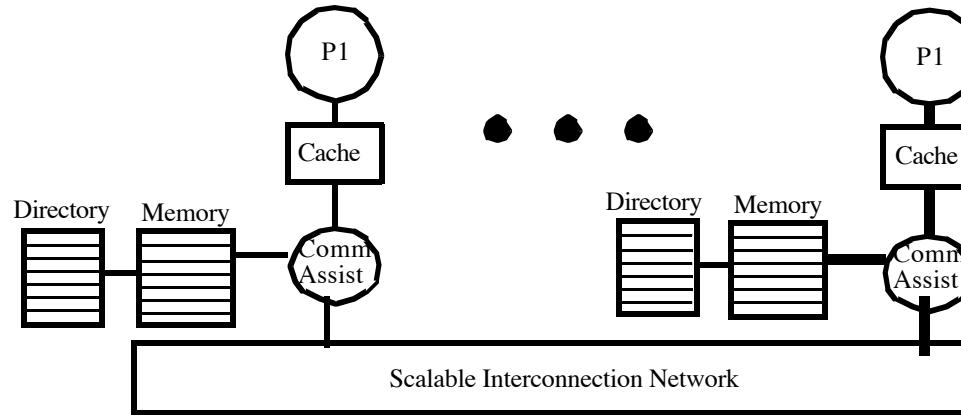
Realizing programming models through net transaction protocols
- efficient node-to-net interface
- interprets transactions

Scalable distributed memory



Need cache coherence protocols that scale!
- no broadcast or single point of order

Generic Solution: Directories



- Maintain state vector explicitly
 - associate with memory block
 - records state of block in each cache
- On miss, communicate with directory
 - determine location of cached copies
 - determine action to take
 - conduct protocol to maintain coherence

Requirements of a Cache Coherent System

- Provide set of states, state transition diagram, and actions
- Manage coherence protocol
 - (0) Determine when to invoke coherence protocol
 - (a) Find info about state of block in other caches to determine action
 - ◆ whether need to communicate with other cached copies
 - (b) Locate the other copies
 - (c) Communicate with those copies (INVAL/UPDATE)
- (0) is done the same way on all systems
 - state of the line is maintained in the cache
 - protocol is invoked if an “access fault” occurs on the line
- Different approaches distinguished by (a) to (c)

Bus-base Cache Coherence

- All of (a), (b), (c) done through broadcast on bus
 - faulting processor sends out a “search”
 - others respond to the search probe and take necessary action
- Could do it in scalable network too
- Conceptually simple, but broadcast doesn’t scale
 - on bus, bus bandwidth doesn’t scale
 - on scalable network, every fault leads to at least p network transactions
- Scalable coherence:
 - can have same cache states and state transition diagram
 - different mechanisms to manage protocol

Basic Snoop Protocols

- Write Invalidate :
 - Multiple readers, single writer
 - Write to shared data: an invalidate is sent to all caches
 - Read Miss:
 - ◆ Write-through: memory is always up-to-date
 - ◆ Write-back: snoop in caches to find most recent copy
- Write Broadcast (typically write through):
 - Write to shared data: broadcast on bus, snoop and update
- Write serialization: bus serializes requests!
- Write Invalidate versus Broadcast

Snooping Cache Variations

Basic Protocol	Berkeley Protocol	Illinois Protocol	MESI Protocol
Exclusive	Owned Exclusive	Private Dirty	Modified (private,!=Memory)
Shared	Owned Shared	Private Clean	Exclusive (private,=Memory)
Invalid	Shared	Shared	Shared (shared,=Memory)
	Invalid	Invalid	Invalid

Owner can update via bus invalidate operation

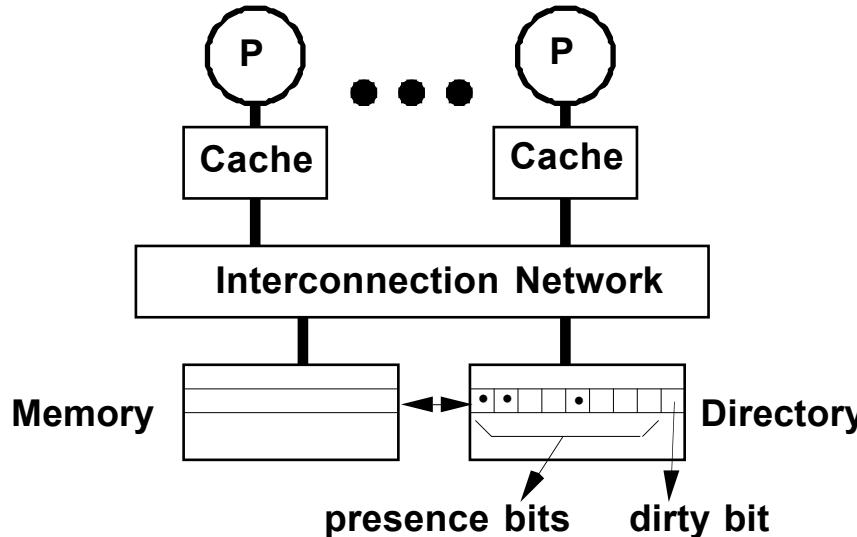
Owner must write back when replaced in cache

If read sourced from memory, then Private Clean
if read sourced from other cache, then Shared
Can write in cache if held private clean or dirty

Scalable Approach: Directories

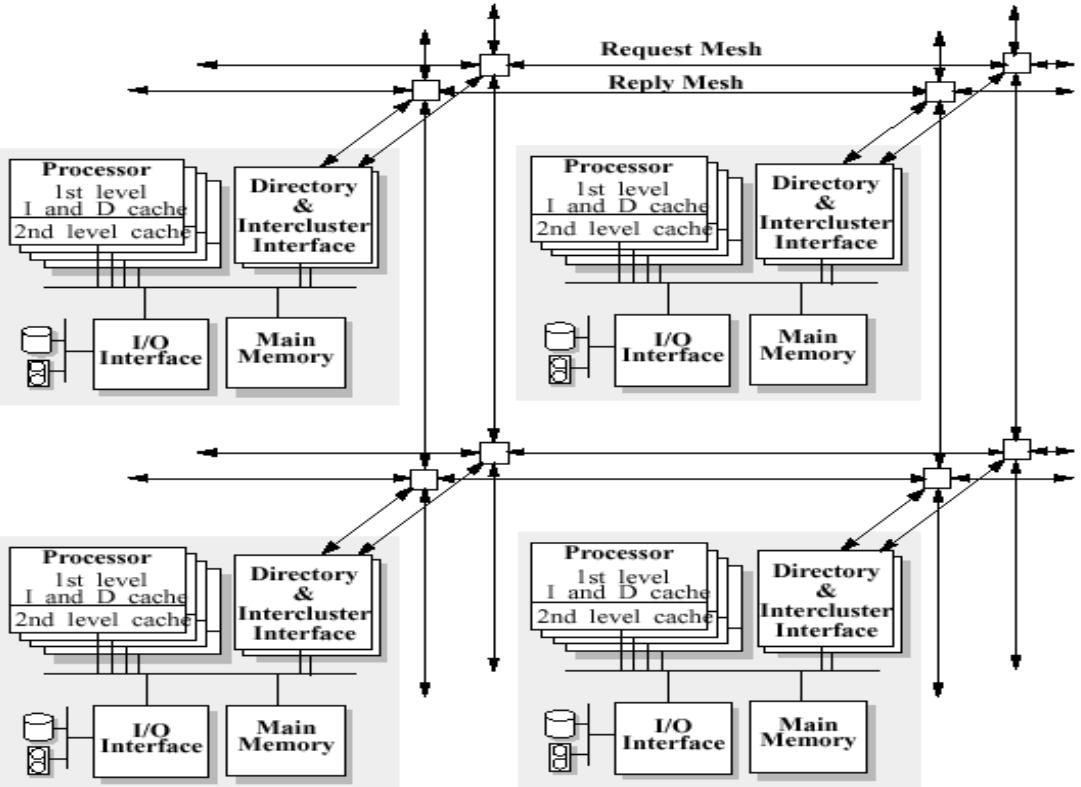
- Every memory block has associated directory information
 - keeps track of copies of cached blocks and their states
 - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
 - in scalable networks, communication with directory and copies is through network transactions
- Many alternatives for organizing directory information

Basic Operation of Directory



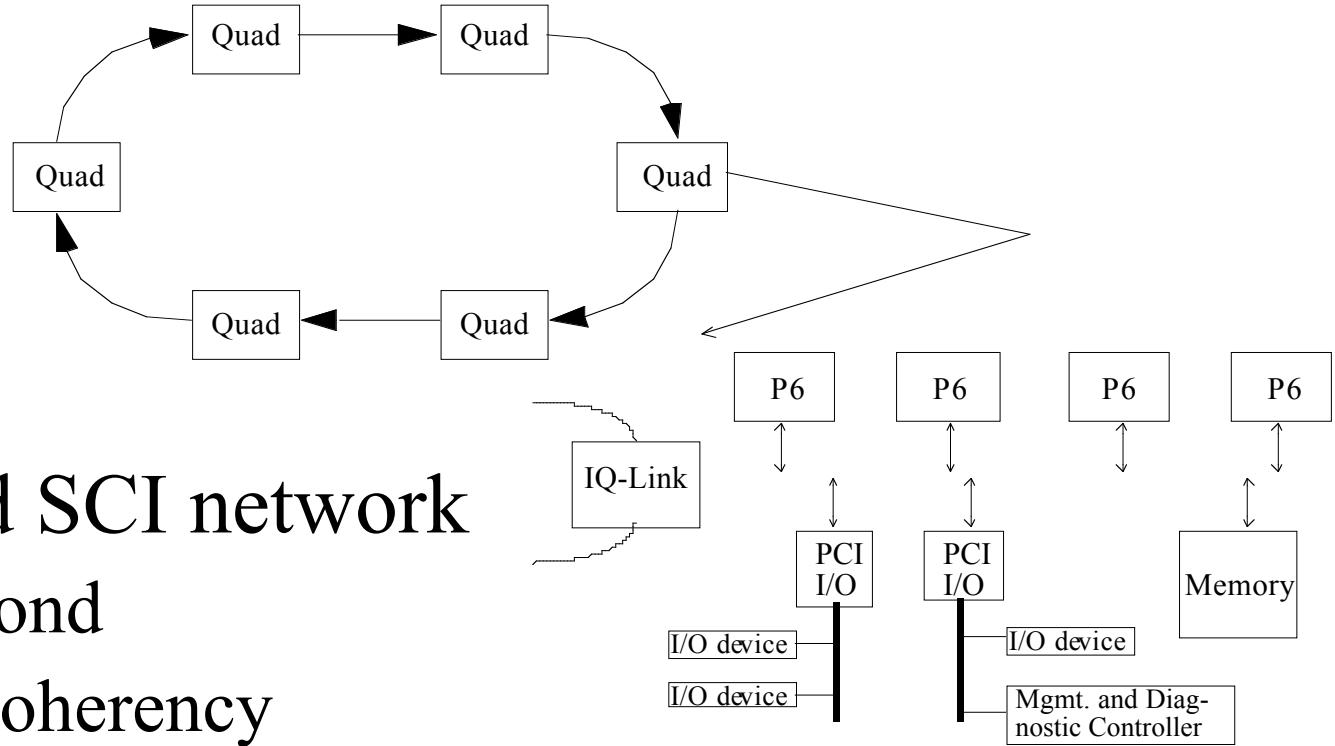
- k processors
- Each cache-block in memory
 - k presence bits and 1 dirty bit
- Each cache-block in cache
 - 1 valid bit and 1 dirty (owner) bit
- Read from memory
 - Dirty bit OFF
 - Dirty bit ON
- Write to memory
 - Dirty bit OFF

DASH Cache-Coherent SMP

- Directory Architecture for Shared Memory
 - Stanford research project (early 1990s) for studying how to build cache-coherent shared memory architectures
 - Directory-based cache
 - D. Lenoski, et al., “The Stanford Dash Multiprocessor,” IEEE Computer, Volume 25 Issue 3, pp: 63-79, March 1992
- 
- The diagram illustrates the DASH Cache-Coherent SMP architecture. It shows four clusters arranged in a square grid. Each cluster contains a Processor unit (with 1st level I and D cache, 2nd level cache, I/O Interface, and Main Memory), a Directory & Intercluster Interface, and a Request Mesh/Reply Mesh connection. The Request Mesh consists of a 2x2 grid of switches connecting the four clusters. The Reply Mesh consists of a 2x2 grid of switches, also connecting the four clusters. Arrows indicate the flow of requests and replies between the clusters via these mesh networks.

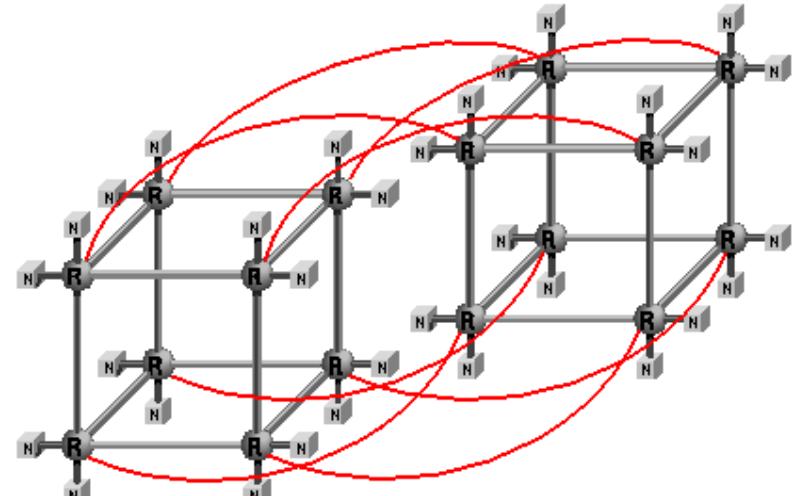
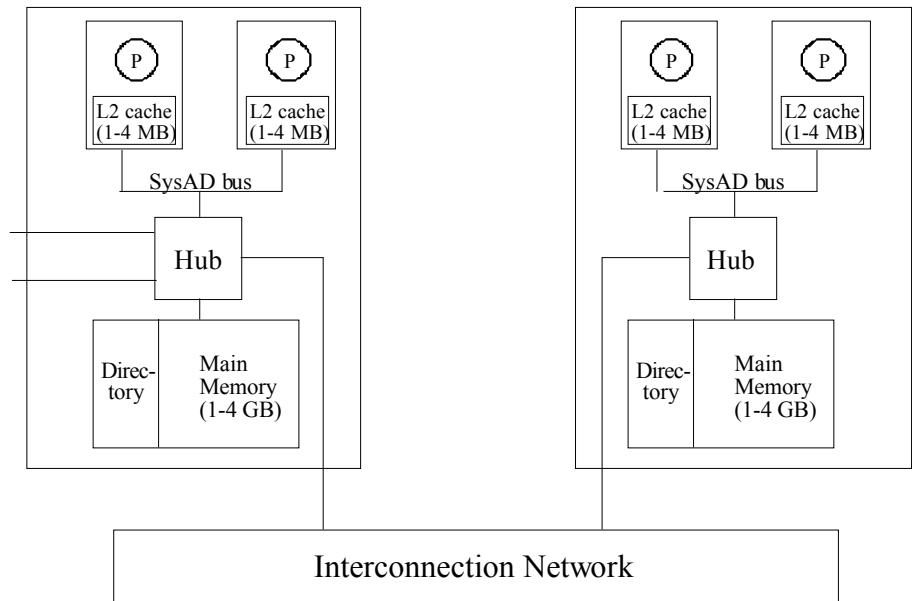
Sequent NUMA-Q

- Ring-based SCI network
 - 1 GB/second
 - Built-in coherency
- Commodity SMPs as building blocks
 - Extend coherency mechanism
- Split transaction bus



SGI Origin 2000

- Scalable shared memory multiprocessor
- MIPS R10000 CPU
- NUMAlink router
- Directory-based cache coherency (MESI)
- ASCI Blue Mountain

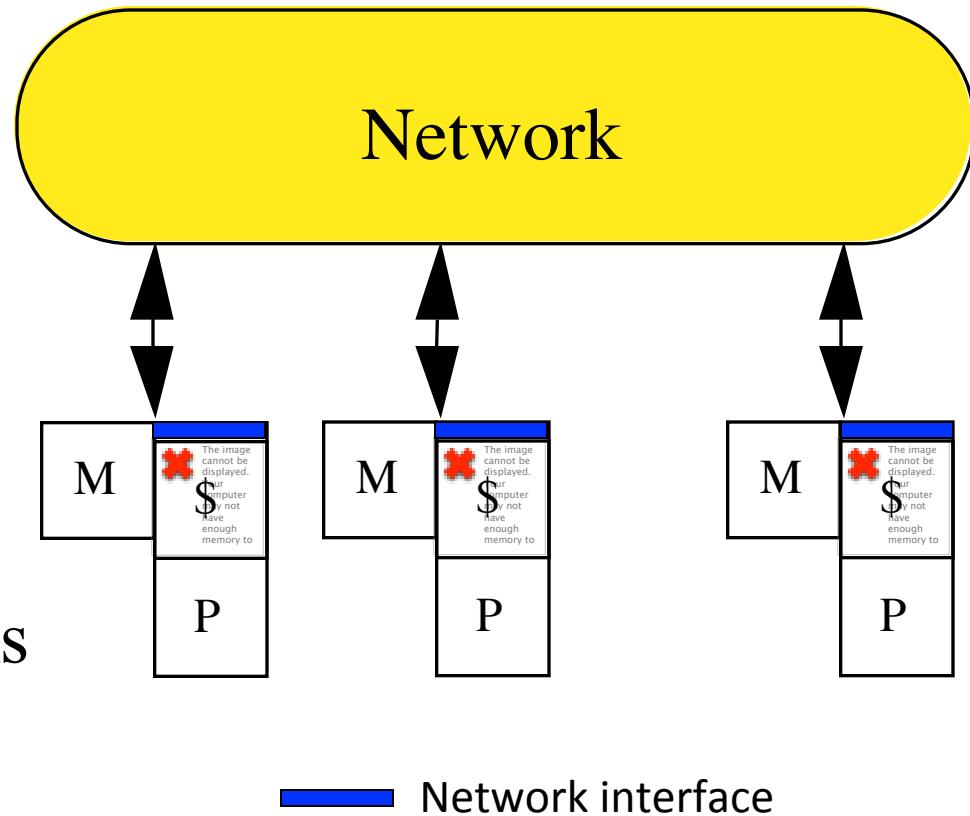


Distributed Memory Multiprocessors

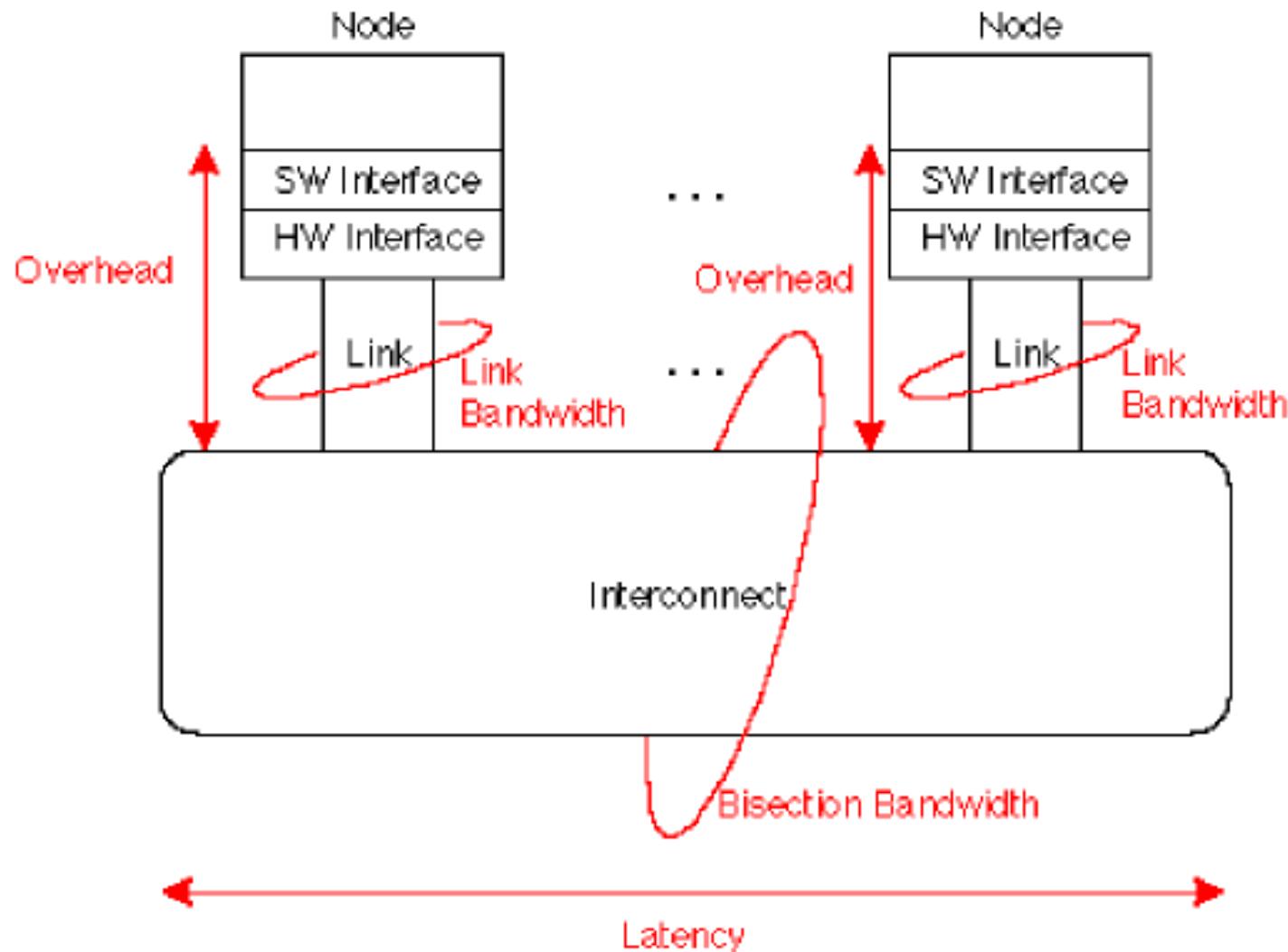
- Each processor has a local memory
 - Physically separated memory address space
- Processors must communicate to access non-local data
 - Message communication (message passing)
 - ◆ *Message passing architecture*
 - Processor interconnection network
- Parallel applications must be partitioned across
 - Processors: execution units
 - Memory: data partitioning
- Scalable architecture
 - Small incremental cost to add hardware (cost of node)

Distributed Memory (MP) Architecture

- Nodes are complete computer systems
 - Including I/O
- Nodes communicate via interconnection network
 - Standard networks
 - Specialized networks
- Network interfaces
 - Communication integration
- Easier to build



Network Performance Measures



Overhead: latency of interface vs. **Latency:** network

Performance Metrics: Latency and Bandwidth

- Bandwidth
 - Need high bandwidth in communication
 - Match limits in network, memory, and processor
 - Network interface speed vs. network bisection bandwidth
- Latency
 - Performance affected since processor may have to wait
 - Harder to overlap communication and computation
 - Overhead to communicate is a problem in many machines
- Latency hiding
 - Increases programming system burden
 - Examples: communication/computation overlaps, prefetch

Scalable, High-Performance Interconnect

- Interconnection network is core of parallel architecture
- Requirements and tradeoffs at many levels
 - Elegant mathematical structure
 - Deep relationship to algorithm structure
 - Hardware design sophistication
- Little consensus
 - Performance metrics?
 - Cost metrics?
 - Workload?
 - ...

What Characterizes an Interconnection Network?

- Topology (what)
 - Interconnection structure of the network graph
- Routing Algorithm (which)
 - Restricts the set of paths that messages may follow
 - Many algorithms with different properties
- Switching Strategy (how)
 - How data in a message traverses a route
 - *circuit switching* vs. *packet switching*
- Flow Control Mechanism (when)
 - When a message or portions of it traverse a route
 - What happens when traffic is encountered?

Topological Properties

- Routing distance
 - Number of links on route from source to destination
- Diameter
 - Maximum routing distance
- Average distance
- Partitioned network
 - Removal of links resulting in disconnected graph
 - Minimal cut
- Scaling increment
 - What is needed to grow the network to next valid degree

Interconnection Network Types

Topology	Degree	Diameter	Ave Dist	Bisection	$D(D \text{ ave}) @ P=1024$
1D Array	2	$N-1$	$N / 3$	1	huge
1D Ring	2	$N/2$	$N/4$	2	
2D Mesh	4	$2 (N^{1/2} - 1)$	$2/3 N^{1/2}$	$N^{1/2}$	63 (21)
2D Torus	4	$N^{1/2}$	$1/2 N^{1/2}$	$2N^{1/2}$	32 (16)
k-ary n-cube	$2n$	$nk/2$	$nk/4$	$nk/4$	15 (7.5) @n=3
Hypercube	$n = \log N$	n	$n/2$	$N/2$	10 (5)

N = # nodes

Communication Performance

- $\text{Time}(n)_{s-d} = \text{overhead} + \text{routing delay} + \text{channel occupancy} + \text{contention delay}$
- $\text{occupancy} = (n + n_h) / b$

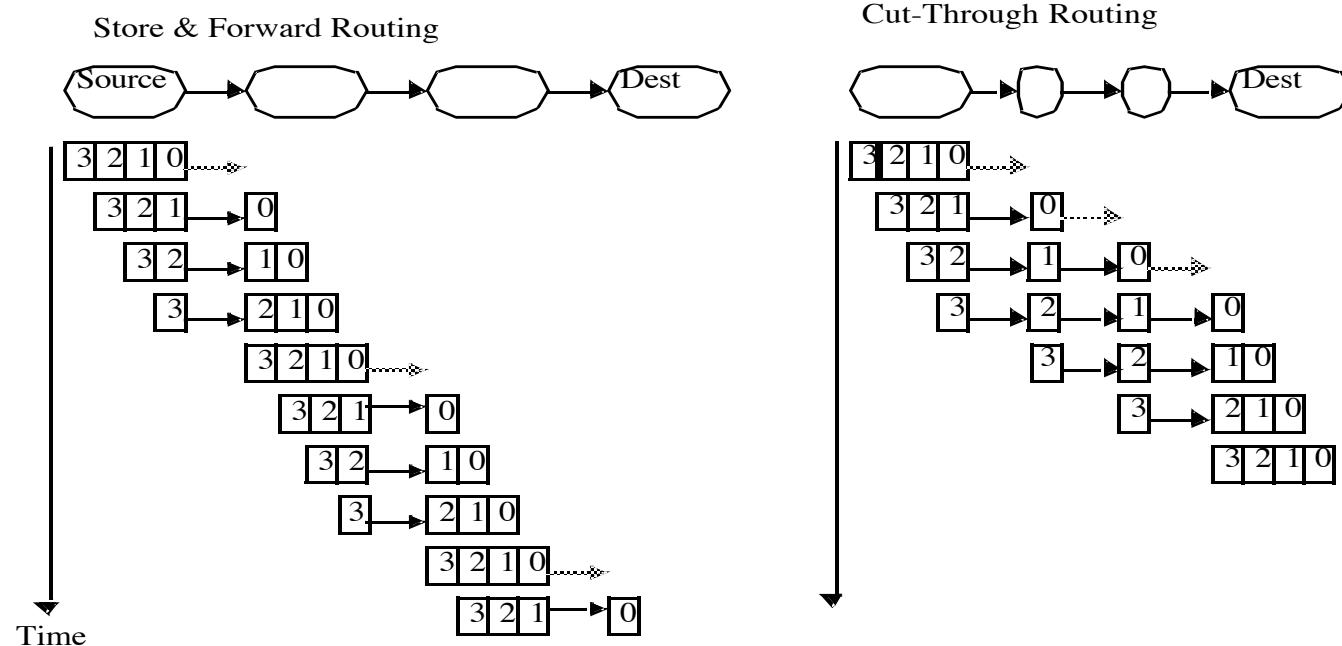
n = message #bytes

n_h = header #bytes

b = bitrate of communication link

- What is the routing delay?
- Does contention occur and what is the cost?

Store-and-Forward vs. Cut-Through Routing



- $h(n/b + \Delta)$ $n/b + h \Delta$
- What if message is fragmented?
- *Wormhole* vs. *Virtual cut-through*

Networks of Real Machines (circa 2000)

Machine	Topology	Speed	Width	Delay	Flit
nCUBE/2	hypercube	25 ns	1	40 cycles	32
CM-5	fat-tree	25 ns	4	10 cycles	4
SP-2	banyan	25 ns	8	5 cycles	16
Paragon	2D mesh	11.5 ns	16	2 cycles	16
T3D	3D torus	6.67 ns	16	2 cycles	16
DASH	torus	30 ns	16	2 cycles	16
Origin	hypercube	2.5 ns	20	16 cycles	160
Myrinet	arbitrary	6.25 ns	16	50 cycles	16

A message is broken up into *flits* for transfer.

Message Passing Model

- Hardware maintains send and receive message buffers
- Send message (synchronous)
 - Build message in local message send buffer
 - Specify receive location (processor id)
 - Initiate send and wait for receive acknowledge
- Receive message (synchronous)
 - Allocate local message receive buffer
 - Receive message byte stream into buffer
 - Verify message (e.g., checksum) and send acknowledge
- Memory to memory copy with acknowledgement and pairwise synchronization

Advantages of Shared Memory Architectures

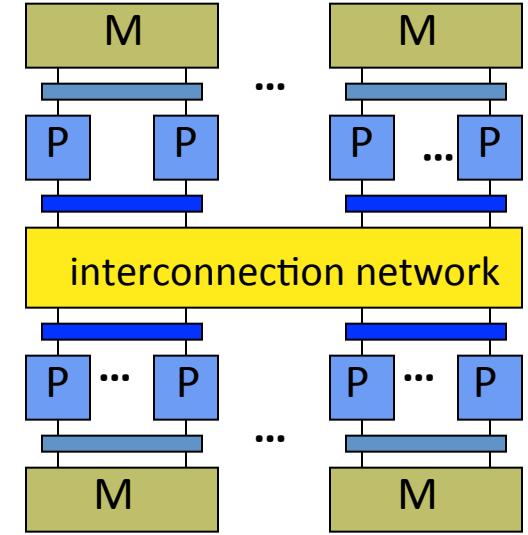
- Compatibility with SMP hardware
- Ease of programming when communication patterns are complex or vary dynamically during execution
- Ability to develop applications using familiar SMP model, attention only on performance critical accesses
- Lower communication overhead, better use of BW for small items, due to implicit communication and memory mapping to implement protection in hardware, rather than through I/O system
- HW-controlled caching to reduce remote communication by caching of all data, both shared and private

Advantages of Distributed Memory Architectures

- The hardware can be simpler (especially versus NUMA) and is more scalable
- Communication is explicit and simpler to understand
- Explicit communication focuses attention on costly aspect of parallel computation
- Synchronization is naturally associated with sending messages, reducing the possibility for errors introduced by incorrect synchronization
- Easier to use sender-initiated communication, which may have some advantages in performance

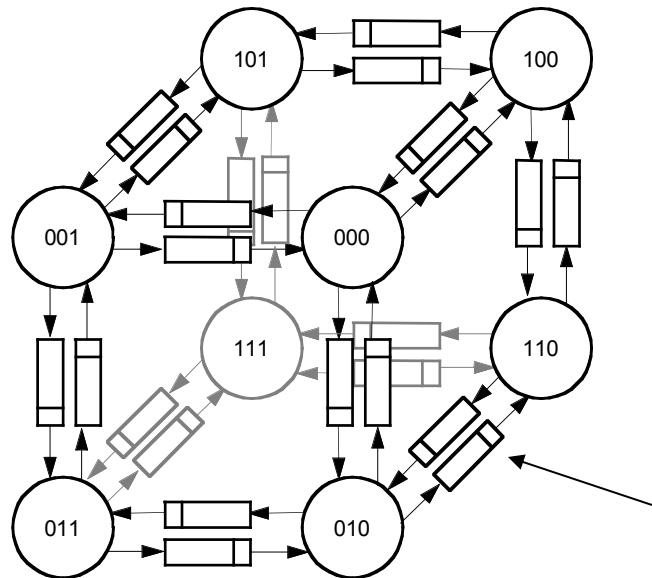
Clusters of SMPs

- Clustering
 - Integrated packaging of nodes
- Motivation
 - Ammortize node costs by sharing packaging and resources
 - Reduce network costs
 - Reduce communications bandwidth requirements
 - Reduce overall latency
 - More parallelism in a smaller space
 - Increase node performance
- Scalable parallel systems today are built as SMP clusters



CalTech Cosmic Cube

- ❑ First distributed memory message passing system
- ❑ Hypercube-based communications network



FIFO on each link
- store and forward

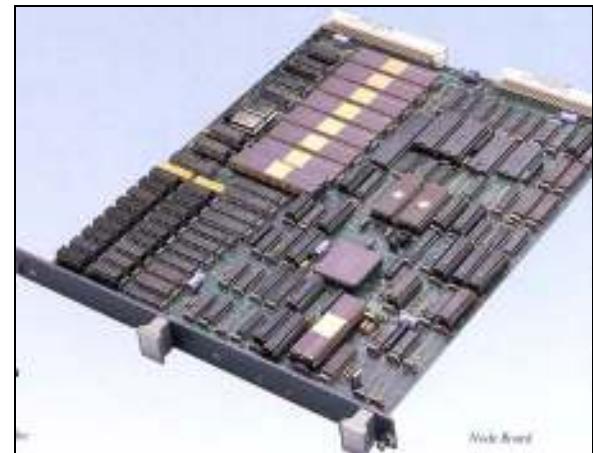
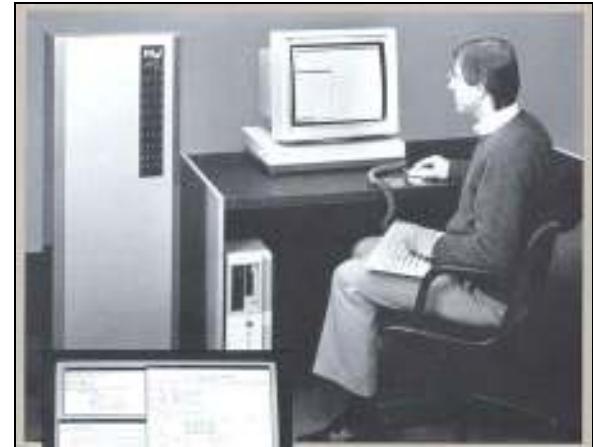


Compute node

- ❑ Chuck Seitz, Geoffrey Fox

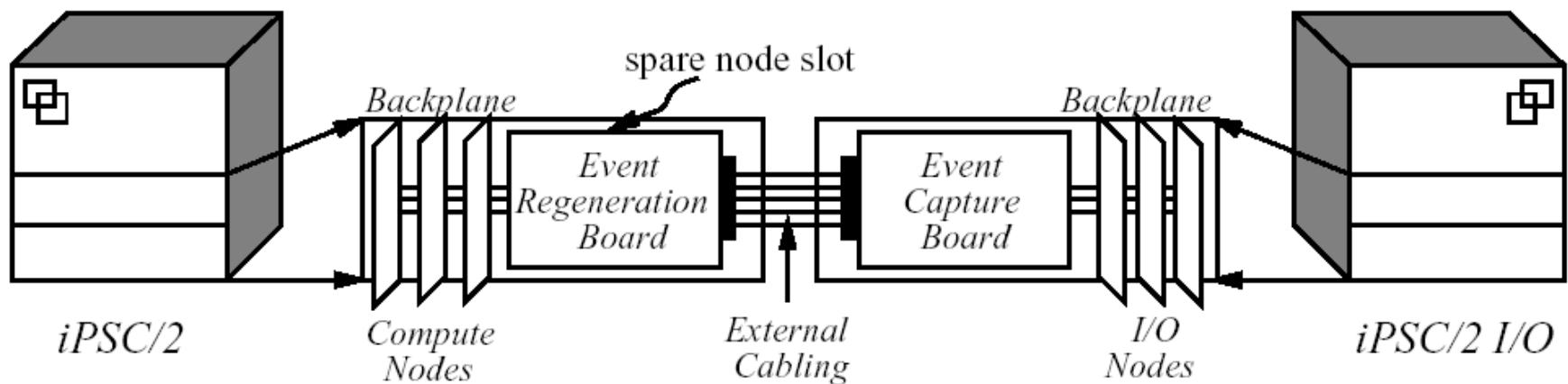
Intel iPSC/1, iPSC/2, iPSC/860

- Shift to general links
 - DMA, enabling non-blocking ops
 - ◆ Buffered by system at destination until recv
 - Store&forward routing
- Diminishing role of topology
 - Any-to-any pipelined routing
 - node-network interface dominates communication time
 - Simplifies programming
 - Allows richer design space



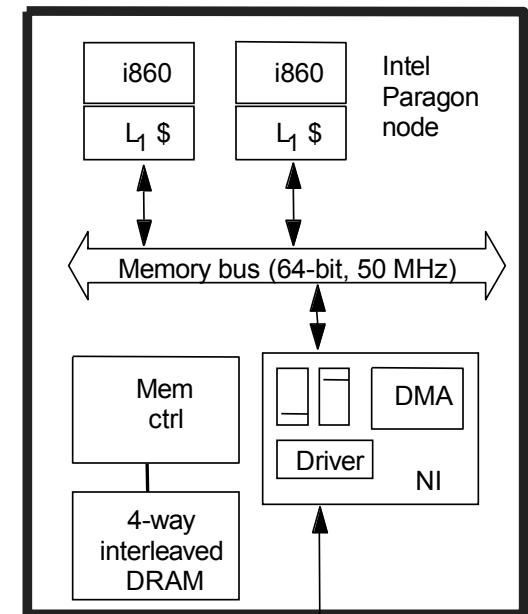
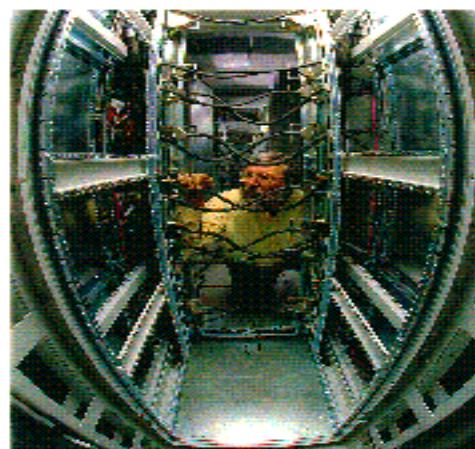
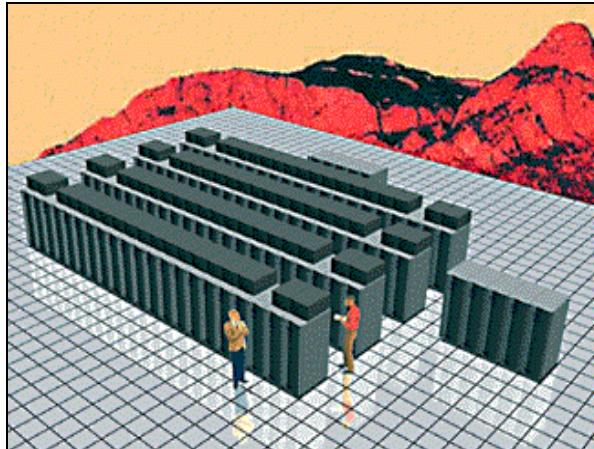
HyperMon Architecture (Who built this?)

- Develop hardware support for tracing
 - Reduces intrusion trace buffering and I/O
- Hardware design
 - Memory-mapped interface
 - Synchronized timers and automatic timestamping
 - Support for event bursts and off-line streaming



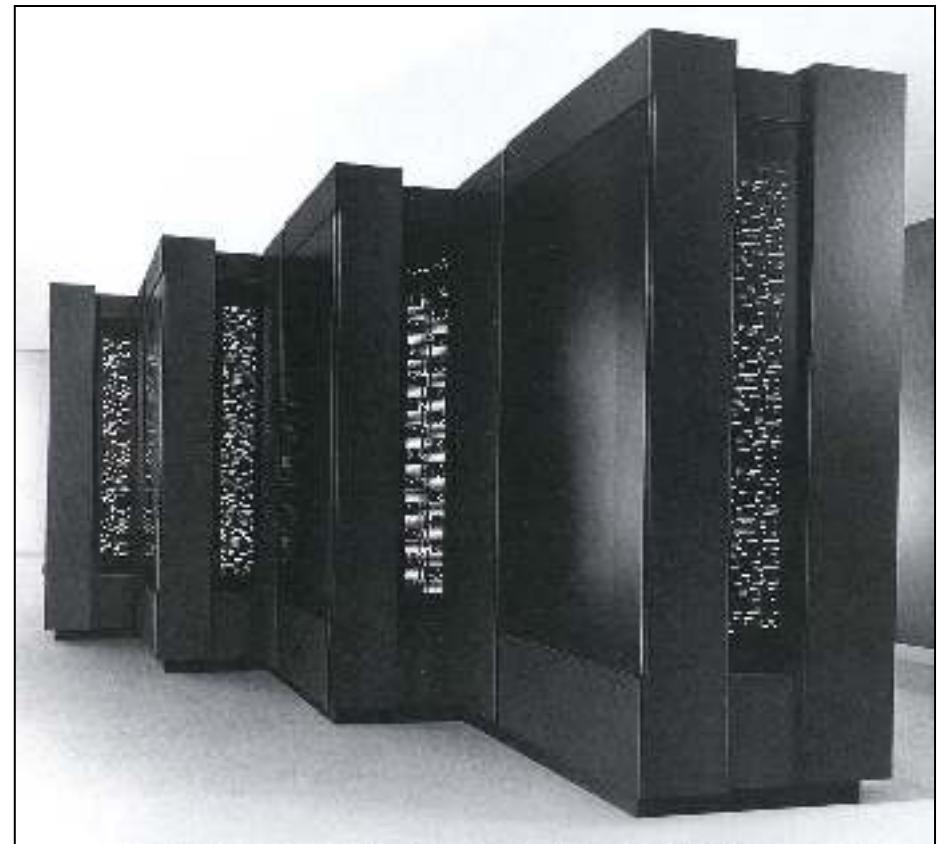
Intel Paragon and ASCI Red

- DARPA project machine
 - Intel i860 processor
 - 2D grid network with processor node attached to every switch
 - 8bit, 175 MHz bidirectional links
- Forerunner design for ASCI Red
 - First Teraflop computer



Thinking Machine CM-5

- Repackaged SparcStation
 - 4 per board
- Fat-Tree network
- Control network for global synchronization
- Suffered from hardware design and installation problems



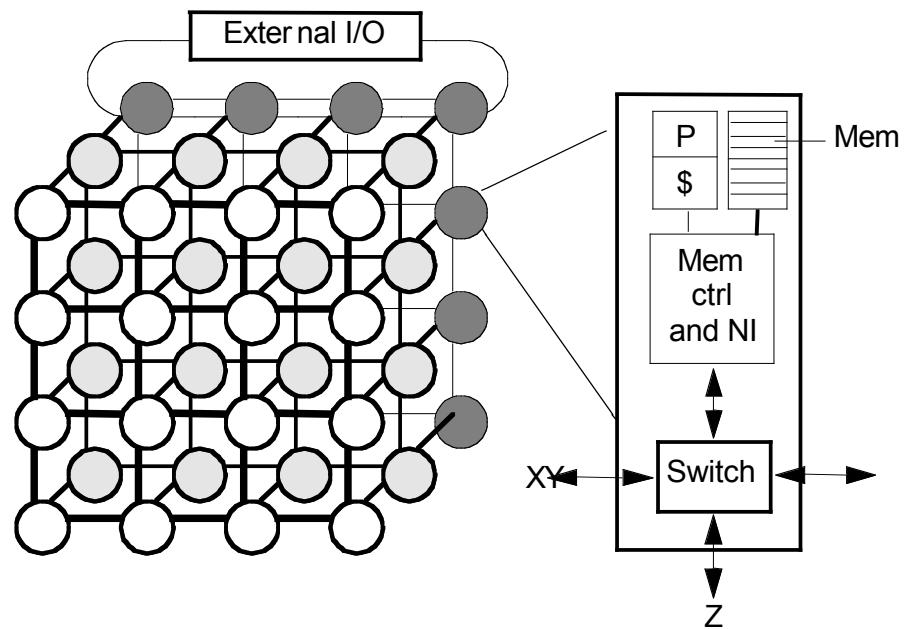
Berkeley Network Of Workstations (NOW)

- 100 Sun Ultra2 workstations
- Intelligent network interface
 - proc + mem
- Myrinet network
 - 160 MB/s per link
 - 300 ns per hop



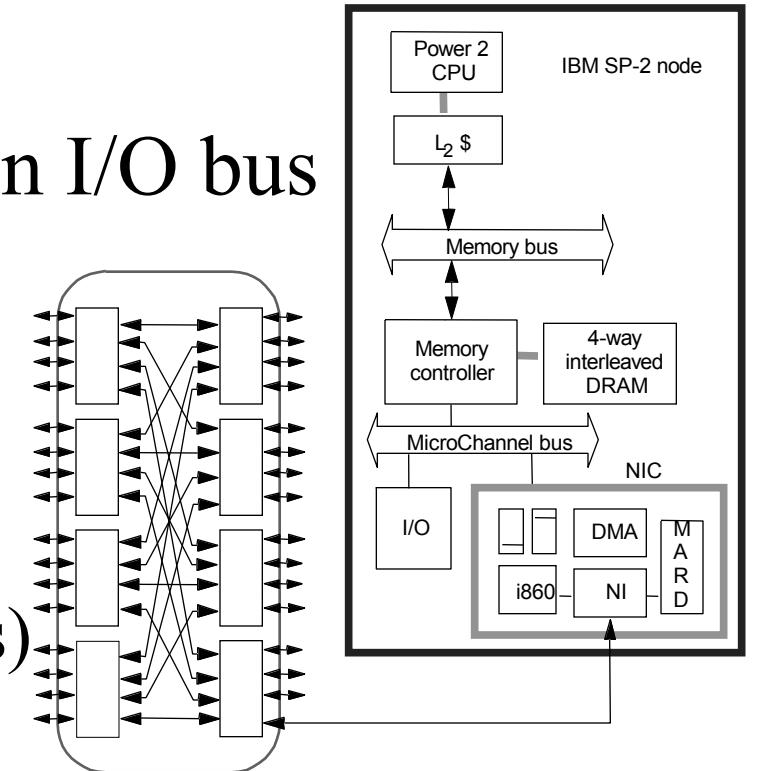
Cray T3E

- Up to 1024 nodes
- 3D torus network
 - 480 MB/s links
- No memory coherence
- Access remote memory
 - Converted to messages
 - SHared MEMory communication
 - ◆ *put / get* operations
- Very successful machine



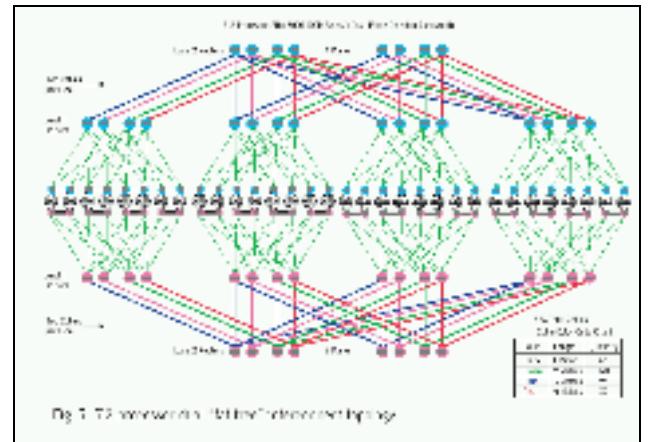
IBM SP-2

- Made out of essentially complete RS6000 workstations
- Network interface integrated in I/O bus
- SP network very advanced
 - Formed from 8-port switches
- Predecessor design to
 - ASCI Blue Pacific (5856 CPUs)
 - ASCI White (8192 CPUs)



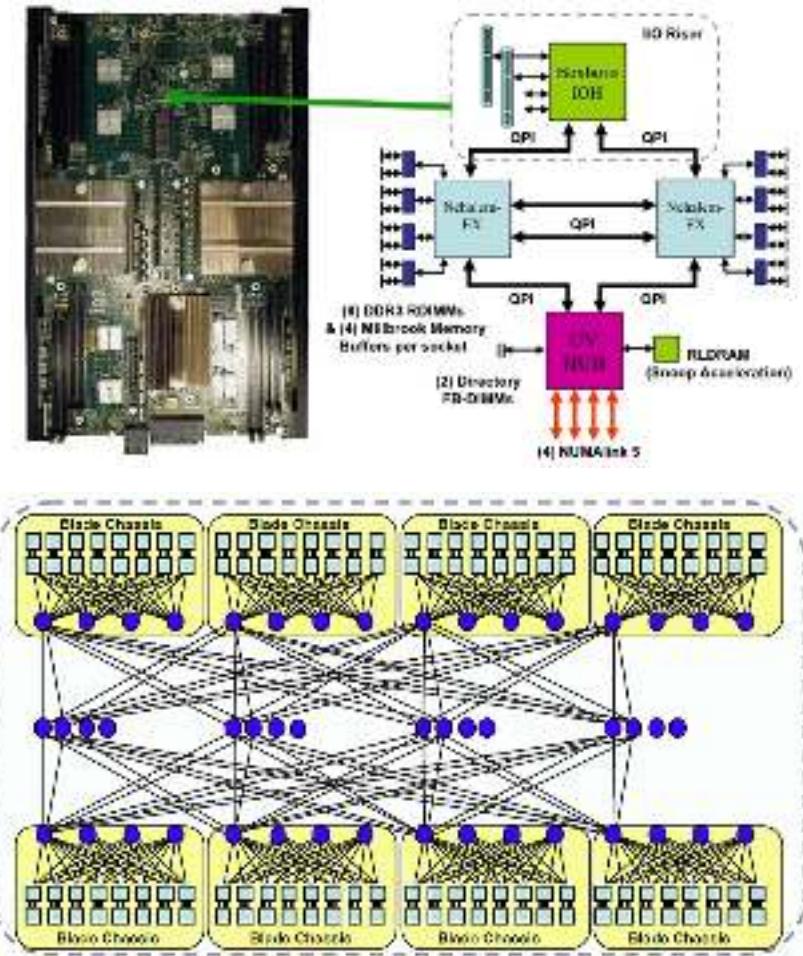
NASA Columbia

- System hardware
 - 20 SGI Altix 3700 superclusters
 - ◆ 512 Itanium2 processors (1.5 GHz)
 - ◆ 1 TB memory
 - 10,240 processors (now 13,312)
 - NUMAflex architecture
 - NUMAlink “fat tree” network
 - Fully shared memory!!!
- Software
 - Linux with PBS Pro job scheduling
 - Intel Fortran/C/C++ compilers



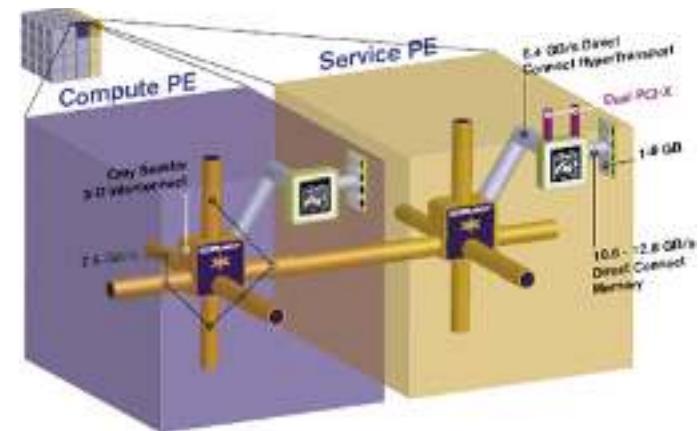
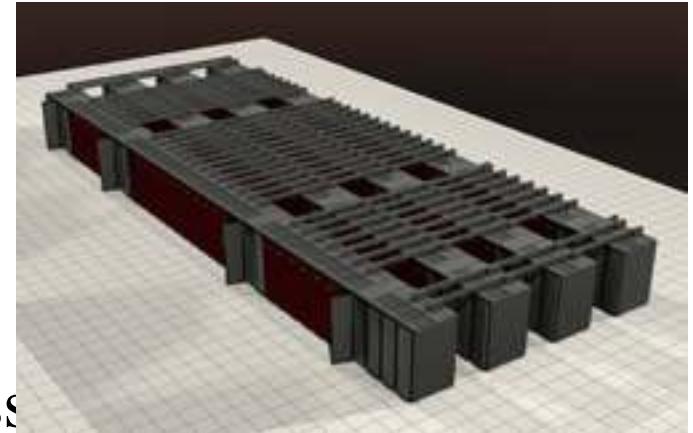
SGI Altix UV

- Latest generation scalable shared memory architecture
- Scaling from 32 to 2,048 cores
 - Intel Nehalem EX
- Architectural provisioning for up to 262,144 cores
- Up to 16 terabytes of global shared memory in a single system image (SSI)
- High-speed 15GB per second interconnect NUMALink 5



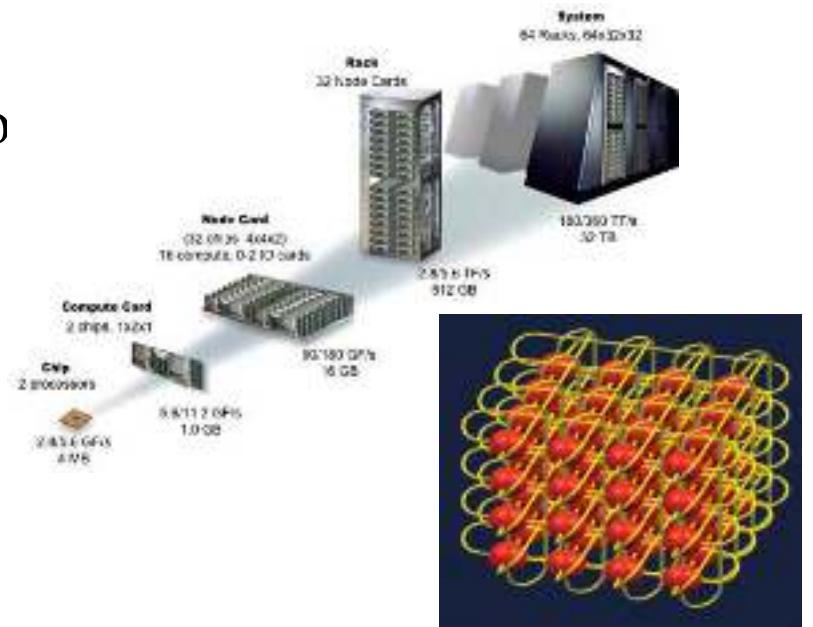
Sandia Red Storm

- System hardware
 - Cray XT3
 - 135 compute node cabinets
 - 12,960 processors
 - ◆ AMD Opteron dual-core
 - 320 / 320 service / I/O node processes
 - 40 TB memory
 - 340 TB disk
 - 3D mesh interconnect
- Software
 - Catamount compute node kernel
 - Linux I/O node kernel
 - MPI



LLNL BG/L

- System hardware
 - IBM BG/L (BlueGene)
 - 65,536 dual-processor compute nodes
 - ◆ PowerPC processors
 - ◆ “double hummer” floating point
 - I/O node per 32 compute nodes
 - 32x32x64 3D torus network
 - Global reduction tree
 - Global barrier and interrupt netwo
 - Scalable tree network for I/O
- Software
 - Compute node kernel (CNK)
 - Linux I/O node kernel (ION)
 - MPI
 - Different operating modes



Tokyo Institute of Technology TSUBAME

- System hardware
 - 655 Sun Fire X4600 servers
 - 11,088 processors
 - ◆ AMD Opteron dual-core
 - ClearSpeed accelerator
 - InfiniBand network
 - 21 TB memory
 - 42 Sun Fire X4500 servers
 - 1 PB of storage space
- Software
 - SuSE Linux Enterprise Server 9 SP3
 - Sun N1 Grid Engine 6.0
 - Lustre Client Software

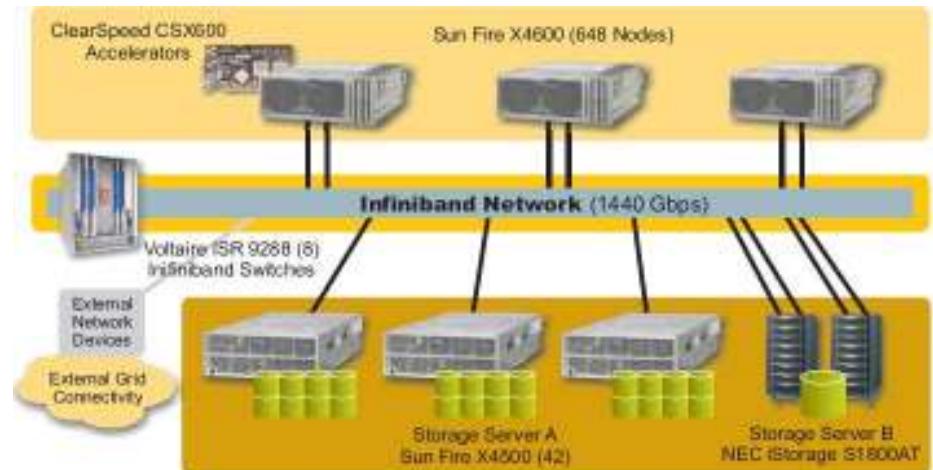


Figure 1. The TSUBAME grid system architecture

Tokyo Institute of Technology TSUBAME2



Thin Node 1408 nodes



HP ProLiant SL390s

GPU : NVIDIA Tesla M2050(Fermi Core)x3 515GFLOPS VRAM 3GB/GPU
CPU : Intel Xeon X5670 2.93GHz x2
6 core/socket 76.7 GFLOPS(12cores/node) ≫ Turbo boost: 3.196GHz
Memory : 58GB DDR3 1333MHz (partly 103GB)
SSD : 60GB x2 (120GB/node) (partly 120GB x2 (240GB/node))

Medium Node 24 nodes



HP ProLiant DL580 G7

CPU: Intel Xeon X7550 (Nehalem-EX)
2.0 GHz x4 sockets (32cores/node)
GPU: NVIDIA Tesla S1070
Memory: 137 GB (DDR3 1066MHz)
SSD: 120GB x4 (480GB/node)
Infiniband: QDR

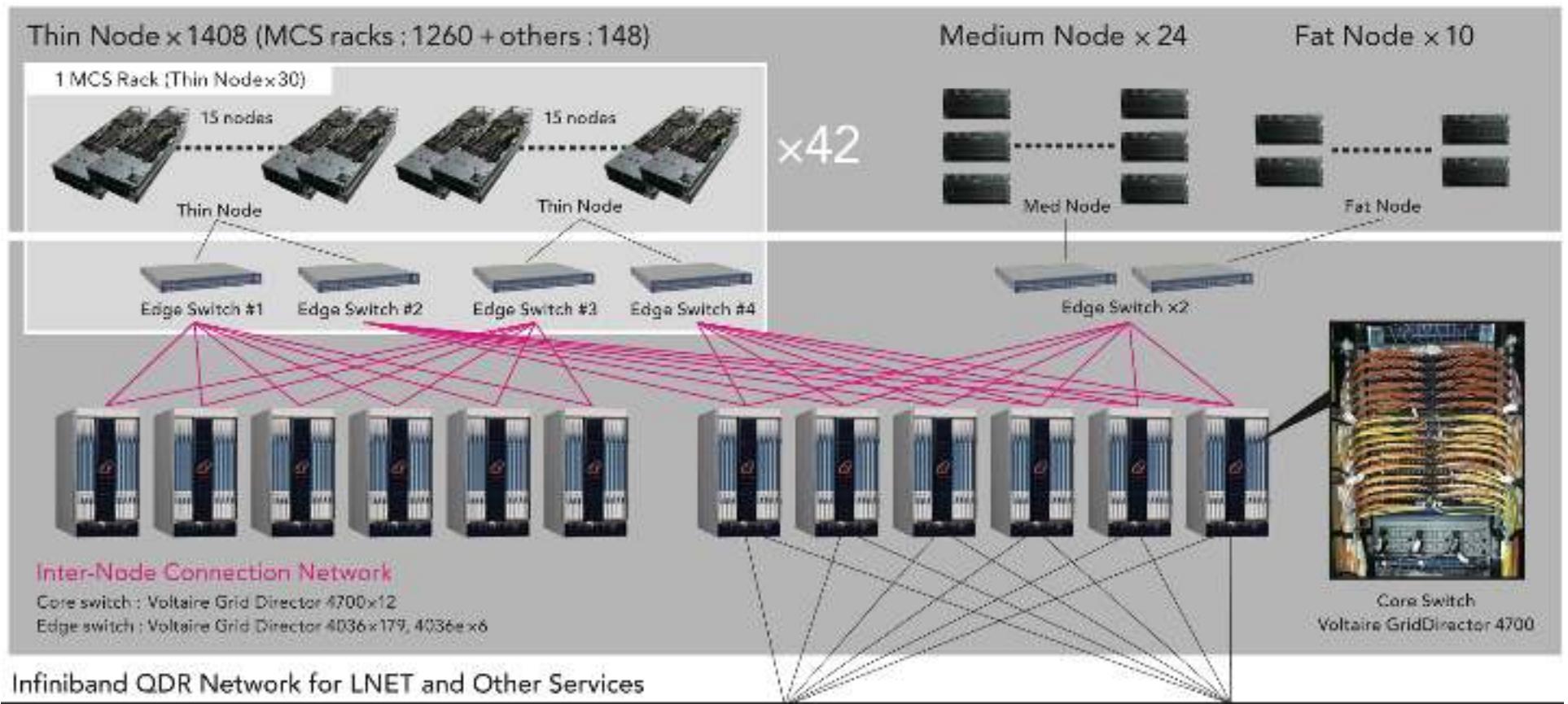
Fat Node 10 nodes



HP ProLiant DL580 G7

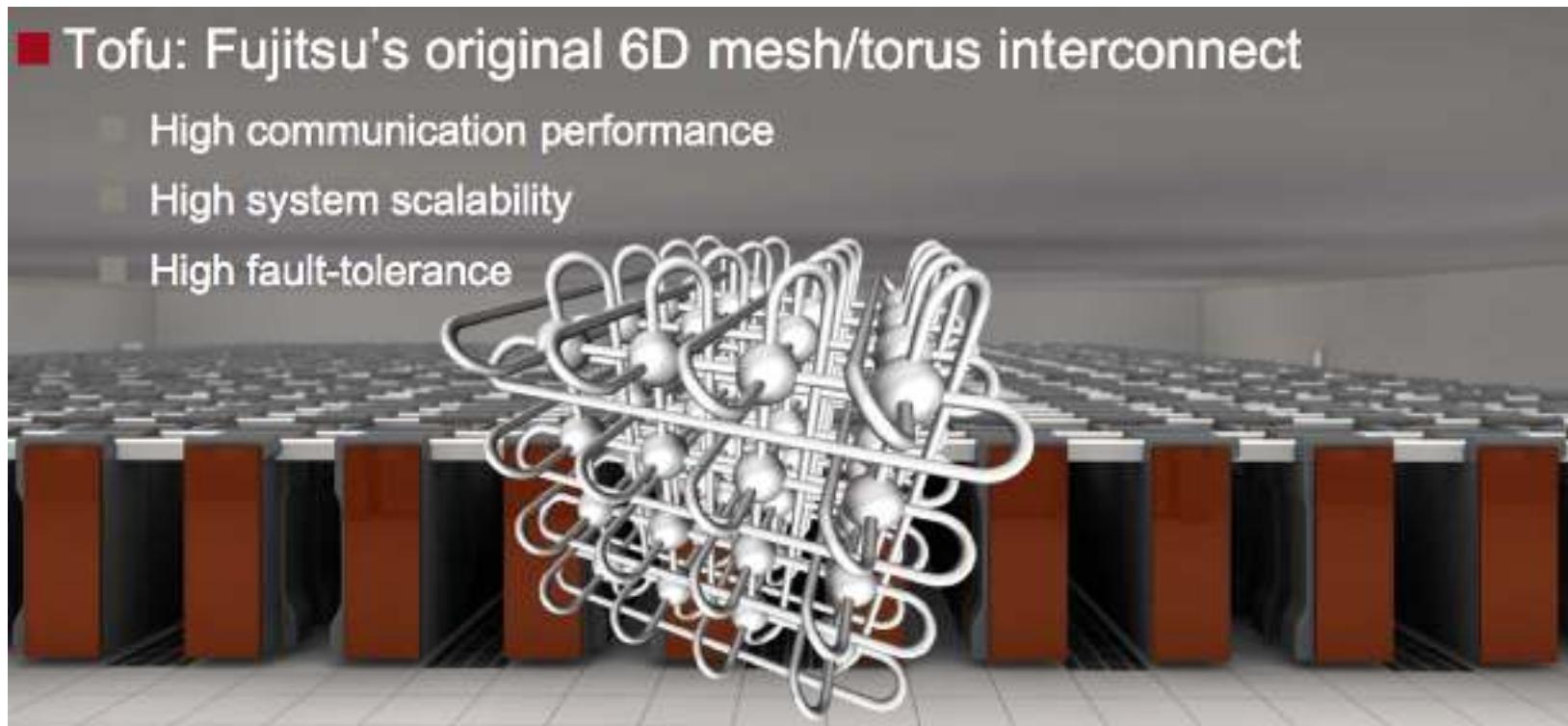
CPU: Intel Xeon X7550 (Nehalem-EX)
2.0 GHz x4 sockets (32cores/node)
GPU: NVIDIA Tesla S1070
Memory: 274 GB (8 nodes),
548 GB (2 nodes)
DDR3 1066MHz
SSD: 120GB x5 (600GB/node)
Infiniband: QDR

TSUBAME2 – Interconnect



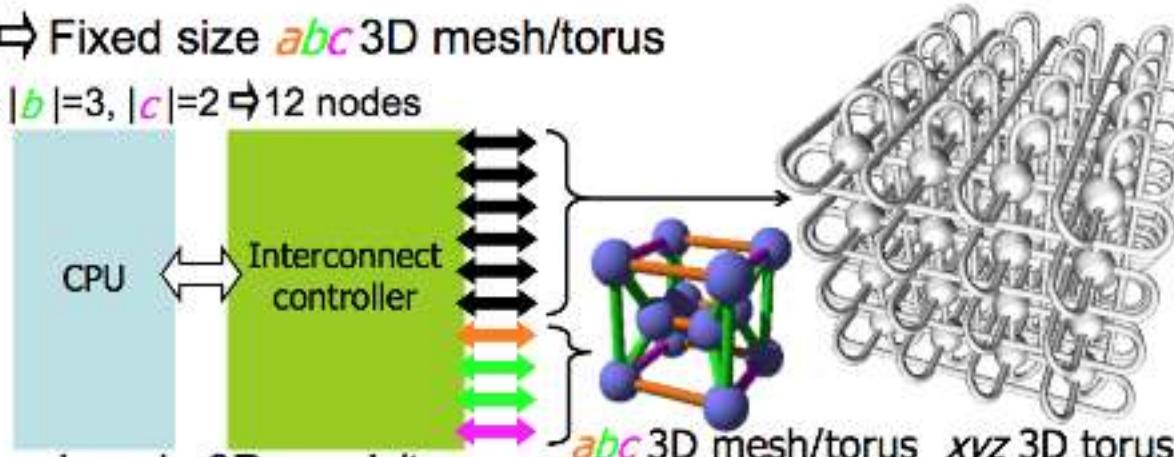
Japanese K Computer – Interconnect

- 80,000 CPUs (SPARC64 VIIIfx), 640,000 cores
- 800 racks
- 8.6 Petaflops (Linpack)



Japanese K Computer – Interconnect

- 6 links \Rightarrow Scalable xyz 3D torus
- 4 links \Rightarrow Fixed size abc 3D mesh/torus
 - $|a|=2, |b|=3, |c|=2 \Rightarrow 12$ nodes

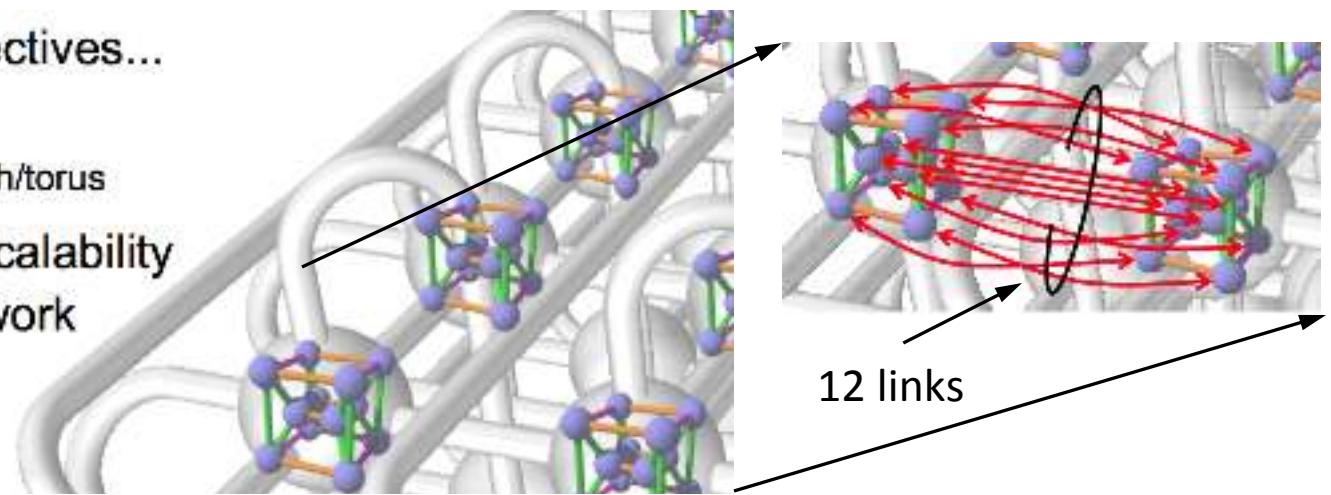


- Total topology is 6D mesh/torus
 - Cartesian product of xyz and abc mesh/torus

- From the other perspectives...

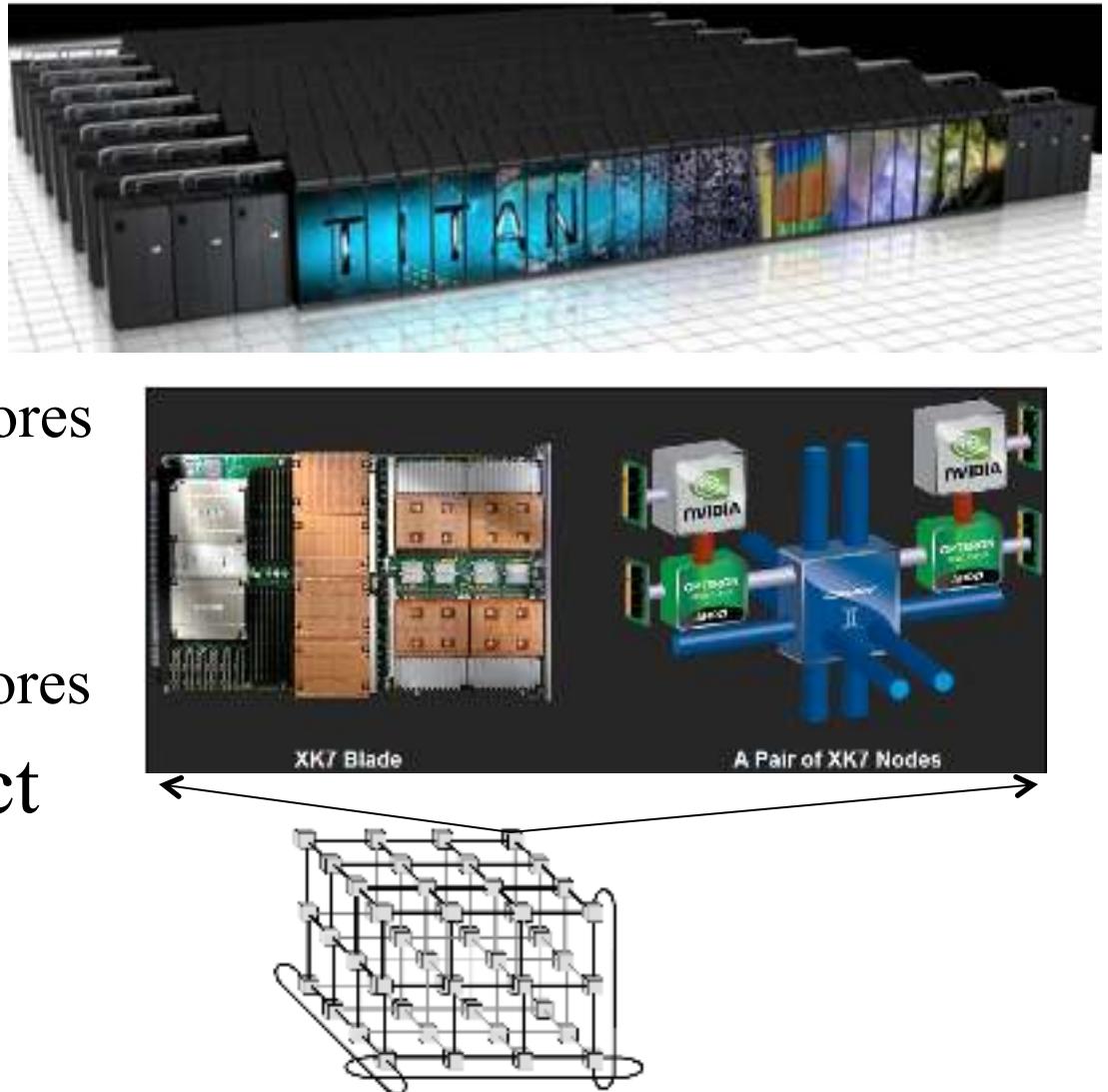
- Overlaid twelve xyz torus
 - $X \times Y \times Z$ array of abc mesh/torus

- Twelve times higher scalability than the 3D torus network



ORNL Titan (<http://www.olcf.ornl.gov/titan>)

- Cray XK7
 - 18,688 nodes
 - AMD Opteron
 - ◆ 16-core Interlagos
 - ◆ 299,008 Opteron cores
 - NVIDIA K20x
 - ◆ 18,688 GPUs
 - ◆ 50,233,344 GPU cores
- Gemini interconnect
 - 3D torus
- 20+ petaflops



Next Class

- Parallel performance models



Parallel Performance Theory - 1

Parallel Computing

CIS 410/510

Department of Computer and Information Science



UNIVERSITY OF OREGON

Outline

- Performance scalability
- Analytical performance measures
- Amdahl's law and Gustafson-Barsis' law

What is Performance?

- In computing, performance is defined by 2 factors
 - Computational requirements (what needs to be done)
 - Computing resources (what it costs to do it)
- Computational problems translate to requirements
- Computing resources interplay and tradeoff

$$\textit{Performance} \sim \frac{1}{\textit{Resources for solution}}$$



Hardware



Time



Energy

... and ultimately



Money

Why do we care about Performance?

- Performance itself is a measure of how well the computational requirements can be satisfied
- We evaluate performance to understand the relationships between requirements and resources
 - Decide how to change “solutions” to target objectives
- Performance measures reflect decisions about how and how well “solutions” are able to satisfy the computational requirements

“The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible.”
Charles Babbage, 1791 – 1871

What is Parallel Performance?

- Here we are concerned with performance issues when using a parallel computing environment
 - Performance with respect to parallel computation
- Performance is the *raison d'être* for parallelism
 - Parallel performance versus sequential performance
 - If the “performance” is not better, parallelism is not necessary
- *Parallel processing* includes techniques and technologies necessary to compute in parallel
 - Hardware, networks, operating systems, parallel libraries, languages, compilers, algorithms, tools, ...
- Parallelism must deliver performance
 - How? How well?

Performance Expectation (Loss)

- If each processor is rated at k MFLOPS and there are p processors, should we see $k*p$ MFLOPS performance?
- If it takes 100 seconds on 1 processor, shouldn't it take 10 seconds on 10 processors?
- Several causes affect performance
 - Each must be understood separately
 - But they interact with each other in complex ways
 - ◆ Solution to one problem may create another
 - ◆ One problem may mask another
- Scaling (system, problem size) can change conditions
- Need to understand *performance space*

Embarrassingly Parallel Computations

- An embarrassingly parallel computation is one that can be obviously divided into completely independent parts that can be executed simultaneously
 - In a truly embarrassingly parallel computation there is no interaction between separate processes
 - In a nearly embarrassingly parallel computation results must be distributed and collected/combined in some way
- Embarrassingly parallel computations have potential to achieve maximal speedup on parallel platforms
 - If it takes T time sequentially, there is the potential to achieve T/P time running in parallel with P processors
 - What would cause this not to be the case always?

Scalability

- A program can scale up to use many processors
 - What does that mean?
- How do you evaluate scalability?
- How do you evaluate scalability goodness?
- Comparative evaluation
 - If double the number of processors, what to expect?
 - Is scalability linear?
- Use parallel efficiency measure
 - Is efficiency retained as problem size increases?
- Apply performance metrics

Performance and Scalability

□ Evaluation

- *Sequential* runtime (T_{seq}) is a function of
 - ◆ problem size and architecture
- *Parallel* runtime (T_{par}) is a function of
 - ◆ problem size and parallel architecture
 - ◆ # processors used in the execution
- Parallel performance affected by
 - ◆ algorithm + architecture

□ Scalability

- Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem

Performance Metrics and Formulas

- T_1 is the execution time on a single processor
- T_p is the execution time on a p processor system
- $S(p)$ (S_p) is the *speedup*

$$S(p) = \frac{T_1}{T_p}$$

- $E(p)$ (E_p) is the *efficiency*

$$\text{Efficiency} = \frac{S_p}{p}$$

- $\text{Cost}(p)$ (C_p) is the *cost*

$$\text{Cost} = p \times T_p$$

- Parallel algorithm is *cost-optimal*
 - *Parallel time = sequential time* ($C_p = T_1$, $E_p = 100\%$)

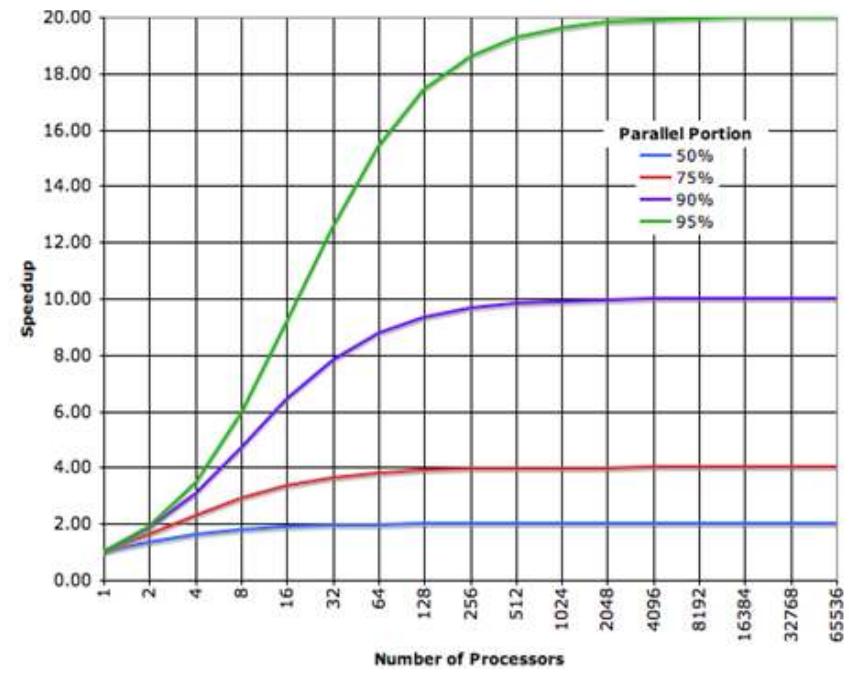
Amdahl's Law (Fixed Size Speedup)

- Let f be the fraction of a program that is sequential
 - $1-f$ is the fraction that can be parallelized
- Let T_1 be the execution time on 1 processor
- Let T_p be the execution time on p processors
- S_p is the *speedup*

$$\begin{aligned} S_p &= T_1 / T_p \\ &= T_1 / (fT_1 + (1-f)T_1/p)) \\ &= 1 / (f + (1-f)/p)) \end{aligned}$$

- As $p \rightarrow \infty$

$$S_p = 1/f$$



Amdahl's Law and Scalability

- Scalability
 - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem
- When does Amdahl's Law apply?
 - When the problem size is fixed
 - *Strong scaling* ($p \rightarrow \infty$, $S_p = S_\infty \rightarrow 1/f$)
 - Speedup bound is determined by the degree of sequential execution time in the computation, not # processors!!!
 - Uhh, this is not good ... Why?
 - Perfect efficiency is hard to achieve
- See original paper by Amdahl on webpage

Gustafson-Barsis' Law (Scaled Speedup)

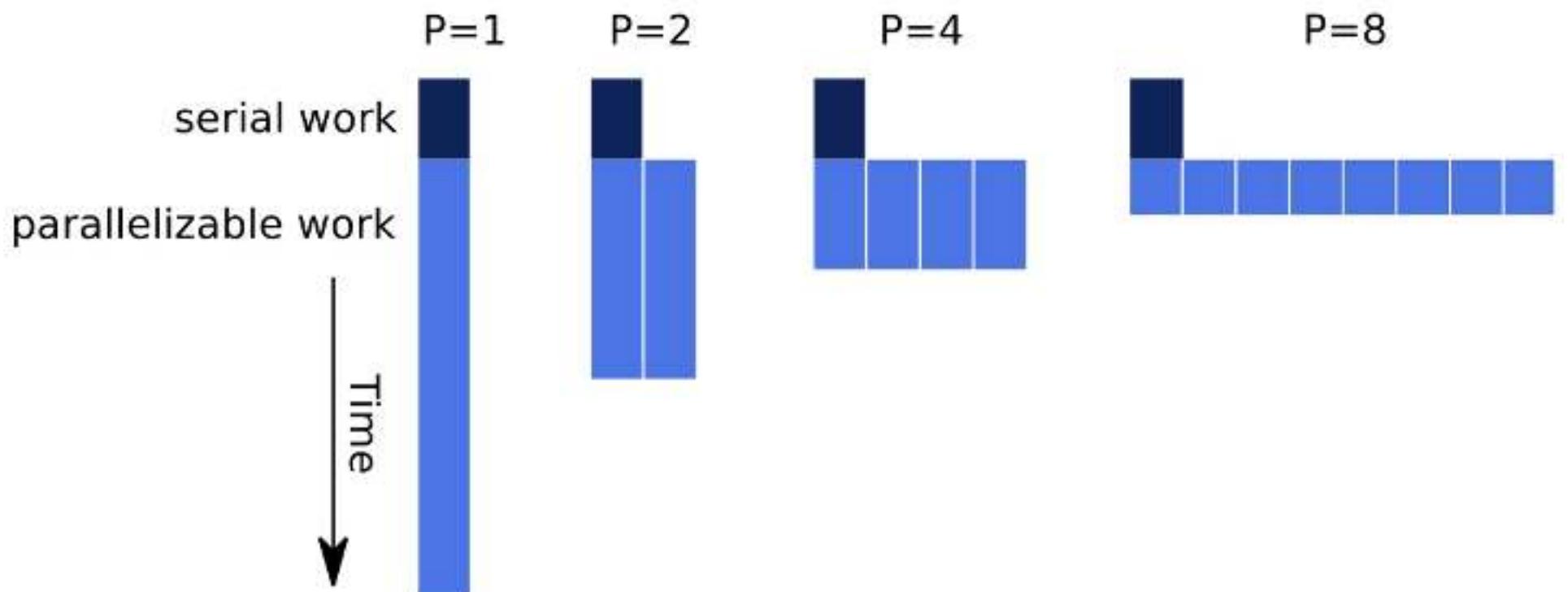
- Often interested in larger problems when scaling
 - How big of a problem can be run (HPC Linpack)
 - Constrain problem size by parallel time
- Assume parallel time is kept constant
 - $T_p = C = (f + (1-f)) * C$
 - f_{seq} is the fraction of T_p spent in sequential execution
 - f_{par} is the fraction of T_p spent in parallel execution
- What is the execution time on one processor?
 - Let $C=1$, then $T_s = f_{seq} + p(1 - f_{seq}) = 1 + (p-1)f_{par}$
- What is the speedup in this case?
 - $S_p = T_s / T_p = T_s / 1 = f_{seq} + p(1 - f_{seq}) = 1 + (p-1)f_{par}$

Gustafson-Barsis' Law and Scalability

- Scalability
 - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem
- When does Gustafson's Law apply?
 - When the problem size can increase as the number of processors increases
 - *Weak scaling* ($S_p = 1 + (p-1)f_{par}$)
 - Speedup function includes the number of processors!!!
 - Can maintain or increase parallel efficiency as the problem scales
- See original paper by Gustafson on webpage

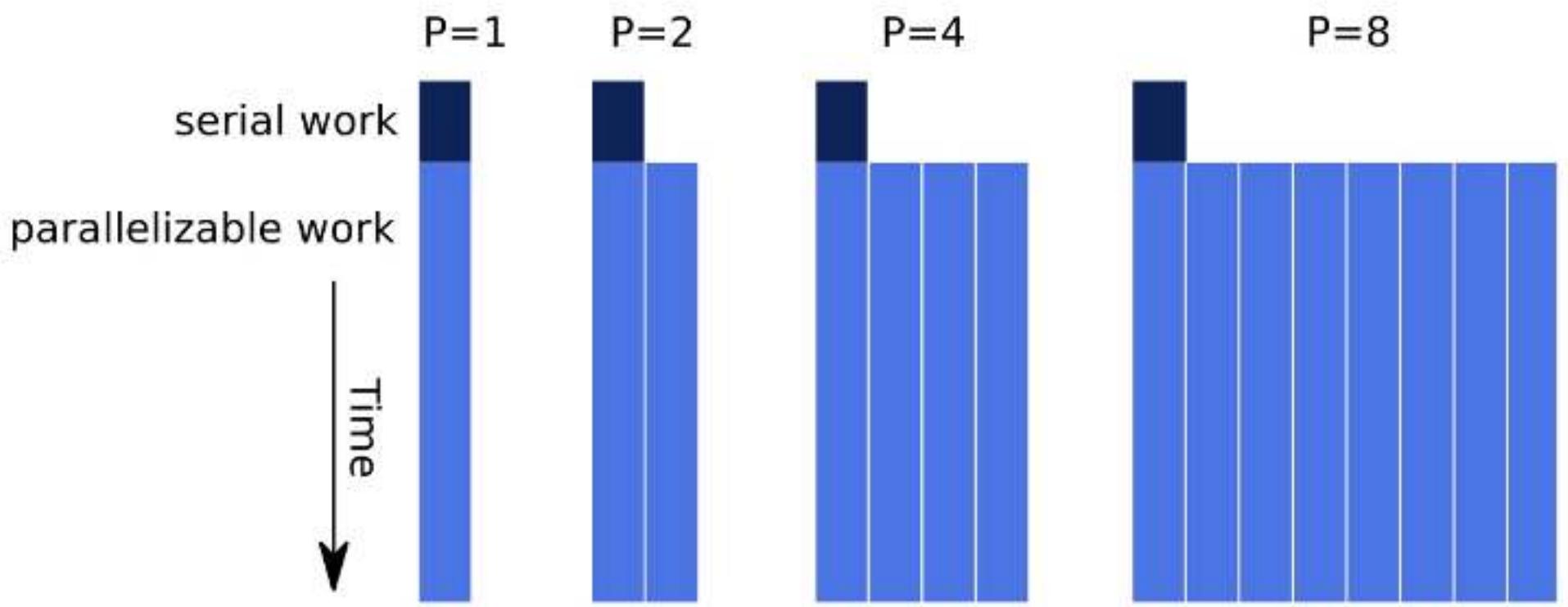
Amdahl versus Gustafson-Baris

Amdahl



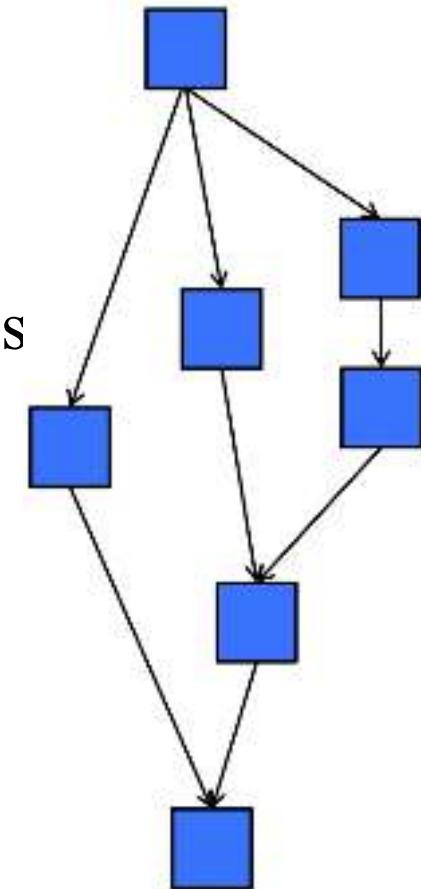
Amdahl versus Gustafson-Baris

Gustafson-Baris



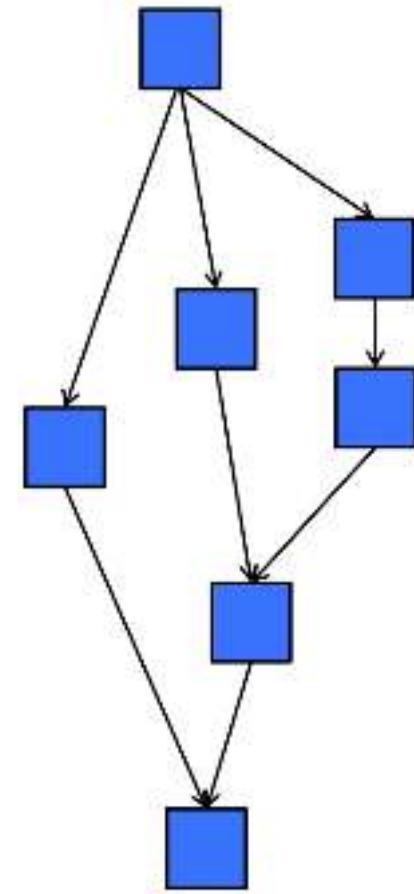
DAG Model of Computation

- Think of a program as a directed acyclic graph (DAG) of tasks
 - A task can not execute until all the inputs to the tasks are available
 - These come from outputs of earlier executing tasks
 - DAG shows explicitly the task dependencies
- Think of the hardware as consisting of workers (processors)
- Consider a *greedy* scheduler of the DAG tasks to workers
 - No worker is idle while there are tasks still to execute



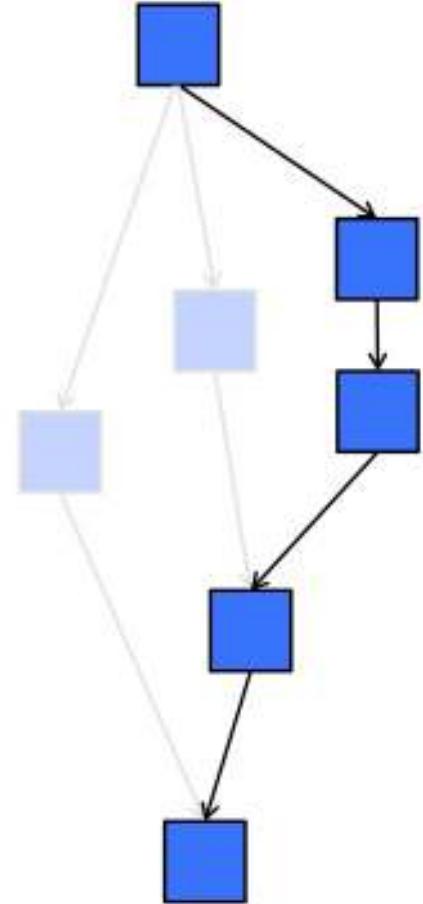
Work-Span Model

- T_P = time to run with P workers
- T_1 = *work*
 - Time for serial execution
 - ◆ execution of all tasks by 1 worker
 - Sum of all work
- T_∞ = *span*
 - Time along the *critical path*
- Critical path
 - Sequence of task execution (path) through DAG that takes the longest time to execute
 - Assumes an infinite # workers available



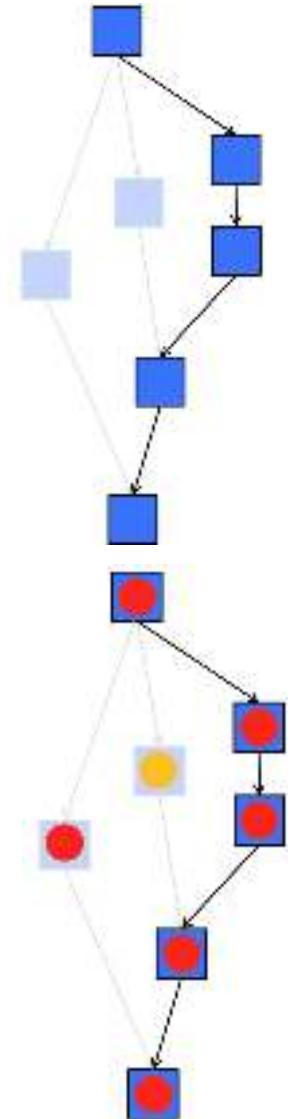
Work-Span Example

- Let each task take 1 unit of time
- DAG at the right has 7 tasks
- $T_1 = 7$
 - All tasks have to be executed
 - Tasks are executed in a serial order
 - Can they execute in any order?
- $T_\infty = 5$
 - Time along the *critical path*
 - In this case, it is the longest pathlength of any task order that maintains necessary dependencies



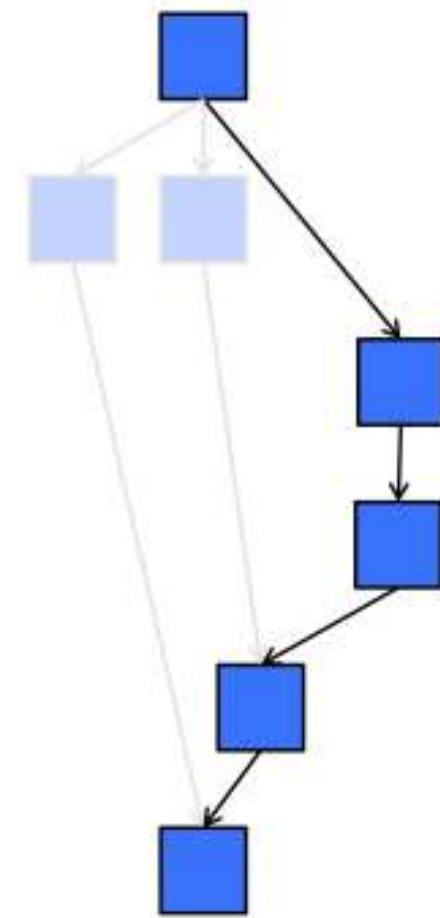
Lower/Upper Bound on Greedy Scheduling

- Suppose we only have P workers
- We can write a work-span formula to derive a lower bound on T_P
 - $\text{Max}(T_I / P, T_\infty) \leq T_P$
- T_∞ is the best possible execution time
- Brent's Lemma derives an upper bound
 - Capture the additional cost executing the other tasks not on the critical path
 - Assume can do so without overhead
 - $T_P \leq (T_I - T_\infty) / P + T_\infty$

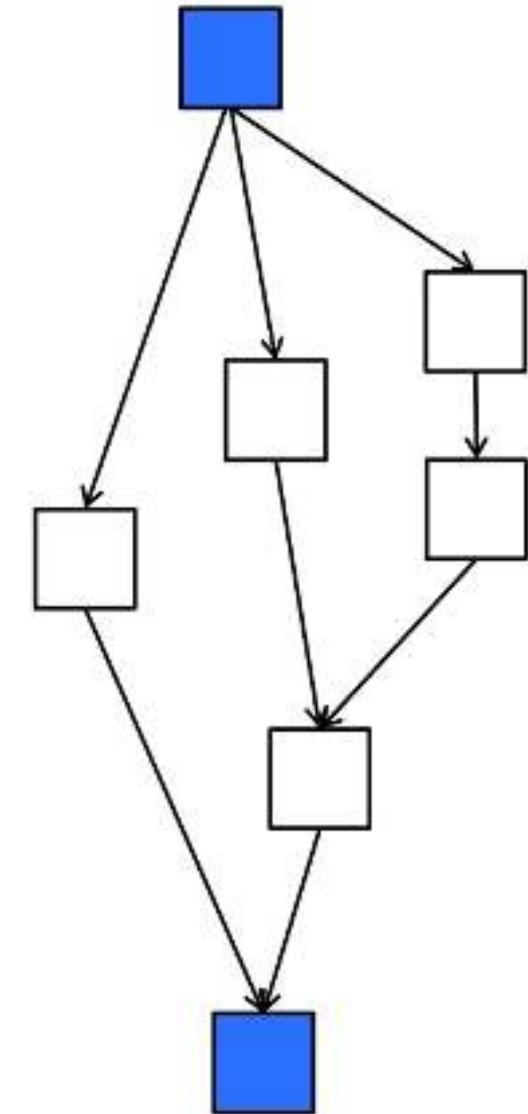
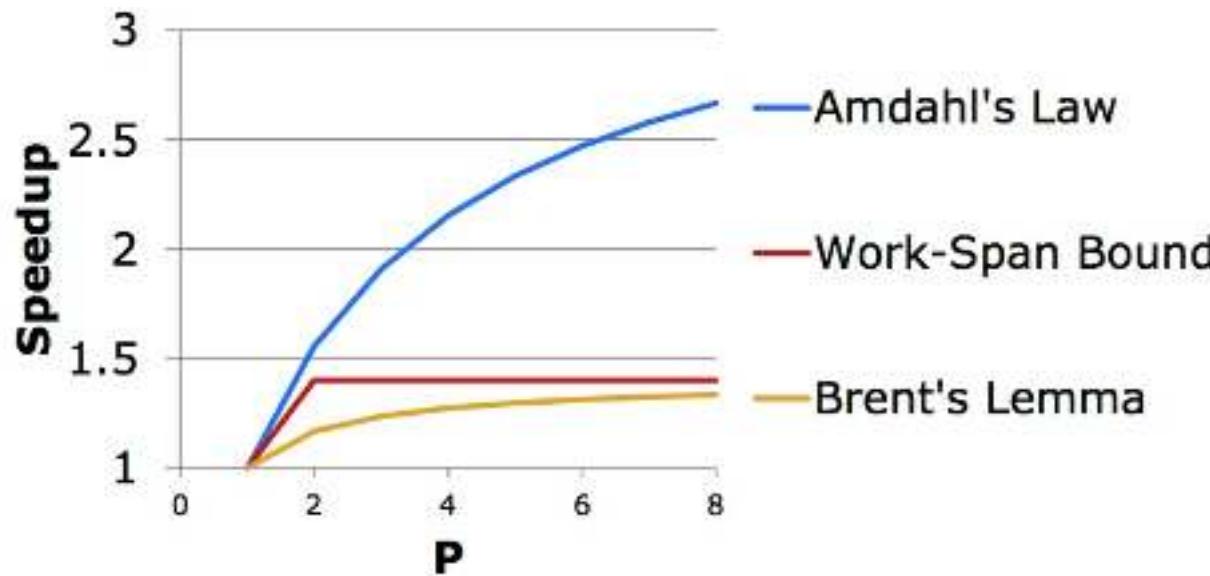


Consider Brent's Lemma for 2 Processors

- $T_1 = 7$
- $T_\infty = 5$
- $$\begin{aligned} T_2 &\leq (T_1 - T_\infty) / P + T_\infty \\ &\leq (7 - 5) / 2 + 5 \\ &\leq 6 \end{aligned}$$



Amdahl was an optimist!



Estimating Running Time

- Scalability requires that T_∞ be dominated by T_1

$$T_P \approx T_1 / P + T_\infty \text{ if } T_\infty \ll T_1$$

- Increasing work hurts parallel execution proportionately
- The span impacts scalability, even for finite P

Parallel Slack

- Sufficient parallelism implies linear speedup

$$T_P \approx T_1/P \quad \text{if} \quad T_1/T_\infty \gg P$$



Linear speedup



Parallel slack

Asymptotic Complexity

- Time complexity of an algorithm summarizes how the execution time grows with input size
- Space complexity summarizes how memory requirements grow with input size
- Standard work-span model considers only computation, not communication or memory
- Asymptotic complexity is a strong indicator of performance on large-enough problem sizes and reveals an algorithm's fundamental limits

Definitions for Asymptotic Notation

- Let $T(N)$ mean the execution time of an algorithm
- Big O notation
 - $T(N)$ is a member of $O(f(N))$ means that
 $T(N) \leq c \cdot f(N)$ for constant c
- Big Omega notation
 - $T(N)$ is a member of $\Omega(f(N))$ means that
 $T(N) \geq c \cdot f(N)$ for constant c
- Big Theta notation
 - $T(N)$ is a member of $\Theta(f(N))$ means that
 $c_1 \cdot f(n) \leq T(N) < c_2 \cdot f(N)$ for constants c_1 and c_2



Parallel Performance Theory - 2

Parallel Computing

CIS 410/510

Department of Computer and Information Science



UNIVERSITY OF OREGON

Outline

- Scalable parallel execution
- Parallel execution models
- Isoefficiency
- Parallel machine models
- Parallel performance engineering

Scalable Parallel Computing

- Scalability in parallel architecture
 - Processor numbers
 - Memory architecture
 - Interconnection network
 - Avoid critical architecture bottlenecks
- Scalability in computational problem
 - Problem size
 - Computational algorithms
 - ◆ Computation to memory access ratio
 - ◆ Computation to communication ratio
- Parallel programming models and tools
- Performance scalability

Why Aren't Parallel Applications Scalable?

- Sequential performance
- Critical Paths
 - Dependencies between computations spread across processors
- Bottlenecks
 - One processor holds things up
- Algorithmic overhead
 - Some things just take more effort to do in parallel
- Communication overhead
 - Spending increasing proportion of time on communication
- Load Imbalance
 - Makes all processor wait for the “slowest” one
 - Dynamic behavior
- Speculative loss
 - Do A and B in parallel, but B is ultimately not needed

Critical Paths

- Long chain of dependence
 - Main limitation on performance
 - Resistance to performance improvement
- Diagnostic
 - Performance stagnates to a (relatively) fixed value
 - Critical path analysis
- Solution
 - Eliminate long chains if possible
 - Shorten chains by removing work from critical path

Bottlenecks

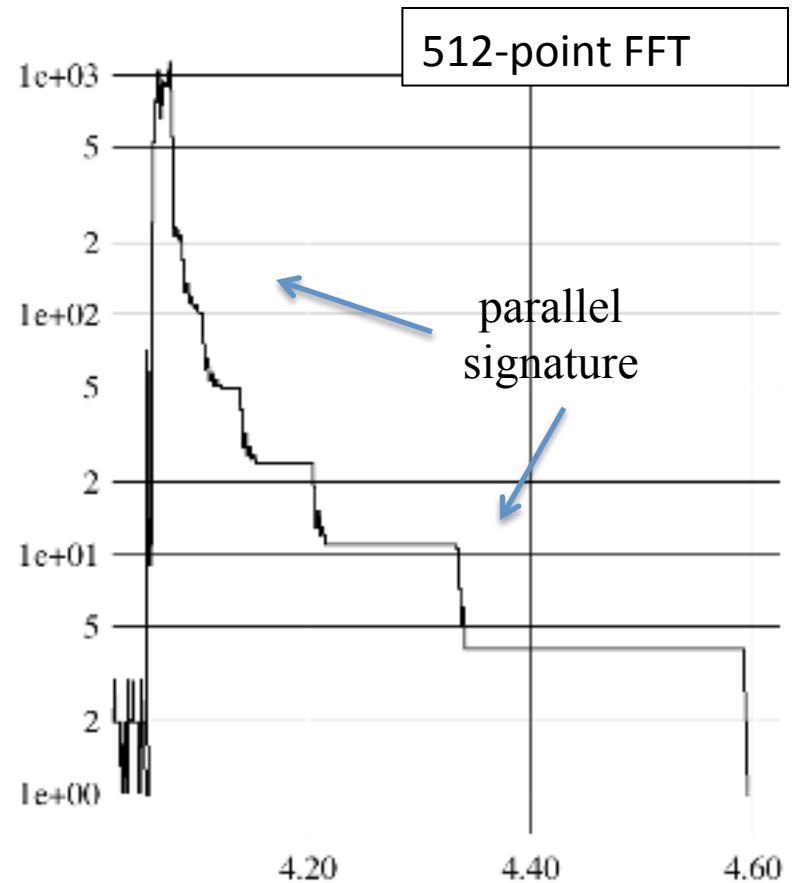
- How to detect?
 - One processor A is busy while others wait
 - Data dependency on the result produced by A
- Typical situations:
 - N-to-1 reduction / computation / 1-to-N broadcast
 - One processor assigning job in response to requests
- Solution techniques:
 - More efficient communication
 - Hierarchical schemes for master slave
- Program may not show ill effects for a long time
- Shows up when scaling

Algorithmic Overhead

- Different sequential algorithms to solve the same problem
- All parallel algorithms are sequential when run on 1 processor
- All parallel algorithms introduce addition operations (Why?)
 - *Parallel overhead*
- Where should be the starting point for a parallel algorithm?
 - Best sequential algorithm might not parallelize at all
 - Or, it doesn't parallelize well (e.g., not scalable)
- What to do?
 - Choose algorithmic variants that minimize overhead
 - Use two level algorithms
- Performance is the rub
 - Are you achieving better parallel performance?
 - Must compare with the best sequential algorithm

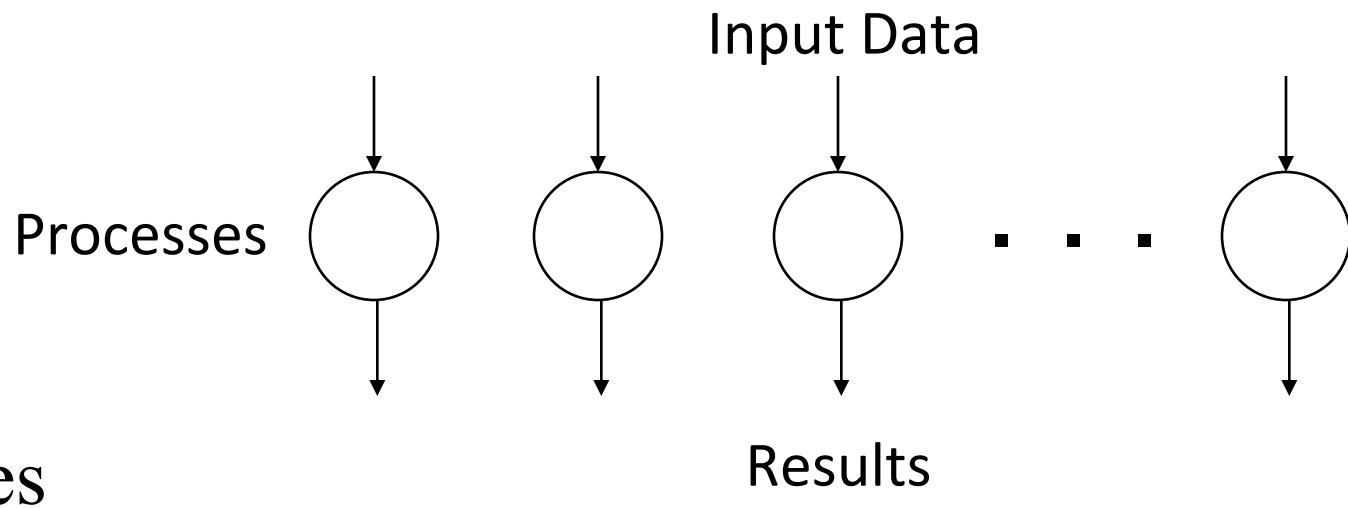
What is the maximum parallelism possible?

- Depends on application, algorithm, program
 - Data dependencies in execution
- Remember MaxPar
 - Analyzes the earliest possible “time” any data can be computed
 - Assumes a simple model for time it takes to execute instruction or go to memory
 - Result is the maximum parallelism available
- Parallelism varies!



Embarrassingly Parallel Computations

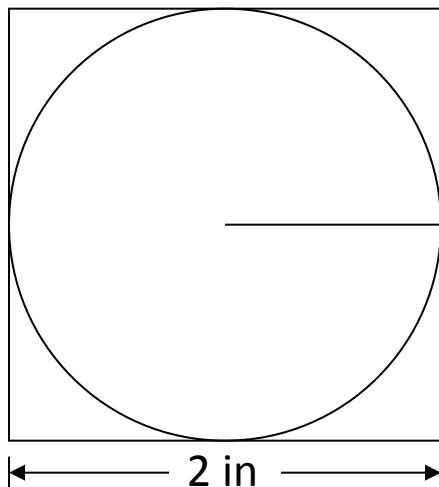
- No or very little communication between processes
- Each process can do its tasks without any interaction with other processes



- Examples
 - Numerical integration
 - Mandelbrot set
 - Monte Carlo methods

Calculating π with Monte Carlo

- Consider a circle of unit radius
- Place circle inside a square box with side of 2 in

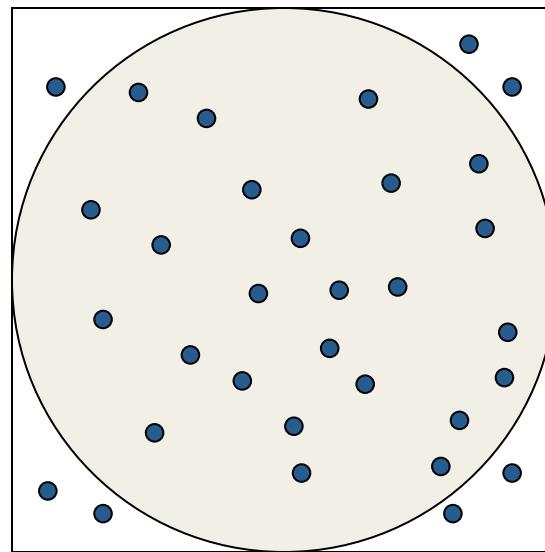


- The ratio of the circle area to the square area is:

$$\frac{\pi * 1 * 1}{2 * 2} = \frac{\pi}{4}$$

Monte Carlo Calculation of π

- Randomly choose a number of points in the square
- For each point p , determine if p is inside the circle
- The ratio of points in the circle to points in the square will give an approximation of $\pi/4$



Performance Metrics and Formulas

- T_1 is the execution time on a single processor
- T_p is the execution time on a p processor system
- $S(p)$ (S_p) is the *speedup*

$$S(p) = \frac{T_1}{T_p}$$

- $E(p)$ (E_p) is the *efficiency*

$$\text{Efficiency} = \frac{S_p}{p}$$

- $\text{Cost}(p)$ (C_p) is the *cost*

$$\text{Cost} = p \times T_p$$

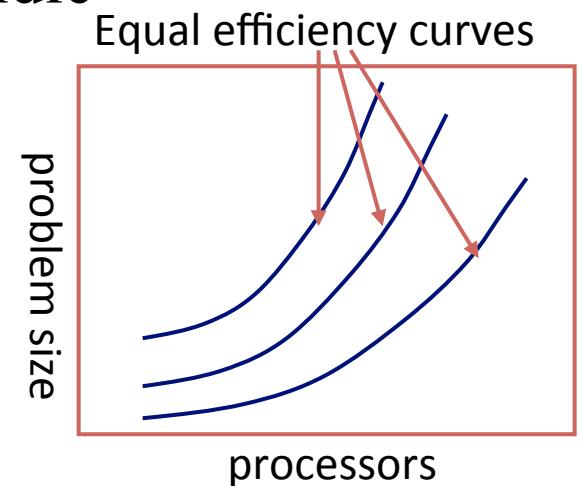
- Parallel algorithm is *cost-optimal*
 - *Parallel time = sequential time* ($C_p = T_1$, $E_p = 100\%$)

Analytical / Theoretical Techniques

- Involves simple algebraic formulas and ratios
 - Typical variables are:
 - ◆ data size (N), number of processors (P), machine constants
 - Want to model performance of individual operations, components, algorithms in terms of the above
 - ◆ be careful to characterize variations across processors
 - ◆ model them with max operators
 - Constants are important in practice
 - ◆ Use asymptotic analysis carefully
- Scalability analysis
 - Isoefficiency (Kumar)

Isoefficiency

- Goal is to quantify scalability
- How much increase in problem size is needed to retain the same efficiency on a larger machine?
- Efficiency
 - $T_I / (p * T_p)$
 - $T_p = \text{computation} + \text{communication} + \text{idle}$
- Isoefficiency
 - Equation for equal-efficiency curves
 - If no solution
 - ◆ problem is not scalable in the sense defined by isoefficiency
- See original paper by Kumar on webpage



Scalability of Adding n Numbers

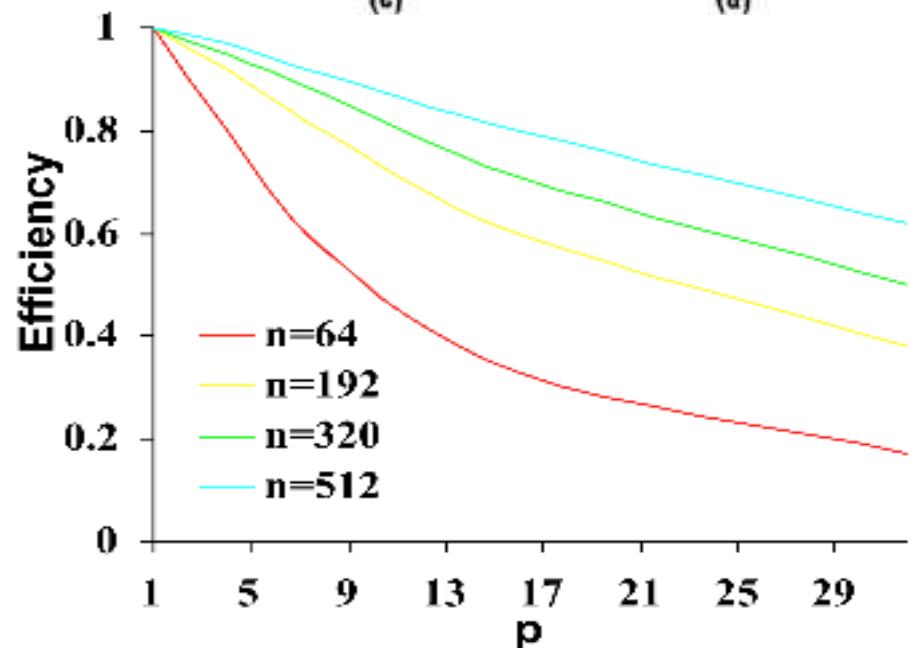
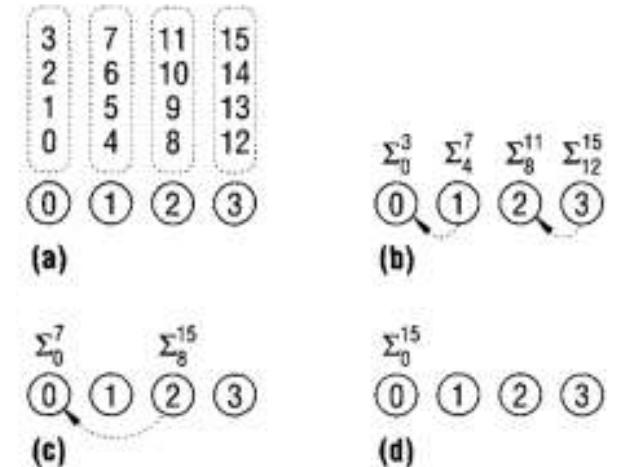
- Scalability of a parallel system is a measure of its capacity to increase speedup with more processors
- Adding n numbers on p processors with strip partition:

$$T_{par} = \frac{n}{p} - 1 + 2 \log p$$

$$\text{Speedup} = \frac{\frac{n-1}{n}}{\frac{n}{p} - 1 + 2 \log p}$$

$$\approx \frac{n}{\frac{n}{p} + 2 \log p}$$

$$\text{Efficiency} = \frac{S}{p} = \frac{n}{n + 2 p \log p}$$



Problem Size and Overhead

- Informally, problem size is expressed as a parameter of the input size
- A consistent definition of the size of the problem is the total number of basic operations (T_{seq})
 - Also refer to problem size as “work ($W = T_{seq}$)
- Overhead of a parallel system is defined as the part of the cost not in the best serial algorithm
- Denoted by T_O , it is a function of W and p

$$T_O(W,p) = pT_{par} - W \quad (pT_{par} \text{ includes overhead})$$

$$T_O(W,p) + W = pT_{par}$$

Isoefficiency Function

- With a fixed efficiency, W is as a function of p

$$T_{par} = \frac{W + T_o(W, p)}{p} \quad W = T_{seq}$$

$$\text{Speedup} = \frac{W}{T_{par}} = \frac{Wp}{W + T_o(W, p)}$$

$$\text{Efficiency} = \frac{S}{p} = \frac{W}{W + T_o(W, p)} = \frac{1}{1 + \frac{T_o(W, p)}{W}}$$

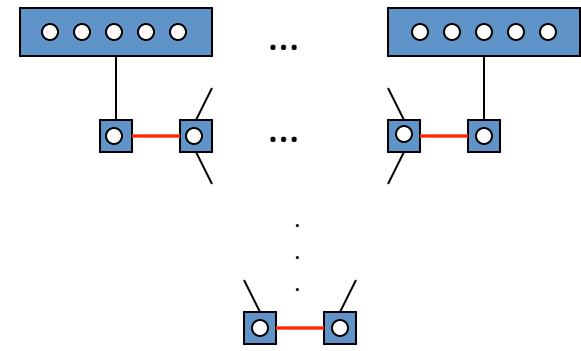
$$E = \frac{1}{1 + \frac{T_o(W, p)}{W}} \rightarrow \frac{T_o(W, p)}{W} = \frac{1 - E}{E}$$

$$W = \frac{E}{1 - E} T_o(W, p) = K T_o(W, p) \quad \text{Isoefficiency Function}$$

Isoefficiency Function of Adding n Numbers

- Overhead function:

- $T_O(W,p) = pT_{par} - W = 2p\log(p)$



- Isoefficiency function:

- $W=K*2p\log(p)$

- If p doubles, W needs also to be doubled to roughly maintain the same efficiency
- Isoefficiency functions can be more difficult to express for more complex algorithms

More Complex Isoefficiency Functions

- A typical overhead function T_O can have several distinct terms of different orders of magnitude with respect to both p and W
- We can balance W against each term of T_O and compute the respective isoefficiency functions for individual terms
 - Keep only the term that requires the highest grow rate with respect to p
 - This is the asymptotic isoefficiency function

Isoefficiency

- Consider a parallel system with an overhead function

$$T_O = p^{3/2} + p^{3/4}W^{3/4}$$

- Using only the first term

$$W = Kp^{3/2}$$

- Using only the second term

$$W = Kp^{3/4}W^{3/4}$$

$$W^{1/4} = Kp^{3/4}$$

$$W = K^4 p^3$$

- $K^4 p^3$ is the overall asymptotic isoeficiency function

Parallel Computation (Machine) Models

- PRAM (parallel RAM)
 - Basic parallel machine
- BSP (Bulk Synchronous Parallel)
 - Isolates regions of computation from communication
- LogP
 - Used for studying distribute memory systems
 - Focuses on the interconnection network
- Roofline
 - Based in analyzing “feeds” and “speeds”

PRAM

- Parallel Random Access Machine (PRAM)
- Shared-memory multiprocessor model
- Unlimited number of processors
 - Unlimited local memory
 - Each processor knows its ID
- Unlimited shared memory
- Inputs/outputs are placed in shared memory
- Memory cells can store an arbitrarily large integer
- Each instruction takes unit time
- Instructions are synchronized across processors (SIMD)

PRAM Complexity Measures

- For each individual processor
 - *Time*: number of instructions executed
 - *Space*: number of memory cells accessed
- PRAM machine
 - *Time*: time taken by the longest running processor
 - *Hardware*: maximum number of active processors
- Technical issues
 - How processors are activated
 - How shared memory is accessed

Processor Activation

- P_0 places the number of processors (p) in the designated shared-memory cell
 - Each active P_i , where $i < p$, starts executing
 - $O(1)$ time to activate
 - All processors halt when P_0 halts
- Active processors explicitly activate additional processors via FORK instructions
 - Tree-like activation
 - $O(\log p)$ time to activate

PRAM is a Theoretical (Unfeasible) Model

- Interconnection network between processors and memory would require a very large amount of area
- The message-routing on the interconnection network would require time proportional to network size
- Algorithm's designers can forget the communication problems and focus their attention on the parallel computation only
- There exist algorithms simulating any PRAM algorithm on bounded degree networks
- Design general algorithms for the PRAM model and simulate them on a feasible network

Classification of PRAM Models

- *EREW* (Exclusive Read Exclusive Write)
 - No concurrent read/writes to the same memory location
- *CREW* (Concurrent Read Exclusive Write)
 - Multiple processors may read from the same global memory location in the same instruction step
- *ERCW* (Exclusive Read Concurrent Write)
 - Concurrent writes allowed
- *CRCW* (Concurrent Read Concurrent Write)
 - Concurrent reads and writes allowed
- $\text{CRCW} > (\text{ERCW}, \text{CREW}) > \text{EREW}$

CRCW PRAM Models

- COMMON: all processors concurrently writing into the same address must be writing the same value
- ARBITRARY: if multiple processors concurrently write to the address, one of the competing processors is randomly chosen and its value is written into the register
- PRIORITY: if multiple processors concurrently write to the address, the processor with the highest priority succeeds in writing its value to the memory location
- COMBINING: the value stored is some combination of the values written, e.g., sum, min, or max
- COMMON-CRCW model most often used

Complexity of PRAM Algorithms

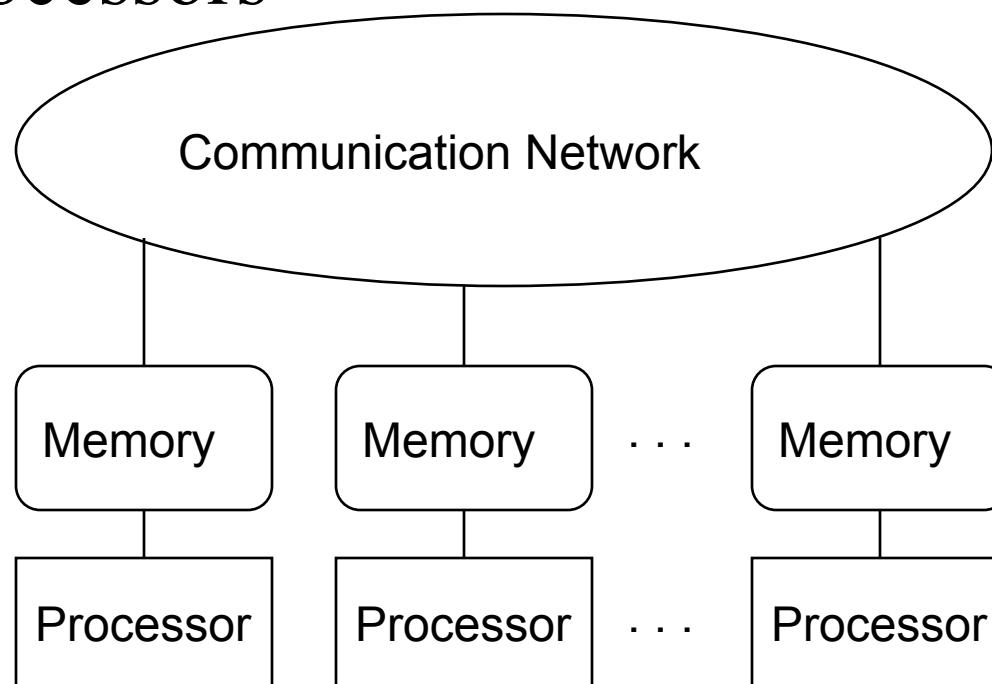
Problem Model	EREW	CRCW
Search	$O(\log n)$	$O(1)$
List Ranking	$O(\log n)$	$O(\log n)$
Prefix	$O(\log n)$	$O(\log n)$
Tree Ranking	$O(\log n)$	$O(\log n)$
Finding Minimum	$O(\log n)$	$O(1)$

BSP Overview

- Bulk Synchronous Parallelism
- A parallel programming model
- Invented by Leslie Valiant at Harvard
- Enables performance prediction
- SPMD (Single Program Multiple Data) style
- Supports both direct memory access and message passing semantics
- BSPlib is a BSP library implemented at Oxford

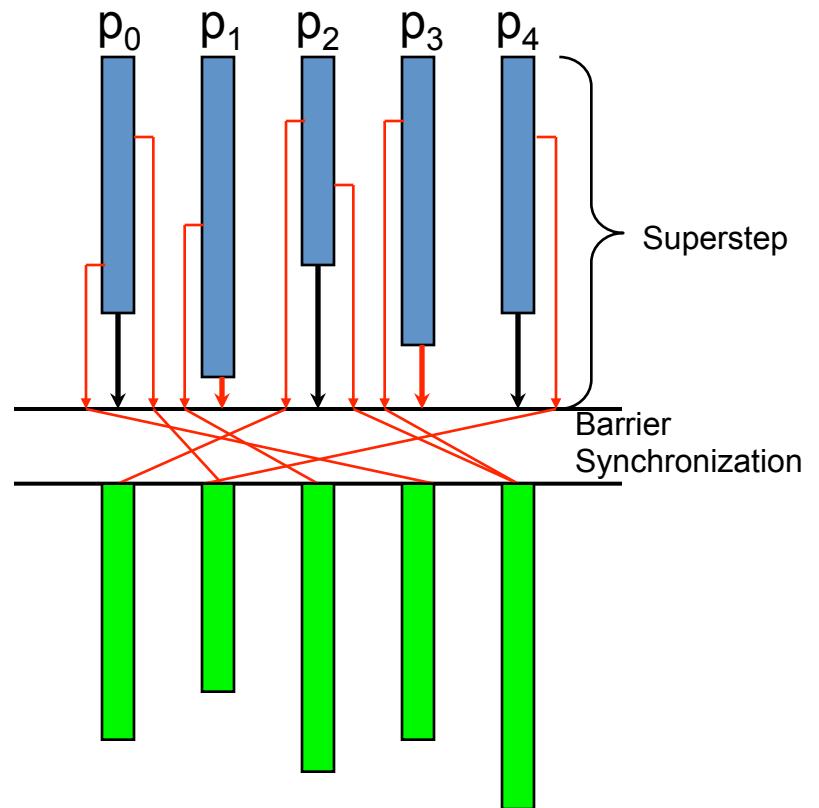
Components of BSP Computer

- A set of processor-memory pairs
- A communication point-to-point network
- A mechanism for efficient barrier synchronization of all processors



BSP Supersteps

- A BSP computation consists of a sequence of *supersteps*
- In each superstep, processes execute computations using locally available data, and issue communication requests
- Processes synchronized at the end of the superstep, at which all communications issued have been completed



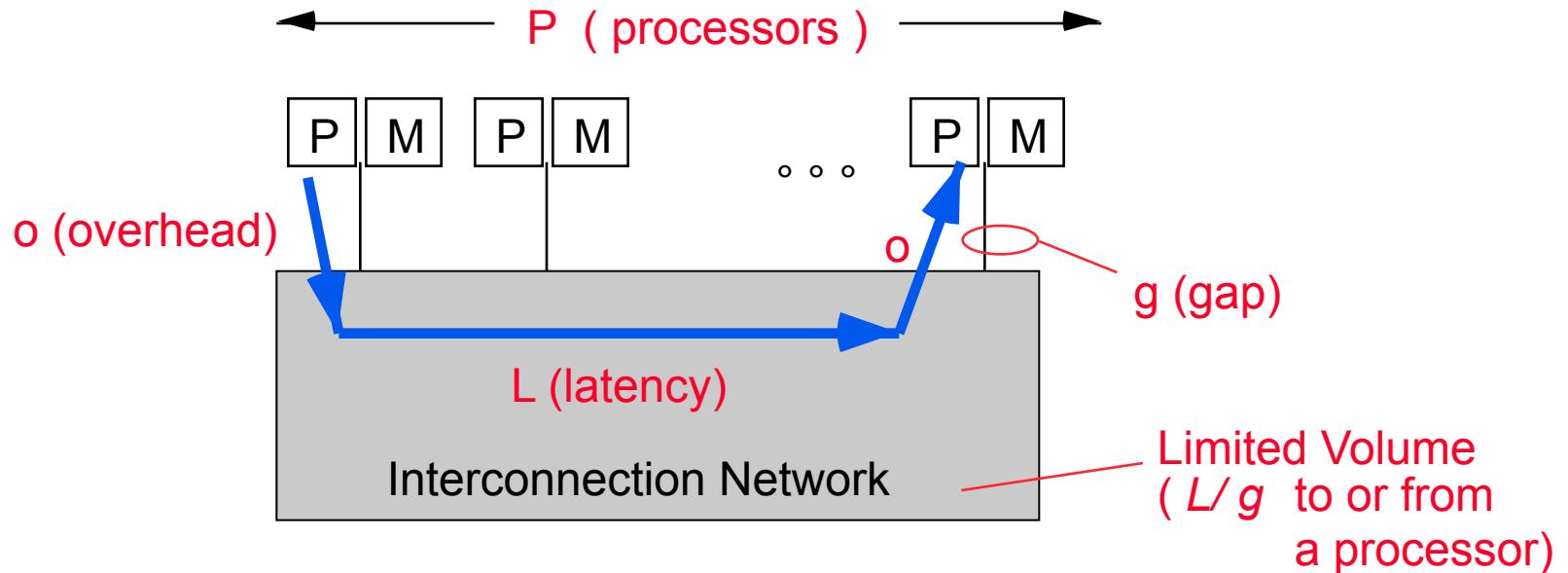
BSP Performance Model Parameters

- p = number of processors
- l = barrier latency, cost of achieving barrier synchronization
- g = communication cost per word
- s = processor speed
- l, g , and s are measured in FLOPS
- Any processor sends and receives at most h messages in a single superstep (called h -relation communication)
- Time for a superstep = max number of local operations performed by any one processor + $g * h + l$

The LogP Model (Culler, Berkeley)

- Processing
 - Powerful microprocessor, large DRAM, cache => P
- Communication
 - Significant latency (100's of cycles) => L
 - Limited bandwidth (1 – 5% of memory) => g
 - Significant overhead (10's – 100's of cycles)
 - ◆ on both ends
 - ◆ no consensus on topology
 - ◆ should not exploit structure
 - Limited capacity
- No consensus on programming model
 - Should not enforce one

LogP



- Latency in sending a (small) message between modules
- Overhead felt by the processor on sending or receiving message
- gap between successive sends or receives ($1/BW$)
- Processors

LogP "Philosophy"

- Think about:
 - Mapping of N words onto P processors
 - Computation within a processor
 - ◆ its cost and balance
 - Communication between processors
 - ◆ its cost and balance
- Characterize processor and network performance
- Do not think about what happens in the network
- This should be enough

Typical Values for g and l

	p	g	l
Multiprocessor Sun	2-4	3	50-100
SGI Origin 2000	2-8	10-15	1000-4000
IBM-SP2	2-8	10	2000-5000
NOW (Network of Workstations)	2-8	40	5000-20000

Parallel Programming

- To use a scalable parallel computer, you must be able to write parallel programs
- You must understand the programming model and the programming languages, libraries, and systems software used to implement it
- Unfortunately, parallel programming is not easy

Parallel Programming: Are we having fun yet?



Source: Bernd Mohr

Parallel Programming Models

- Two general models of parallel program
 - Task parallel
 - ◆ problem is broken down into tasks to be performed
 - ◆ individual tasks are created and communicate to coordinate operations
 - Data parallel
 - ◆ problem is viewed as operations of parallel data
 - ◆ data distributed across processes and computed locally
- Characteristics of scalable parallel programs
 - Data domain decomposition to improve data locality
 - Communication and latency do not grow significantly

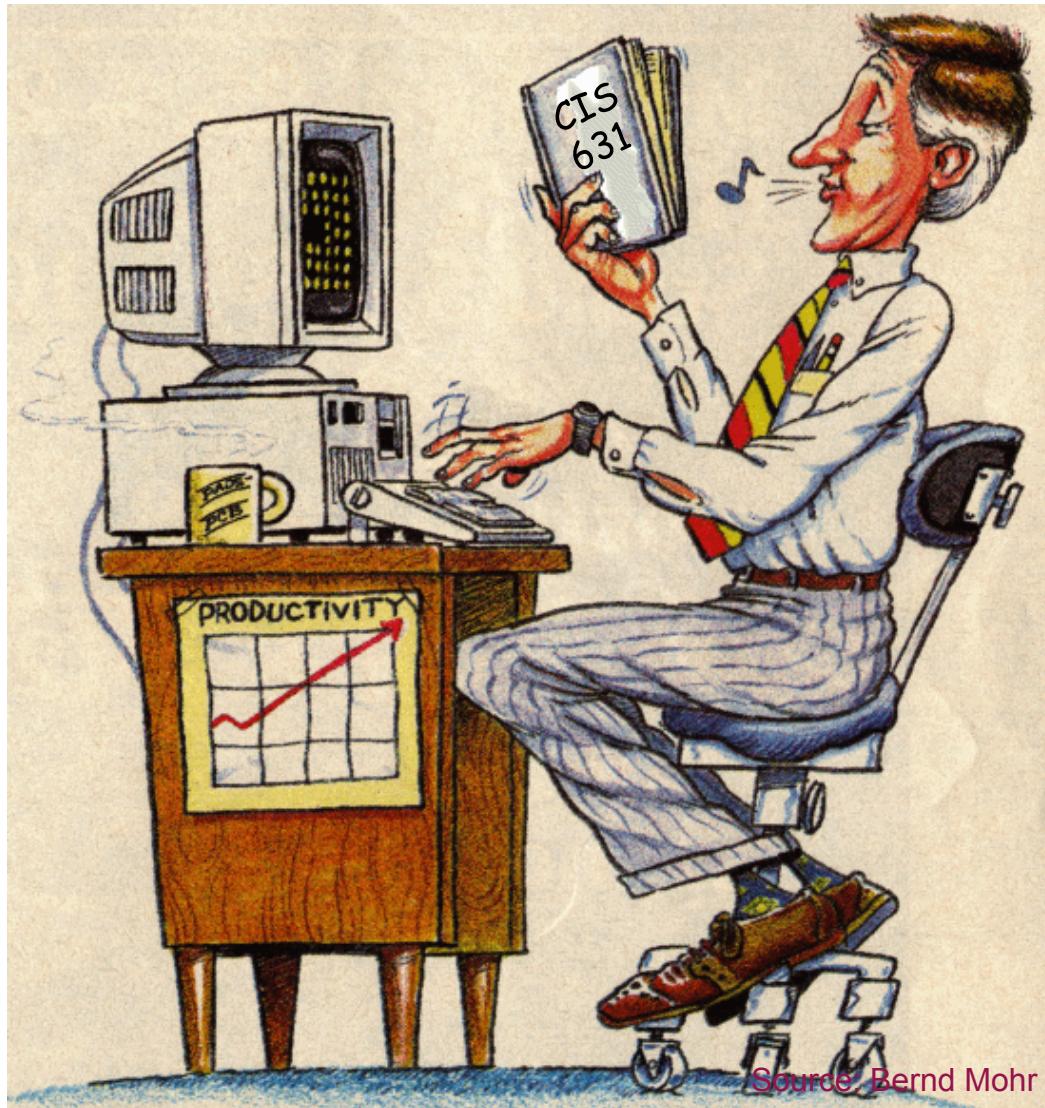
Shared Memory Parallel Programming

- Shared memory address space
- (Typically) easier to program
 - Implicit communication via (shared) data
 - Explicit synchronization to access data
- Programming methodology
 - Manual
 - ◆ multi-threading using standard thread libraries
 - Automatic
 - ◆ parallelizing compilers
 - ◆ OpenMP parallelism directives
 - Explicit threading (e.g. POSIX threads)

Distributed Memory Parallel Programming

- Distributed memory address space
- (Relatively) harder to program
 - Explicit data distribution
 - Explicit communication via messages
 - Explicit synchronization via messages
- Programming methodology
 - Message passing
 - ◆ plenty of libraries to chose from (MPI dominates)
 - ◆ send-receive, one-sided, active messages
 - Data parallelism

Parallel Programming: Still a Problem?



Parallel Computing and Scalability

- Scalability in parallel architecture
 - Processor numbers
 - Memory architecture
 - Interconnection network
 - Avoid critical architecture bottlenecks
- Scalability in computational problem
 - Problem size
 - Computational algorithms
 - ◆ computation to memory access ratio
 - ◆ computation to communication ratio
- Parallel programming models and tools
- Performance scalability

Parallel Performance and Complexity

- To use a scalable parallel computer well, you must write high-performance parallel programs
- To get high-performance parallel programs, you must understand and optimize performance for the combination of programming model, algorithm, language, platform, ...
- Unfortunately, parallel performance measurement, analysis and optimization can be an easy process
- Parallel performance is complex



Parallel Performance Evaluation

- Study of performance in parallel systems
 - Models and behaviors
 - Evaluative techniques
- Evaluation methodologies
 - Analytical modeling and statistical modeling
 - Simulation-based modeling
 - Empirical measurement, analysis, and modeling
- Purposes
 - Planning
 - Diagnosis
 - Tuning

Parallel Performance Engineering and Productivity

- Scalable, optimized applications deliver HPC promise
- Optimization through *performance engineering* process
 - Understand performance complexity and inefficiencies
 - Tune application to run optimally on high-end machines
- How to make the process more effective and productive?
- What performance technology should be used?
 - Performance technology part of larger environment
 - Programmability, reusability, portability, robustness
 - Application development and optimization productivity
- Process, performance technology, and its use will change as parallel systems evolve
- Goal is to deliver effective performance with high productivity value now and in the future

Motivation

- Parallel / distributed systems are complex
 - Four layers
 - ◆ application
 - algorithm, data structures
 - ◆ parallel programming interface / middleware
 - compiler, parallel libraries, communication, synchronization
 - ◆ operating system
 - process and memory management, IO
 - ◆ hardware
 - CPU, memory, network
- Mapping/interaction between different layers

Performance Factors

- Factors which determine a program's performance are complex, interrelated, and sometimes hidden
- Application related factors
 - Algorithms dataset sizes, task granularity, memory usage patterns, load balancing. I/O communication patterns
- Hardware related factors
 - Processor architecture, memory hierarchy, I/O network
- Software related factors
 - Operating system, compiler/preprocessor, communication protocols, libraries

Utilization of Computational Resources

- Resources can be under-utilized or used inefficiently
 - Identifying these circumstances can give clues to where performance problems exist
- Resources may be “virtual”
 - Not actually a physical resource (e.g., thread, process)
- Performance analysis tools are essential to optimizing an application's performance
 - Can assist you in understanding what your program is “really doing”
 - May provide suggestions how program performance should be improved

Performance Analysis and Tuning: The Basics

- Most important goal of performance tuning is to reduce a program's wall clock execution time
 - Iterative process to optimize efficiency
 - Efficiency is a relationship of execution time
- So, where does the time go?
- Find your program's hot spots and eliminate the bottlenecks in them
 - **Hot spot**: an area of code within the program that uses a disproportionately high amount of processor time
 - **Bottleneck** : an area of code within the program that uses processor resources inefficiently and therefore causes unnecessary delays
- Understand *what, where, and how* time is being spent

Sequential Performance

- Sequential performance is all about:
 - How time is distributed
 - What resources are used where and when
- “Sequential” factors
 - Computation
 - ◆ choosing the right algorithm is important
 - ◆ compilers can help
 - Memory systems and cache and memory
 - ◆ more difficult to assess and determine effects
 - ◆ modeling can help
 - Input / output

Parallel Performance

- Parallel performance is about sequential performance AND parallel interactions
 - Sequential performance is the performance within each thread of execution
 - “Parallel” factors lead to overheads
 - ◆ concurrency (threading, processes)
 - ◆ interprocess communication (message passing)
 - ◆ synchronization (both explicit and implicit)
 - Parallel interactions also lead to parallelism inefficiency
 - ◆ load imbalances

Sequential Performance Tuning

- Sequential performance tuning is a *time-driven* process
- Find the thing that takes the most time and make it take less time (i.e., make it more efficient)
- May lead to program restructuring
 - Changes in data storage and structure
 - Rearrangement of tasks and operations
- May look for opportunities for better resource utilization
 - Cache management is a big one
 - Locality, locality, locality!
 - Virtual memory management may also pay off
- May look for opportunities for better processor usage

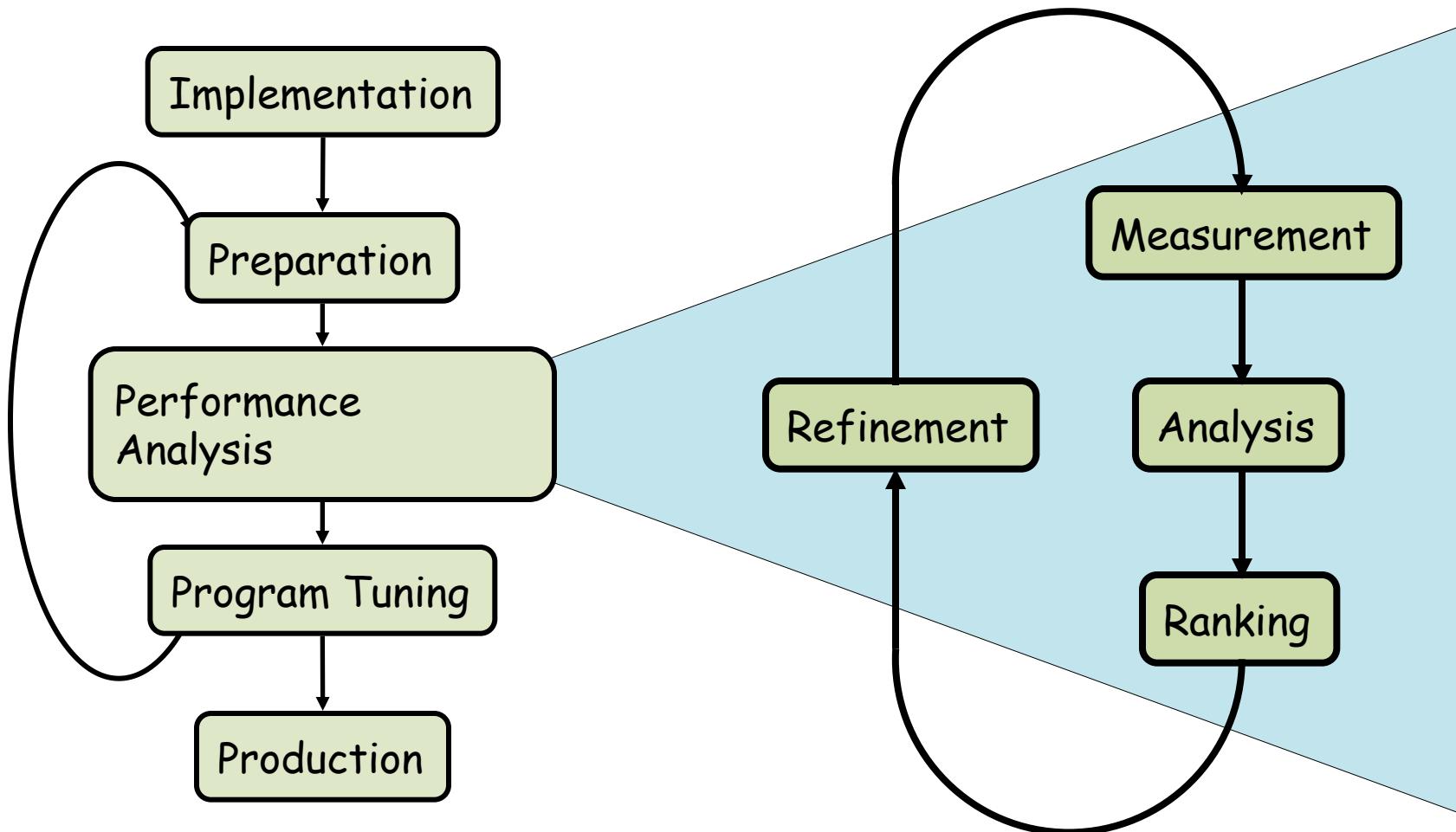
Parallel Performance Tuning

- In contrast to sequential performance tuning, parallel performance tuning might be described as *conflict-driven* or *interaction-driven*
- Find the points of parallel interactions and determine the overheads associated with them
- Overheads can be the cost of performing the interactions
 - Transfer of data
 - Extra operations to implement coordination
- Overheads also include time spent waiting
 - Lack of work
 - Waiting for dependency to be satisfied

Interesting Performance Phenomena

- Superlinear speedup
 - Speedup in parallel execution is greater than linear
 - $S_p > p$
 - How can this happen?
- Need to keep in mind the relationship of performance and resource usage
- Computation time (i.e., real work) is not simply a linear distribution to parallel threads of execution
- Resource utilization thresholds can lead to performance inflections

Parallel Performance Engineering Process





Parallel Programming Patterns Overview and Map Pattern

Parallel Computing

CIS 410/510

Department of Computer and Information Science



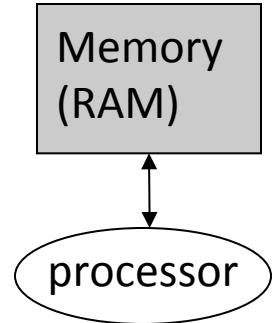
UNIVERSITY OF OREGON

Outline

- Parallel programming models
- Dependencies
- Structured programming patterns overview
 - Serial / parallel control flow patterns
 - Serial / parallel data management patterns
- Map pattern
 - Optimizations
 - ◆ sequences of Maps
 - ◆ code Fusion
 - ◆ cache Fusion
 - Related Patterns
 - Example: Scaled Vector Addition (SAXPY)

Parallel Models 101

- Sequential models
 - von Neumann (RAM) model
- Parallel model
 - A parallel computer is simple a collection of *processors interconnected* in some manner to *coordinate* activities and *exchange data*
 - Models that can be used as general frameworks for describing and analyzing parallel algorithms
 - ◆ *Simplicity*: description, analysis, architecture independence
 - ◆ *Implementability*: able to be realized, reflect performance
- Three common parallel models
 - Directed acyclic graphs, shared-memory, network



Directed Acyclic Graphs (DAG)

- Captures data flow parallelism
- Nodes represent operations to be performed
 - Inputs are nodes with no incoming arcs
 - Output are nodes with no outgoing arcs
 - Think of nodes as tasks
- Arcs are paths for flow of data results
- DAG represents the operations of the algorithm and implies precedent constraints on their order

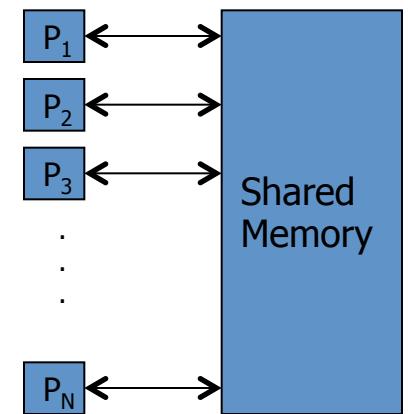
for ($i=1; i<100; i++$)

$$a[i] = a[i-1] + 100;$$



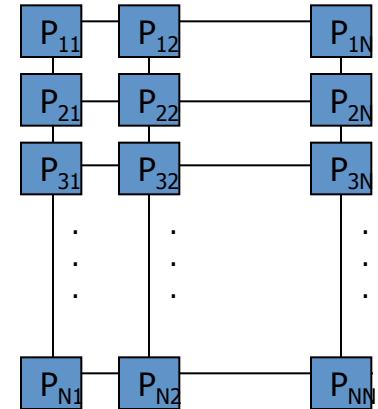
Shared Memory Model

- Parallel extension of RAM model (PRAM)
 - Memory size is infinite
 - Number of processors in unbounded
 - Processors communicate via the memory
 - Every processor accesses any memory location in 1 cycle
 - Synchronous
 - ◆ All processors execute same algorithm synchronously
 - READ phase
 - COMPUTE phase
 - WRITE phase
 - ◆ Some subset of the processors can stay idle
 - Asynchronous



Network Model

- $G = (N, E)$
 - N are processing nodes
 - E are bidirectional communication links
- Each processor has its own memory
- No shared memory is available
- Network operation may be synchronous or asynchronous
- Requires communication primitives
 - Send (X, i)
 - Receive (Y, j)
- Captures message passing model for algorithm design



Parallelism

- Ability to execute different parts of a computation concurrently on different machines
- Why do you want parallelism?
 - Shorter running time or handling more work
- What is being parallelized?
 - *Task*: instruction, statement, procedure, ...
 - *Data*: data flow, size, replication
 - Parallelism granularity
 - ◆ Coarse-grain versus fine-grained
- Thinking about parallelism
- Evaluation

Why is parallel programming important?

- Parallel programming has matured
 - Standard programming models
 - Common machine architectures
 - Programmer can focus on computation and use suitable programming model for implementation
- Increase portability between models and architectures
- Reasonable hope of portability across platforms
- Problem
 - Performance optimization is still platform-dependent
 - Performance portability is a problem
 - Parallel programming methods are still evolving

Parallel Algorithm

- Recipe to solve a problem “in parallel” on multiple processing elements
- Standard steps for constructing a parallel algorithm
 - Identify work that can be performed concurrently
 - Partition the concurrent work on separate processors
 - Properly manage input, output, and intermediate data
 - Coordinate data accesses and work to satisfy dependencies
- Which are hard to do?

Parallelism Views

- Where can we find parallelism?
- Program (task) view
 - Statement level
 - ◆ Between program statements
 - ◆ Which statements can be executed at the same time?
 - Block level / Loop level / Routine level / Process level
 - ◆ Larger-grained program statements
- Data view
 - How is data operated on?
 - Where does data reside?
- Resource view

Parallelism, Correctness, and Dependence

- Parallel execution, from any point of view, will be constrained by the sequence of operations needed to be performed for a correct result
- Parallel execution must address control, data, and system dependences
- A *dependency* arises when one operation depends on an earlier operation to complete and produce a result before this later operation can be performed
- We extend this notion of dependency to resources since some operations may depend on certain resources
 - For example, due to where data is located

Executing Two Statements in Parallel

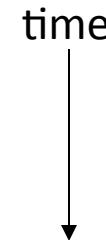
- Want to execute two statements in parallel
- On one processor:
 - Statement 1;
 - Statement 2;
- On two processors:

Processor 1:	Processor 2:
Statement 1;	Statement 2;
- Fundamental (*concurrent*) execution assumption
 - Processors execute independent of each other
 - No assumptions made about speed of processor execution

Sequential Consistency in Parallel Execution

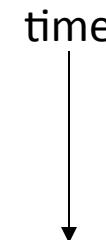
- Case 1:

Processor 1: Processor 2:
statement 1;
 statement 2;



- Case 2:

Processor 1: Processor 2:
 statement 2;
statement 1;



- Sequential consistency

- Statements execution does not interfere with each other
- Computation results are the same (independent of order)

Independent versus Dependent

- In other words the execution of
 - statement1;
 - statement2;must be equivalent to
 - statement2;
 - statement1;
- Their order of execution must not matter!
- If true, the statements are *independent* of each other
- Two statements are *dependent* when the order of their execution affects the computation outcome

Examples

- Example 1
 - S1: $a=1;$
 - S2: $b=1;$
- Example 2
 - S1: $a=1;$
 - S2: $b=a;$
- Example 3
 - S1: $a=f(x);$
 - S2: $a=b;$
- Example 4
 - S1: $a=b;$
 - S2: $b=1;$
- Statements are independent
- Dependent (*true (flow) dependence*)
 - Second is dependent on first
 - Can you remove dependency?
- Dependent (*output dependence*)
 - Second is dependent on first
 - Can you remove dependency? How?
- Dependent (*anti-dependence*)
 - First is dependent on second
 - Can you remove dependency? How?

True Dependence and Anti-Dependence

- Given statements S1 and S2,

S1;

S2;

- S2 has a *true (flow) dependence* on S1

if and only if

S2 reads a value written by S1

$$\begin{array}{c} x = \\ \vdots \\ = x \end{array} \quad \square \quad \delta$$

- S2 has a *anti-dependence* on S1

if and only if

S2 writes a value read by S1

$$\begin{array}{c} = x \\ \vdots \\ x = \end{array} \quad \square \quad \delta^{-1}$$

Output Dependence

- Given statements S1 and S2,
S1;
S2;
- S2 has an *output dependence* on S1
if and only if
S2 writes a variable written by S1

$$x = \begin{array}{c} : \\ \lceil \end{array} \delta^0$$

- Anti- and output dependences are “name” dependencies
 - Are they “true” dependences?
- How can you get rid of output dependences?
 - Are there cases where you can not?

Statement Dependency Graphs

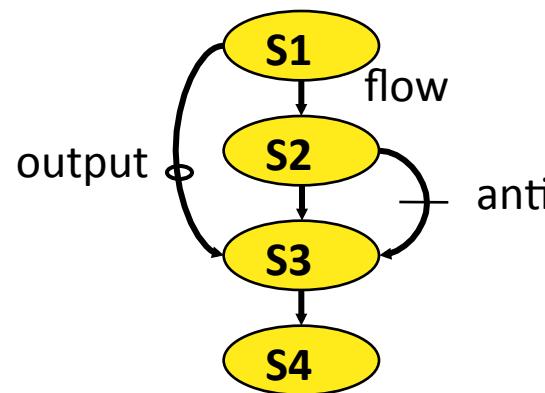
- Can use graphs to show dependence relationships
- Example

S1: $a=1;$

S2: $b=a;$

S3: $a=b+1;$

S4: $c=a;$



- $S_2 \delta S_3 : S_3$ is flow-dependent on S_2
- $S_1 \delta^0 S_3 : S_3$ is output-dependent on S_1
- $S_2 \delta^{-1} S_3 : S_3$ is anti-dependent on S_2

When can two statements execute in parallel?

- Statements S1 and S2 can execute in parallel if and only if there are *no dependences* between S1 and S2
 - True dependences
 - Anti-dependences
 - Output dependences
- Some dependences can be removed by modifying the program
 - Rearranging statements
 - Eliminating statements

How do you compute dependence?

- Data dependence relations can be found by comparing the IN and OUT sets of each node
- The IN and OUT sets of a statement **S** are defined as:
 - **IN(S)** : set of memory locations (variables) that may be used in **S**
 - **OUT(S)** : set of memory locations (variables) that may be modified by **S**
- Note that these sets include all memory locations that may be fetched or modified
- As such, the sets can be conservatively large

IN / OUT Sets and Computing Dependence

- Assuming that there is a path from S_1 to S_2 , the following shows how to intersect the IN and OUT sets to test for data dependence

$out(S_1) \cap in(S_2) \neq \emptyset \quad S_1 \delta S_2$ flow dependence

$in(S_1) \cap out(S_2) \neq \emptyset \quad S_1 \delta^{-1} S_2$ anti - dependence

$out(S_1) \cap out(S_2) \neq \emptyset \quad S_1 \delta^0 S_2$ output dependence

Loop-Level Parallelism

- Significant parallelism can be identified within loops

```
for (i=0; i<100; i++)  
    S1: a[i] = i;
```

```
for (i=0; i<100; i++) {  
    S1: a[i] = i;  
    S2: b[i] = 2*i;  
}
```

- Dependencies? What about i , the loop index?
- *DOALL* loop (a.k.a. *foreach* loop)
 - All iterations are independent of each other
 - All statements be executed in parallel at the same time
 - ◆ Is this really true?

Iteration Space

- Unroll loop into separate statements / iterations
- Show dependences between iterations

for ($i=0; i<100; i++$)

S1: $a[i] = i;$

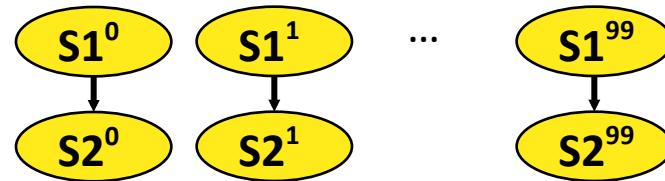


for ($i=0; i<100; i++$) {

S1: $a[i] = i;$

S2: $b[i] = 2*i;$

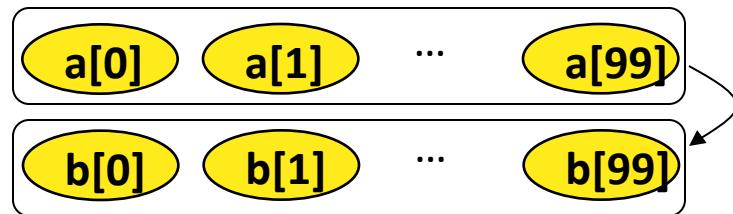
}



Multi-Loop Parallelism

- Significant parallelism can be identified between loops

for ($i=0; i<100; i++$) $a[i] = i;$



for ($i=0; i<100; i++$) $b[i] = i;$

- Dependencies?
- How much parallelism is available?
- Given 4 processors, how much parallelism is possible?
- What parallelism is achievable with 50 processors?

Loops with Dependencies

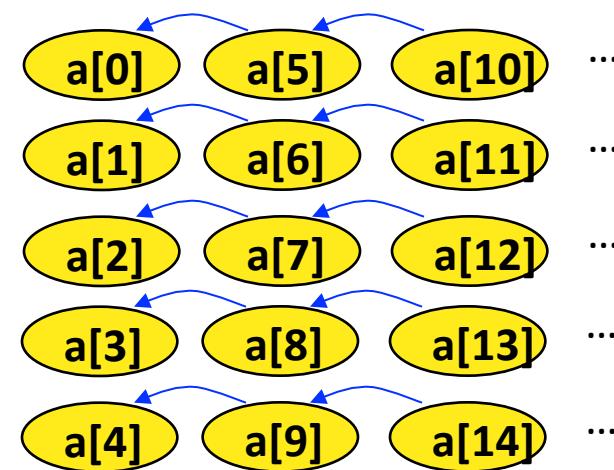
Case 1:

```
for (i=1; i<100; i++)  
    a[i] = a[i-1] + 100;
```



Case 2:

```
for (i=5; i<100; i++)  
    a[i-5] = a[i] + 100;
```



- Dependencies?
 - What type?
- Is the Case 1 loop parallelizable?
- Is the Case 2 loop parallelizable?

Another Loop Example

```
for (i=1; i<100; i++)
```

```
    a[i] = f(a[i-1]);
```

- Dependencies?
 - What type?
- Loop iterations are not parallelizable
 - Why not?

Loop Dependencies

- A *loop-carried* dependence is a dependence that is present only if the statements are part of the execution of a loop (i.e., between two statements instances in two different iterations of a loop)
- Otherwise, it is *loop-independent*, including between two statements instances in the same loop iteration
- Loop-carried dependences can prevent loop iteration parallelization
- The dependence is *lexically forward* if the source comes before the target or *lexically backward* otherwise
 - Unroll the loop to see

Loop Dependence Example

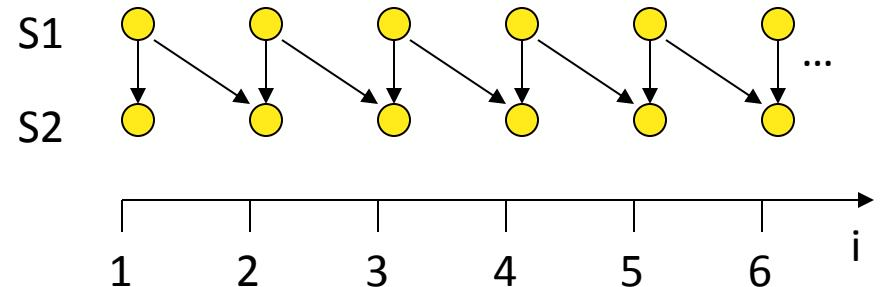
for ($i=0; i<100; i++$)

$a[i+10] = f(a[i]);$

- Dependencies?
 - Between $a[10], a[20], \dots$
 - Between $a[11], a[21], \dots$
- Some parallel execution is possible
 - How much?

Dependences Between Iterations

```
for (i=1; i<100; i++) {  
    S1: a[i] = ...;  
    S2: ... = a[i-1];  
}
```



- Dependencies?
 - Between $a[i]$ and $a[i-1]$
- Is parallelism possible?
 - Statements can be executed in “pipeline” manner

Another Loop Dependence Example

```
for (i=0; i<100; i++)  
    for (j=1; j<100; j++)  
        a[i][j] = f(a[i][j-1]);
```

- Dependencies?
 - Loop-independent dependence on i
 - Loop-carried dependence on j
- Which loop can be parallelized?
 - Outer loop parallelizable
 - Inner loop cannot be parallelized

Still Another Loop Dependence Example

```
for (j=1; j<100; j++)  
    for (i=0; i<100; i++)  
        a[i][j] = f(a[i][j-1]);
```

- Dependencies?
 - Loop-independent dependence on i
 - Loop-carried dependence on j
- Which loop can be parallelized?
 - Inner loop parallelizable
 - Outer loop cannot be parallelized
 - Less desirable (why?)

Key Ideas for Dependency Analysis

- To execute in parallel:
 - Statement order must not matter
 - Statements must not have dependences
- Some dependences can be removed
- Some dependences may not be obvious

Dependencies and Synchronization

- How is parallelism achieved when have dependencies?
 - Think about concurrency
 - Some parts of the execution are independent
 - Some parts of the execution are dependent
- Must control ordering of events on different processors (cores)
 - Dependencies pose constraints on parallel event ordering
 - Partial ordering of execution action
- Use synchronization mechanisms
 - Need for concurrent execution too
 - Maintains partial order

Parallel Patterns

- **Parallel Patterns:** A recurring combination of task distribution and data access that solves a specific problem in parallel algorithm design.
- Patterns provide us with a “vocabulary” for algorithm design
- It can be useful to compare parallel patterns with serial patterns
- Patterns are universal – they can be used in *any* parallel programming system

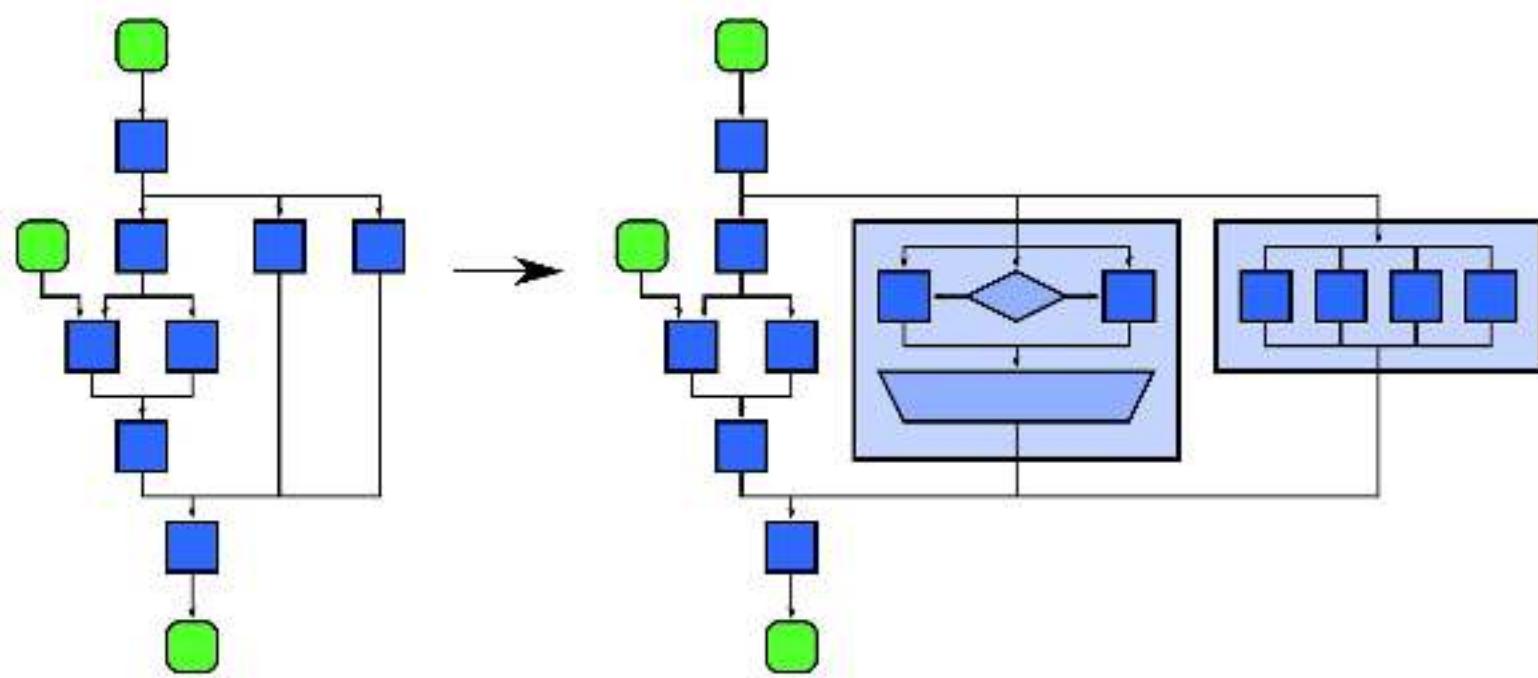
Parallel Patterns

- Nesting Pattern
- Serial / Parallel Control Patterns
- Serial / Parallel Data Management Patterns
- Other Patterns
- Programming Model Support for Patterns

Nesting Pattern

- **Nesting** is the ability to hierarchically compose patterns
- This pattern appears in both serial and parallel algorithms
- “Pattern diagrams” are used to visually show the pattern idea where each “task block” is a location of general code in an algorithm
- Each “task block” can in turn be another pattern in the **nesting pattern**

Nesting Pattern



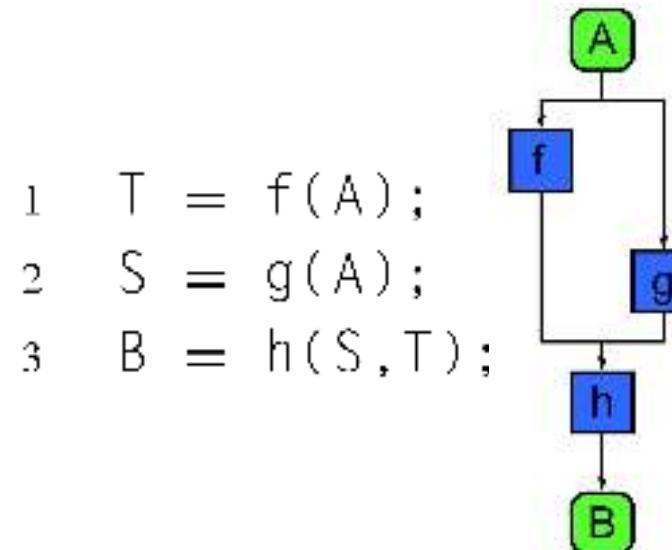
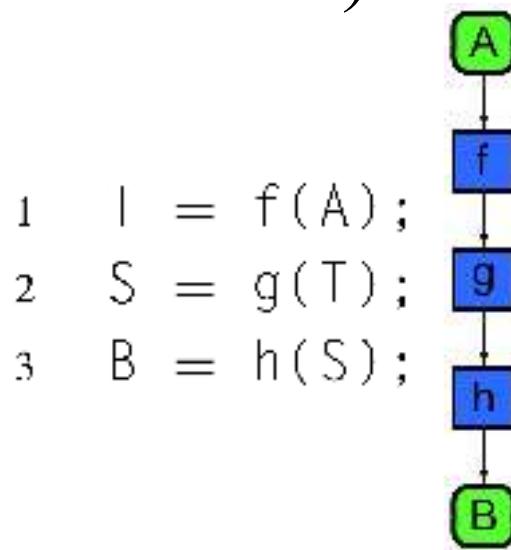
Nesting Pattern: A compositional pattern. Nesting allows other patterns to be composed in a hierarchy so that any task block in the above diagram can be replaced with a pattern with the same input/output and dependencies.

Serial Control Patterns

- Structured serial programming is based on these patterns: **sequence**, **selection**, **iteration**, and **recursion**
- The **nesting** pattern can also be used to hierarchically compose these four patterns
- Though you should be familiar with these, it's extra important to understand these patterns when parallelizing serial algorithms based on these patterns

Serial Control Patterns: Sequence

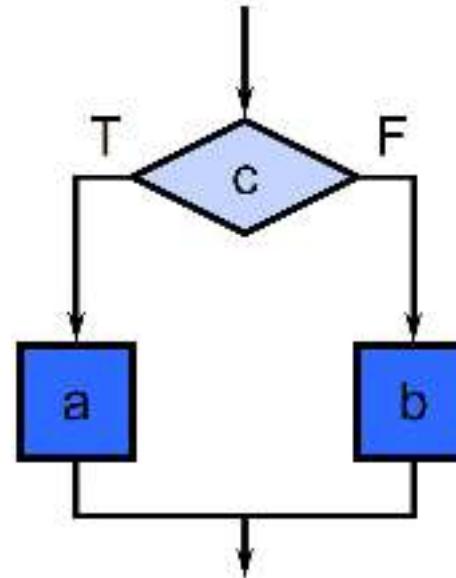
- **Sequence:** Ordered list of tasks that are executed in a specific order
- Assumption – program text ordering will be followed (obvious, but this will be important when parallelized)



Serial Control Patterns: Selection

- **Selection:** condition c is first evaluated. Either task a or b is executed depending on the true or false result of c .
- Assumptions – a and b are never executed before c , and only a or b is executed - never both

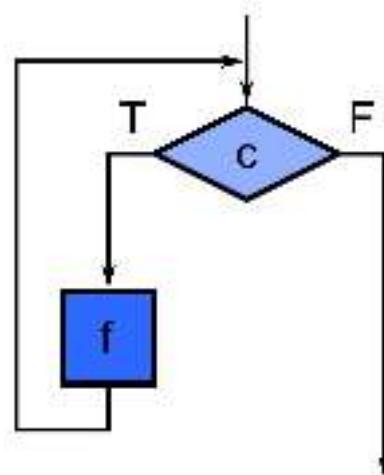
```
1  if (c) {  
2      a;  
3  } else {  
4      b;  
5  }
```



Serial Control Patterns: Iteration

- **Iteration:** a condition c is evaluated. If true, a is evaluated, and then c is evaluated again. This repeats until c is false.
- Complication when parallelizing: potential for dependencies to exist between previous iterations

```
1  for (i = 0; i < n;  
2      a;  
3  }  
-----  
1  while (c) {  
2      a;  
3  }
```



Serial Control Patterns: Recursion

- **Recursion:** dynamic form of nesting allowing functions to call themselves
- Tail recursion is a special recursion that can be converted into iteration – important for functional languages

Parallel Control Patterns

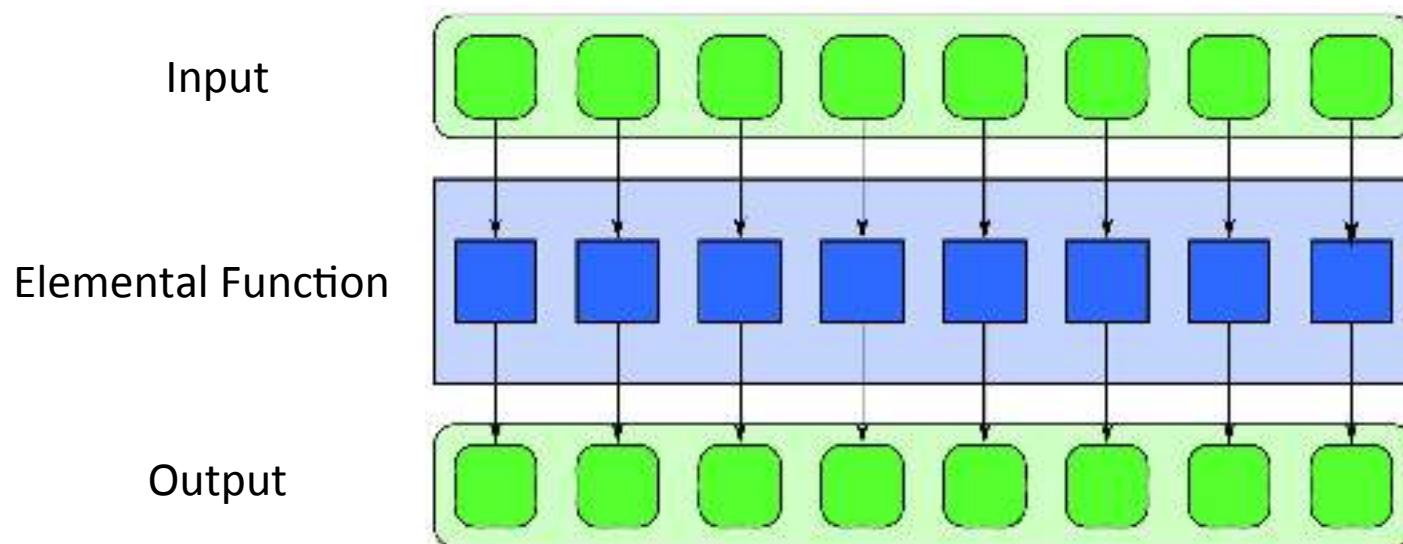
- Parallel control patterns extend serial control patterns
- Each parallel control pattern is related to at least one serial control pattern, but relaxes assumptions of serial control patterns
- Parallel control patterns: **fork-join, map, stencil, reduction, scan, recurrence**

Parallel Control Patterns: Fork-Join

- **Fork-join:** allows control flow to fork into multiple parallel flows, then rejoin later
- Cilk Plus implements this with **spawn** and **sync**
 - The call tree is a parallel call tree and functions are spawned instead of called
 - Functions that spawn another function call will continue to execute
 - Caller *syncs* with the spawned function to join the two
- A “join” is different than a “barrier
 - Sync – only one thread continues
 - Barrier – all threads continue

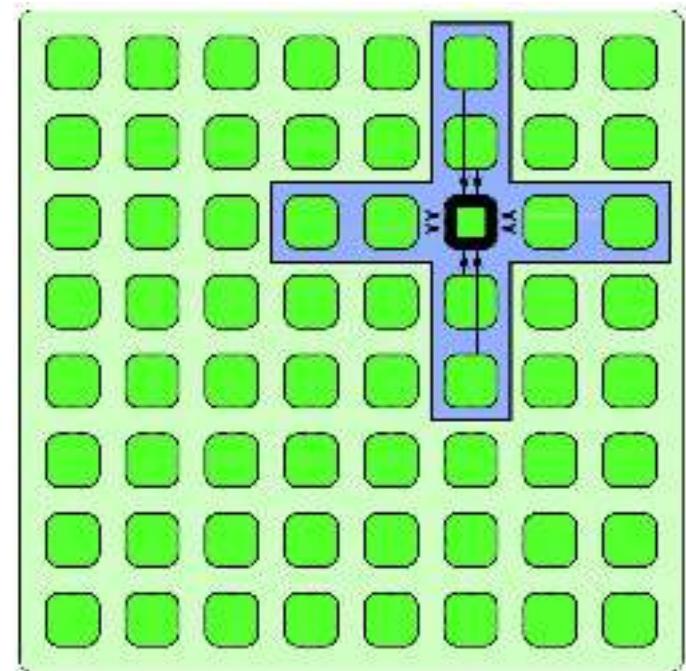
Parallel Control Patterns: Map

- **Map**: performs a function over every element of a collection
- Map replicates a serial iteration pattern where each iteration is independent of the others, the number of iterations is known in advance, and computation only depends on the iteration count and data from the input collection
- The replicated function is referred to as an “elemental function”



Parallel Control Patterns: Stencil

- **Stencil:** Elemental function accesses a set of “neighbors”, stencil is a generalization of map
- Often combined with iteration – used with iterative solvers or to evolve a system through time
- Boundary conditions must be handled carefully in the stencil pattern
- See stencil lecture...

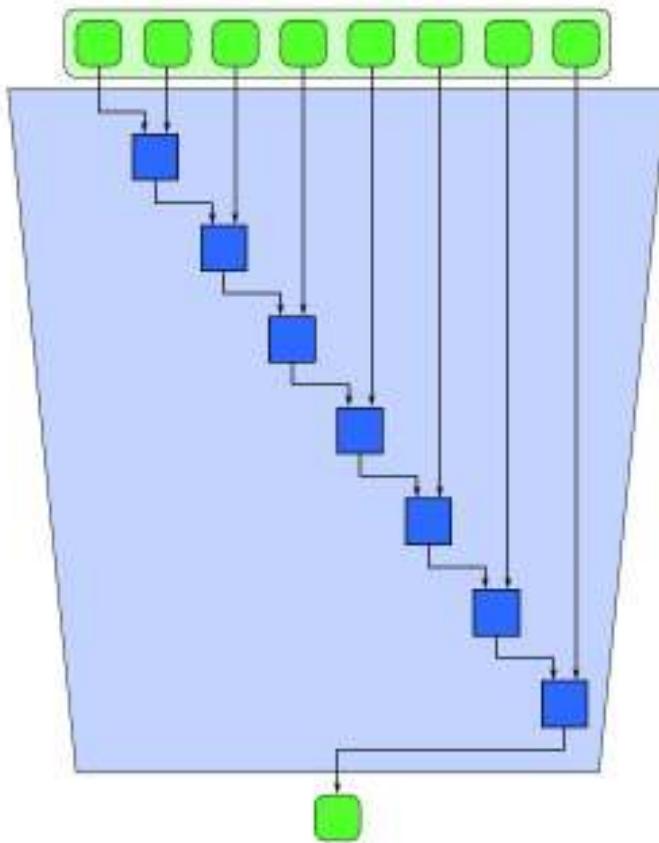


Parallel Control Patterns: Reduction

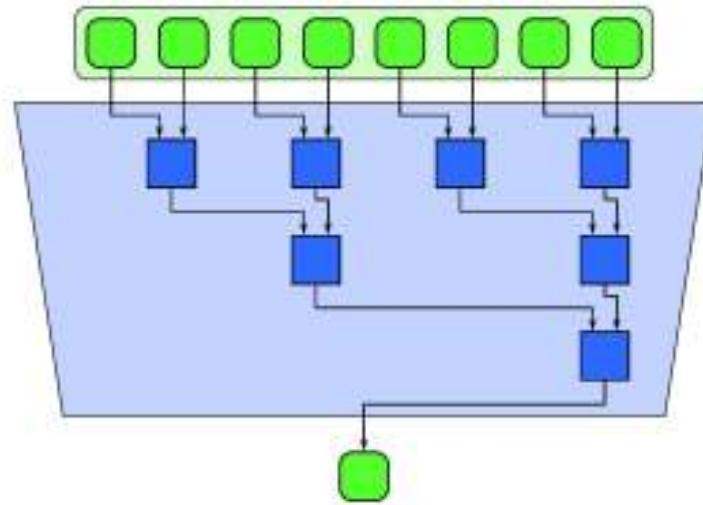
- **Reduction:** Combines every element in a collection using an associative “combiner function”
- Because of the associativity of the combiner function, different orderings of the reduction are possible
- Examples of combiner functions: addition, multiplication, maximum, minimum, and Boolean AND, OR, and XOR

Parallel Control Patterns: Reduction

Serial Reduction



Parallel Reduction

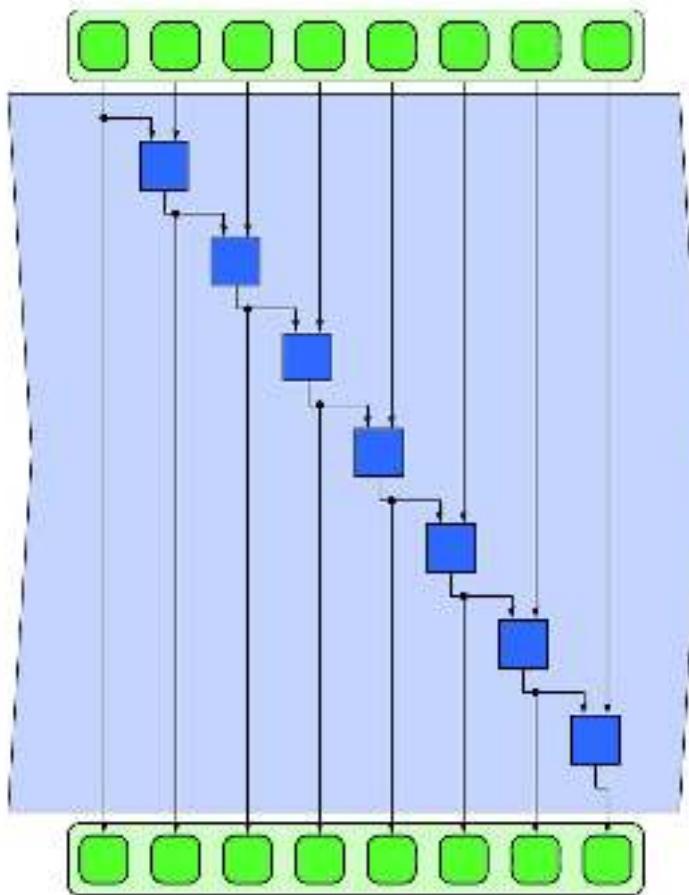


Parallel Control Patterns: Scan

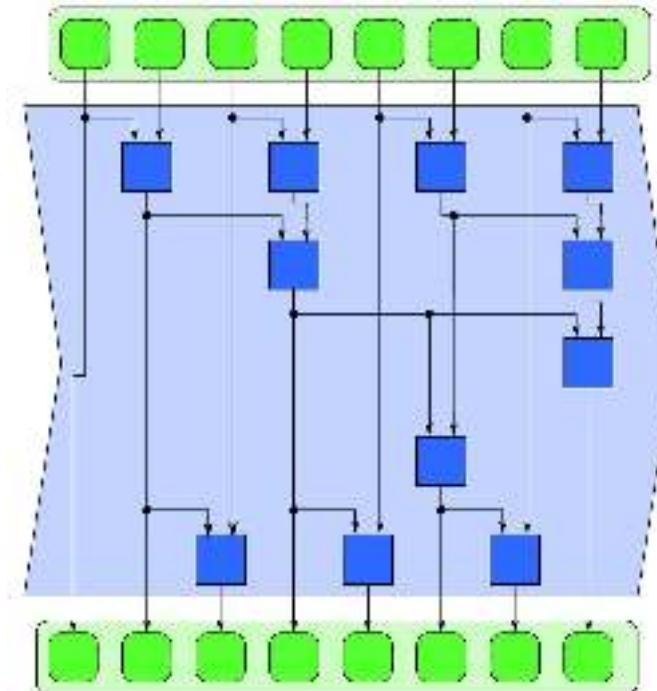
- **Scan**: computes all partial reduction of a collection
- For every output in a collection, a reduction of the input up to that point is computed
- If the function being used is associative, the scan can be parallelized
- Parallelizing a scan is not obvious at first, because of dependencies to previous iterations in the serial loop
- A parallel scan will require more operations than a serial version

Parallel Control Patterns: Scan

Serial Scan



Parallel Scan



Parallel Control Patterns: Recurrence

- **Recurrence:** More complex version of map, where the loop iterations can depend on one another
- Similar to map, but elements can use outputs of adjacent elements as inputs
- For a recurrence to be computable, there *must* be a serial ordering of the recurrence elements so that elements can be computed using previously computed outputs

Serial Data Management Patterns

- Serial programs can manage data in many ways
- Data management deals with how data is allocated, shared, read, written, and copied
- Serial Data Management Patterns: **random read and write, stack allocation, heap allocation, objects**

Serial Data Management Patterns: random read and write

- Memory locations indexed with addresses
- Pointers are typically used to refer to memory addresses
- Aliasing (uncertainty of two pointers referring to the same object) can cause problems when serial code is parallelized

Serial Data Management Patterns: Stack Allocation

- Stack allocation is useful for dynamically allocating data in LIFO manner
- Efficient – arbitrary amount of data can be allocated in constant time
- Stack allocation also preserves locality
- When parallelized, typically each thread will get its own stack so thread locality is preserved

Serial Data Management Patterns: Heap Allocation

- Heap allocation is useful when data cannot be allocated in a LIFO fashion
- But, heap allocation is slower and more complex than stack allocation
- A parallelized heap allocator should be used when dynamically allocating memory in parallel
 - This type of allocator will keep separate pools for each parallel worker

Serial Data Management Patterns: Objects

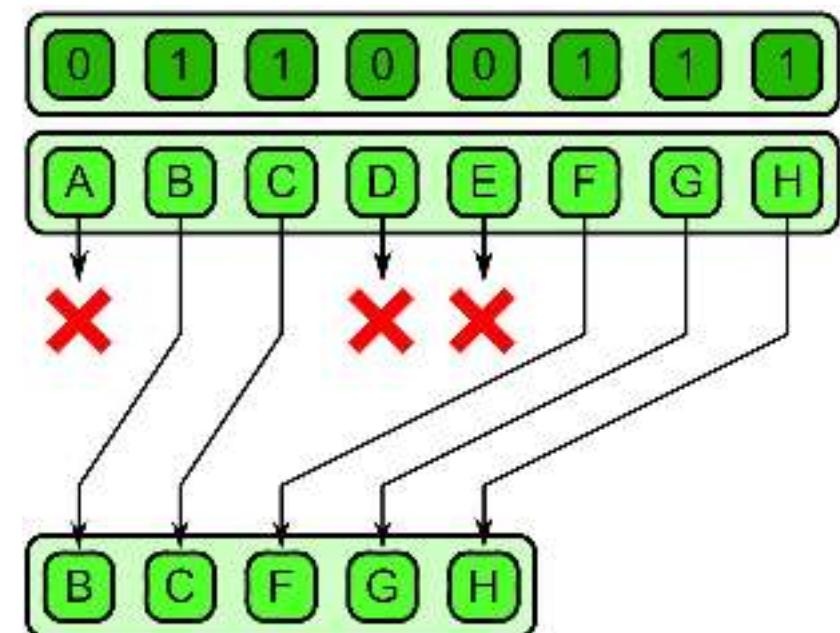
- Objects are language constructs to associate data with code to manipulate and manage that data
- Objects can have member functions, and they also are considered members of a class of objects
- Parallel programming models will generalize objects in various ways

Parallel Data Management Patterns

- To avoid things like race conditions, it is critically important to know when data is, and isn't, potentially shared by multiple parallel workers
- Some parallel data management patterns help us with data locality
- Parallel data management patterns: **pack**, **pipeline**, **geometric decomposition**, **gather**, and **scatter**

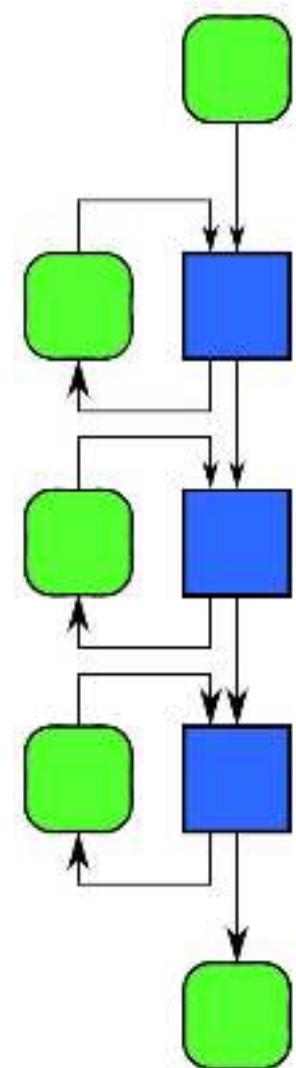
Parallel Data Management Patterns: Pack

- **Pack** is used to eliminate unused space in a collection
- Elements marked *false* are discarded, the remaining elements are placed in a contiguous sequence in the same order
- Useful when used with map
- **Unpack** is the inverse and is used to place elements back in their original locations



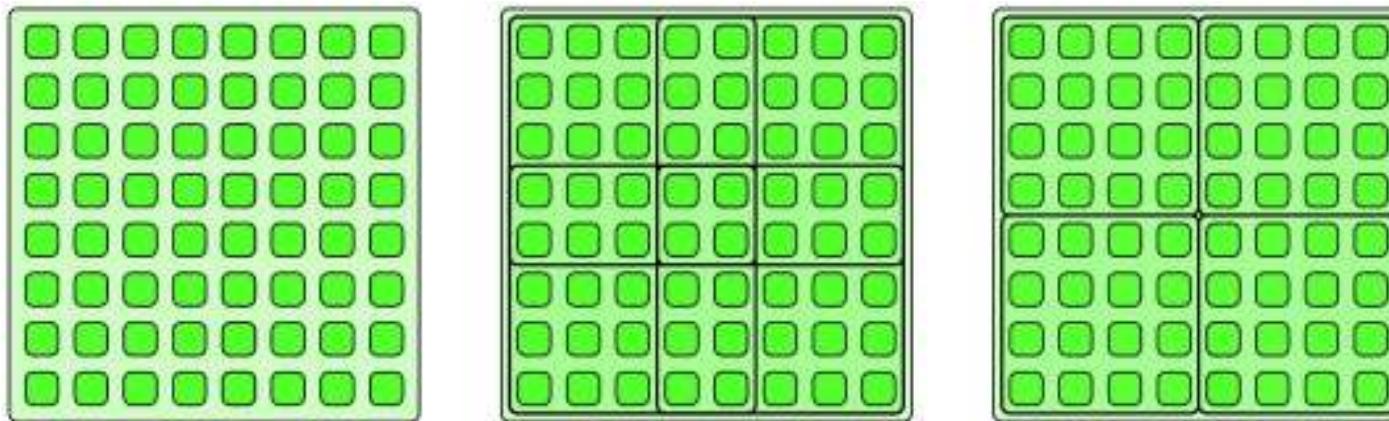
Parallel Data Management Patterns: Pipeline

- **Pipeline** connects tasks in a producer-consumer manner
- A linear pipeline is the basic pattern idea, but a pipeline in a DAG is also possible
- Pipelines are most useful when used with other patterns as they can multiply available parallelism



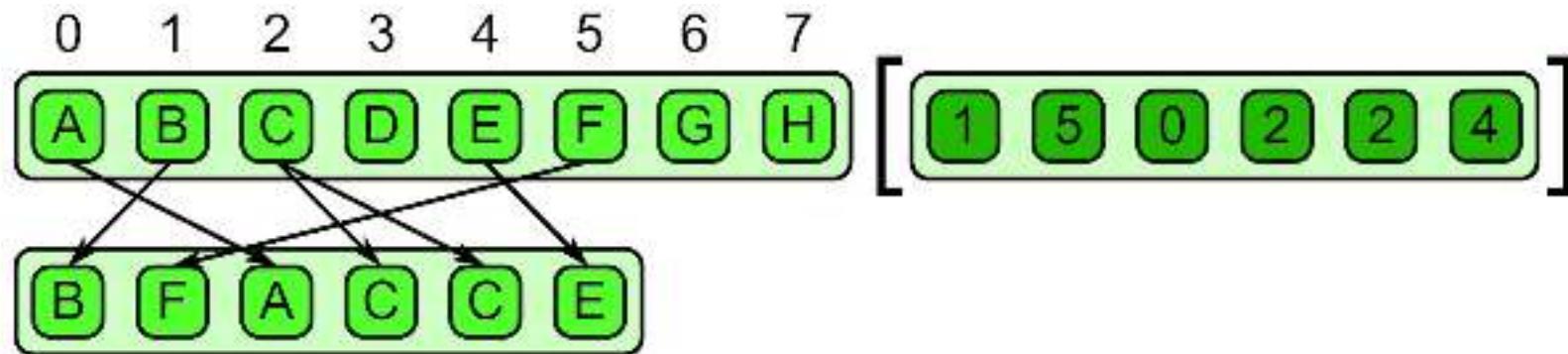
Parallel Data Management Patterns: Geometric Decomposition

- **Geometric Decomposition** – arranges data into subcollections
- Overlapping and non-overlapping decompositions are possible
- This pattern doesn't necessarily move data, it just gives us another view of it



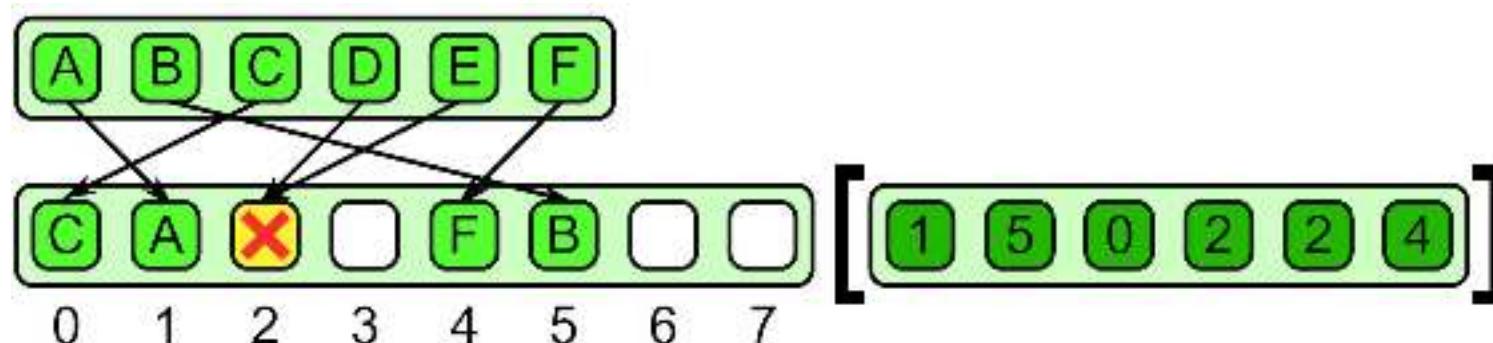
Parallel Data Management Patterns: Gather

- **Gather** reads a collection of data given a collection of indices
- Think of a combination of map and random serial reads
- The output collection shares the same type as the input collection, but it share the same shape as the indices collection



Parallel Data Management Patterns: Scatter

- Scatter is the inverse of gather
- A set of input and indices is required, but each element of the input is written to the output at the given index instead of read from the input at the given index
- Race conditions can occur when we have two writes to the same location!



Other Parallel Patterns

- **Superscalar Sequences**: write a sequence of tasks, ordered only by dependencies
- **Futures**: similar to fork-join, but tasks do not need to be nested hierarchically
- **Speculative Selection**: general version of serial selection where the condition and both outcomes can all run in parallel
- **Workpile**: general map pattern where each instance of elemental function can generate more instances, adding to the “pile” of work

Other Parallel Patterns

- **Search:** finds some data in a collection that meets some criteria
- **Segmentation:** operations on subdivided, non-overlapping, non-uniformly sized partitions of 1D collections
- **Expand:** a combination of pack and map
- **Category Reduction:** Given a collection of elements each with a label, find all elements with same label and reduce them

Programming Model Support for Patterns

Table 3.1 Summary of programming model support for the serial patterns discussed in this book. Note that some of the parallel programming models we consider do not, in fact, support all the common serial programming patterns. In particular, note that recursion and memory allocation are limited on some model.

Serial Pattern	TBB	Cilk Plus	OpenMP	ArBB	OpenCL
(Serial) Nesting	F	F	F	F	F
Sequence	F	F	F	F	F
Selection	F	F	F	F	F
Iteration	F	F	F	F	F
Recursion	F	F	F		?
Random Read	F	F	F	F	F
Random Write	F	F	F		F
Stack Allocation	F	F	F		?
Heap Allocation	F	F	F		
Closures				F	F
Objects	F	F	F(w/C++)	F	

Programming Model Support for Patterns

Table 3.2 Summary of programming model support for the patterns discussed in this book. F: Supported directly, with a special feature. I: Can be implemented easily and efficiently using other features. P: Implementations of one pattern in terms of others, listed under the pattern being implemented. Blank means the particular pattern cannot be implemented in that programming model (or that an efficient implementation cannot be implemented easily). When examples exist in this book of a particular pattern with a particular model, section references are given.

Parallel Pattern	TBB	Cilk Plus	OpenMP	ArBB	OpenCL
Parallel nesting	F	F			
Map	F 4.2.3; 4.3.3 11	F 4.2.4;4.2.5; 4.3.4;4.3.5 11	F 4.2.6; 4.3.6	F 4.2.7;4.2.8; 4.3.7	F 4.2.9; 4.3.8
Stencil	I 10	I 10	I	F 10	I
Workpile	F				
Reduction	F 5.3.4 11	F 5.3.5 11	F 5.3.6	F 5.3.7	I
Scan	F 5.6.5 14	I 5.6.3 P 8.11 14	I 5.6.4 P 5.4.4	F 5.6.6	I
Fork-join	F 8.9.2 13	F 8.7; 8.9.1 13	I		
Recurrence					
Superscalar sequence					
Futures					
Speculative selection		P 8.12			F
Pack	I 14	I 14	I	F	I
Expand	I	I	I	I	I
Pipeline	F 12	I 12	I		
Geometric decomposition	I 15	I 15	I	I	I
Search	I	I	I	I	I
Category reduction	I	I	I	I	I
Gather	I	F	I	F	I
Atomic scatter	F	I	I		
Permutation scatter	F	F	F	F	F
Merge scatter	I	I	I	F	I
Priority scatter					

Programming Model Support for Patterns

Table 3.3 Additional patterns discussed. F: Supported directly, with a special feature. I: Can be implemented easily and efficiently using other features. Blank means the particular pattern cannot be implemented in that programming model (or that an efficient implementation cannot be implemented easily).

Parallel Pattern	TBB	Cilk Plus	OpenMP	ArBB	OpenCL
Superscalar sequence	I	I	I		F
Futures	I	I	I		I
Speculative selection	I				
Workpile	F	I	I		I
Expand	I	I	I	I	I
Search	I	I	I	I	I
Category reduction	I	I	I	I	I
Atomic scatter	F	I	I		I
Permutation scatter	F	F	F	F	F
Merge scatter	I	I	I	F	I
Priority scatter					

Map Pattern - Overview

- Map
- Optimizations
 - Sequences of Maps
 - Code Fusion
 - Cache Fusion
- Related Patterns
- Example Implementation: Scaled Vector Addition (SAXPY)
 - Problem Description
 - Various Implementations

Mapping

- “Do the same thing many times”

```
foreach i in foo:  
    do something
```

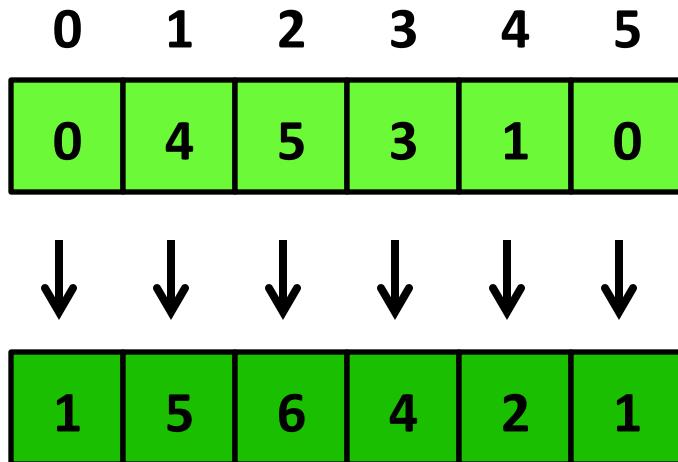
- Well-known higher order function in languages like ML, Haskell, Scala

$$\text{map} : \forall ab. (a \rightarrow b) \text{List}\langle a \rangle \rightarrow \text{List}\langle b \rangle$$

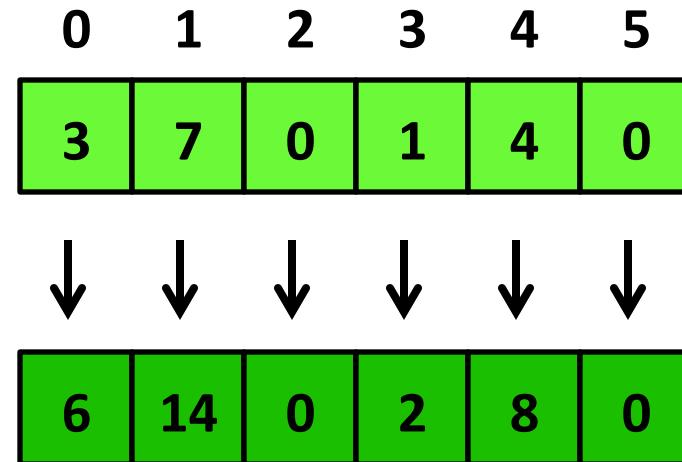
applies a function each element in a list and returns a list of results

Example Maps

Add 1 to every item in an array



Double every item in an array



Key Point: An operation is a map if it can be applied to each element without knowledge of neighbors.

Key Idea

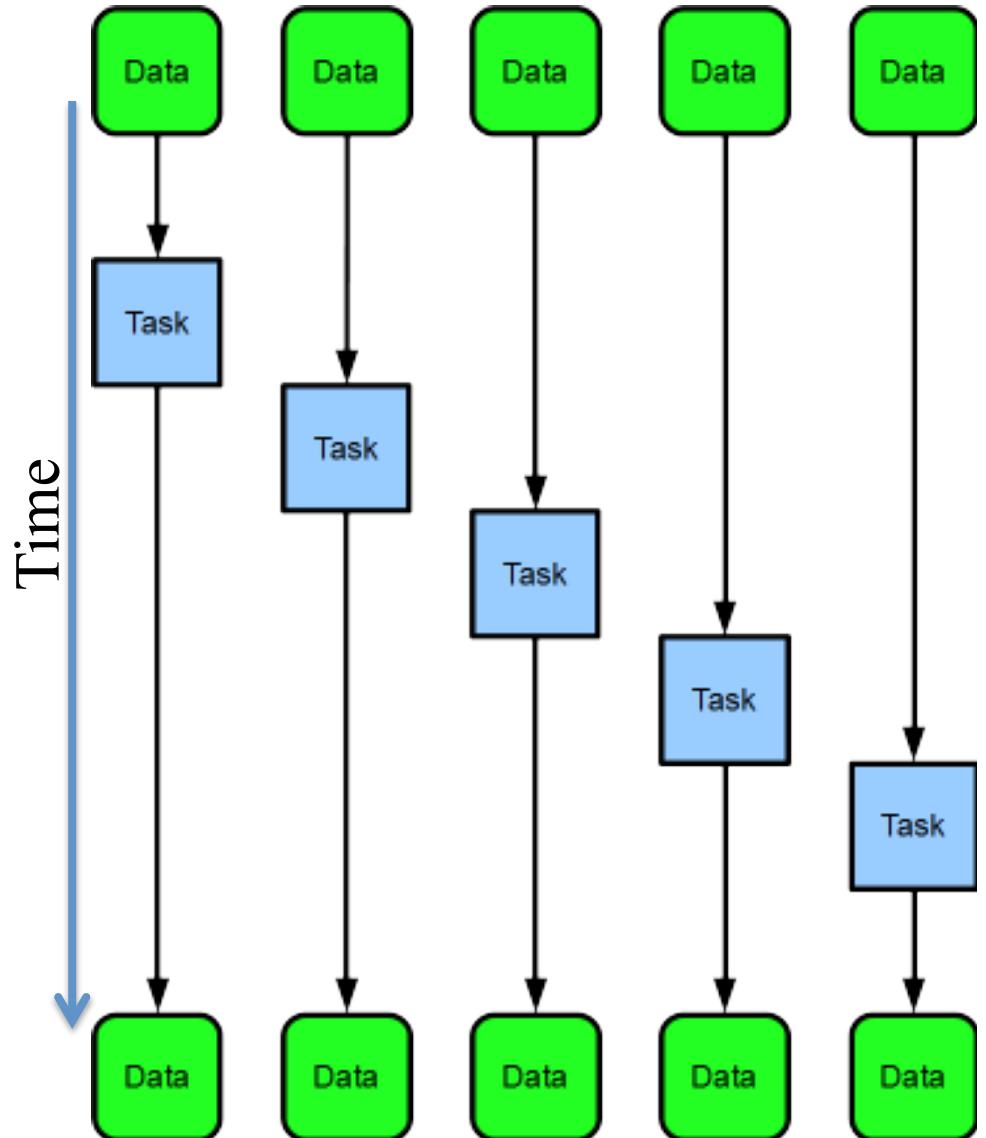
- Map is a “foreach loop” where each iteration is independent

Embarrassingly Parallel

Independence is a big win. We can run map completely in parallel. Significant speedups! More precisely: $T(\infty)$ is $O(1)$ plus implementation overhead that is $O(\log n)$...so $T(\infty) \in O(\log n)$.

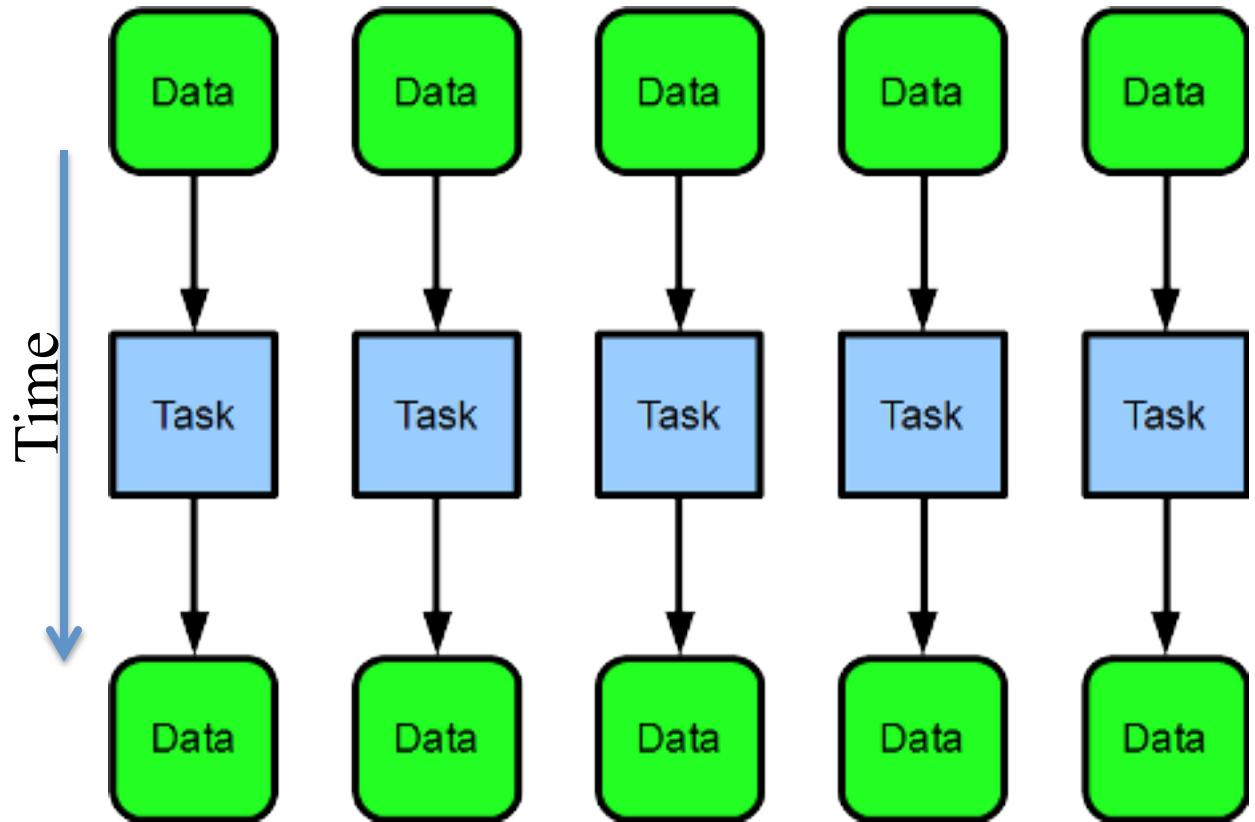
Sequential Map

```
for(int n=0;  
    n< array.length;  
    ++n) {  
    process(array[n]);  
}
```

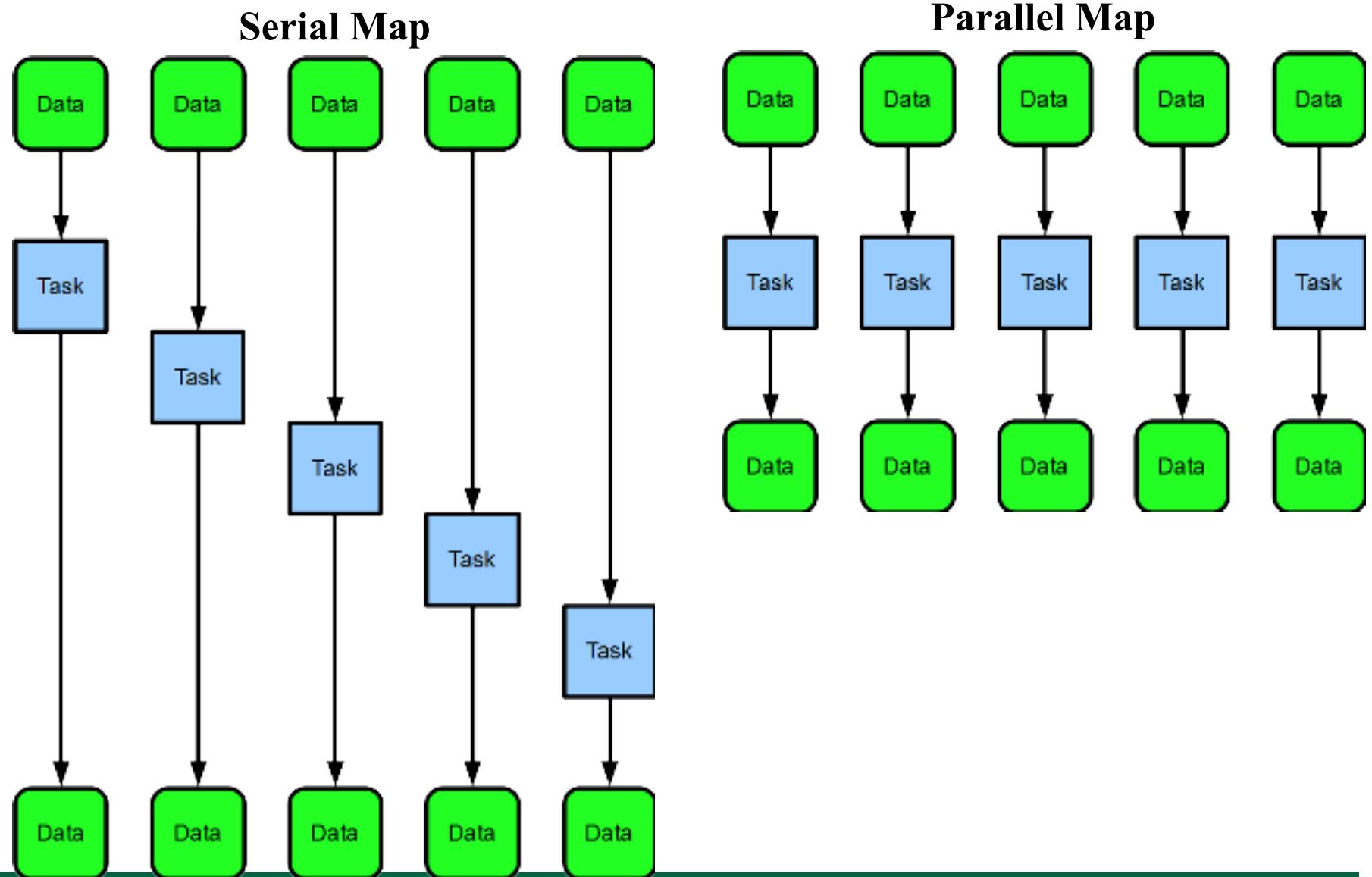


Parallel Map

```
parallel_for_each(  
    x in array) {  
    process(x);  
}
```

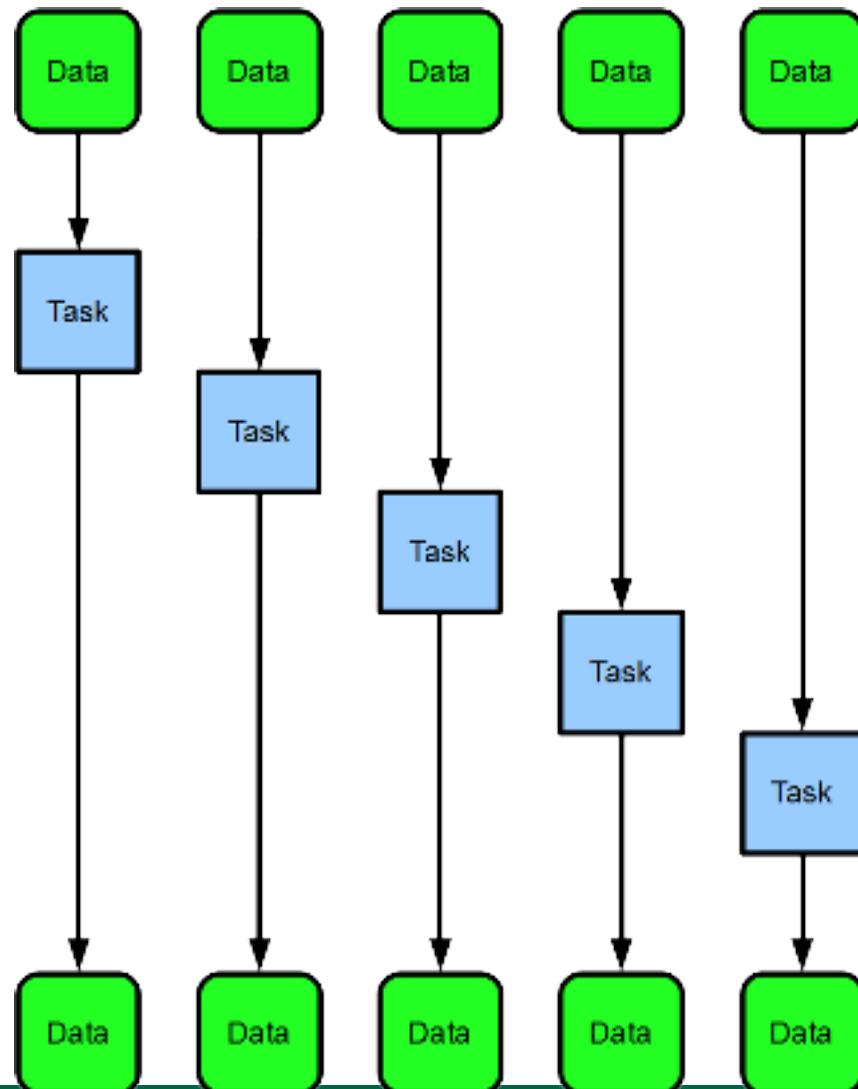


Comparing Maps

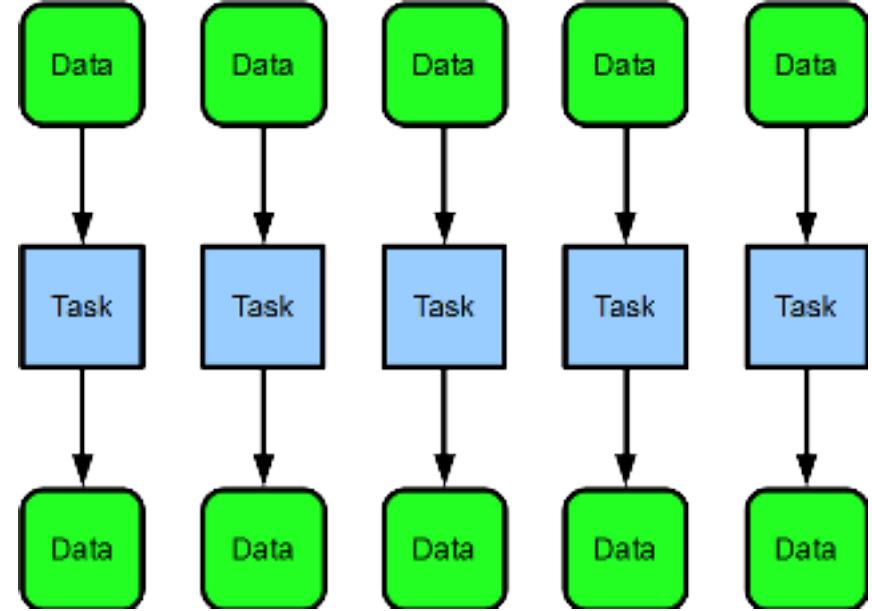


Comparing Maps

Serial Map



Parallel Map



Speedup

The space here is speedup. With the parallel map, our program finished execution early, while the serial map is still running.

Independence

- The key to (embarrassing) parallelism is independence

Warning: No shared state!

Map function should be “pure” (or “pure-ish”) and should not modify shared states

- Modifying shared state breaks perfect independence
- Results of accidentally violating independence:
 - non-determinism
 - data-races
 - undefined behavior
 - segfaults

Implementation and API

- OpenMP and CilkPlus contain a parallel **for** language construct
- Map is a mode of use of parallel **for**
- TBB uses **higher order functions** with lambda expressions/“functors”
- Some languages (CilkPlus, Matlab, Fortran) provide **array notation** which makes some maps more concise

Array Notation

`A[:] = A[:] * 5;`

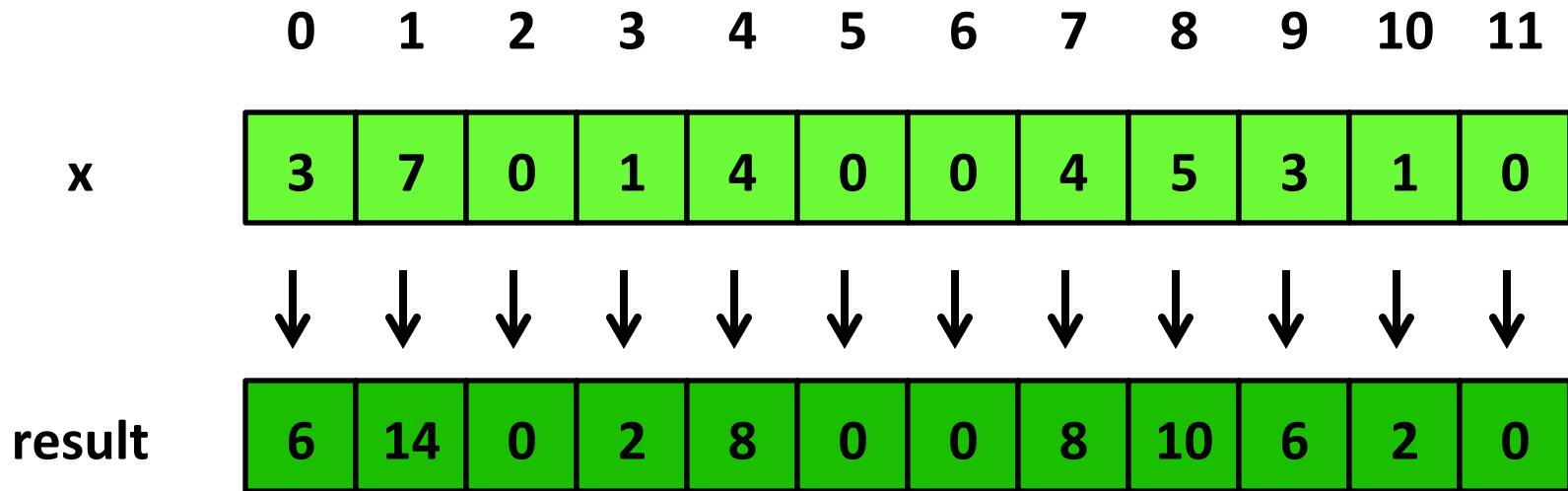
is CilkPlus array notation for “multiply every element in *A* by 5”

Unary Maps

Unary Maps

So far we have only dealt with mapping over a single collection...

Map with 1 Input, 1 Output



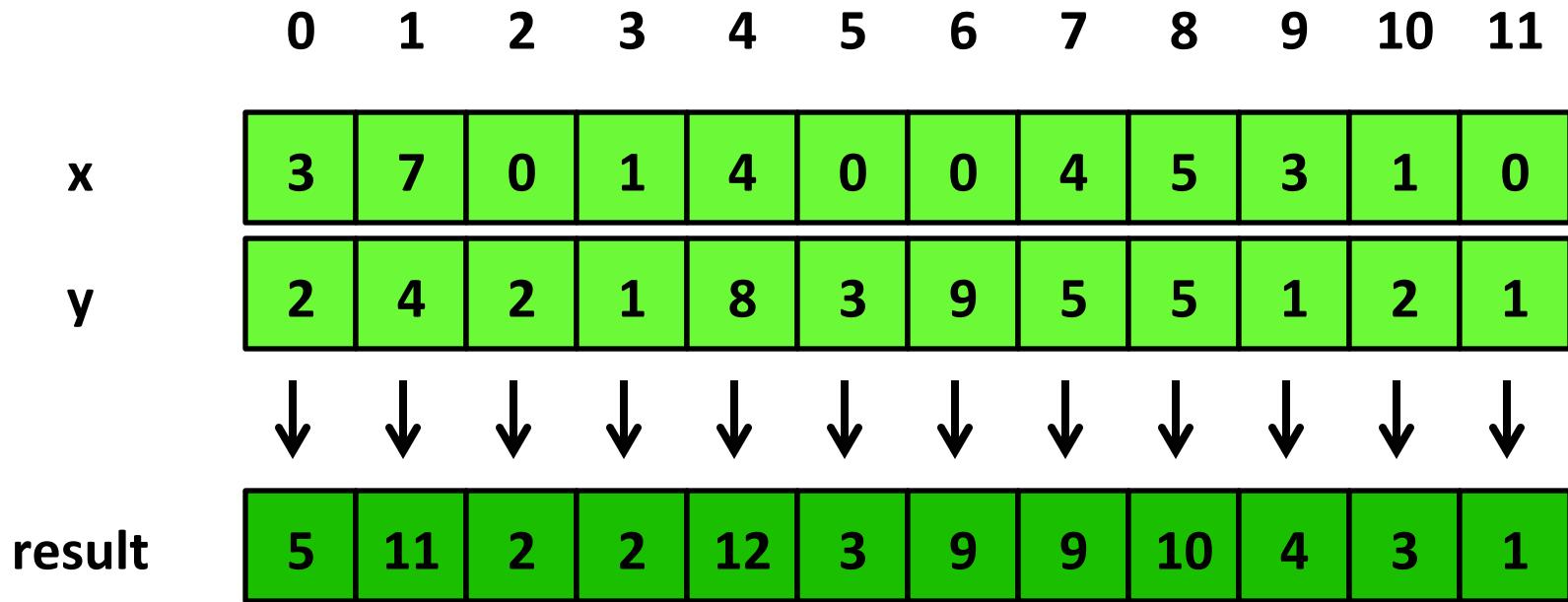
```
int oneToOne ( int x[11] ) {  
    return x*2;  
}
```

N-ary Maps

N-ary Maps

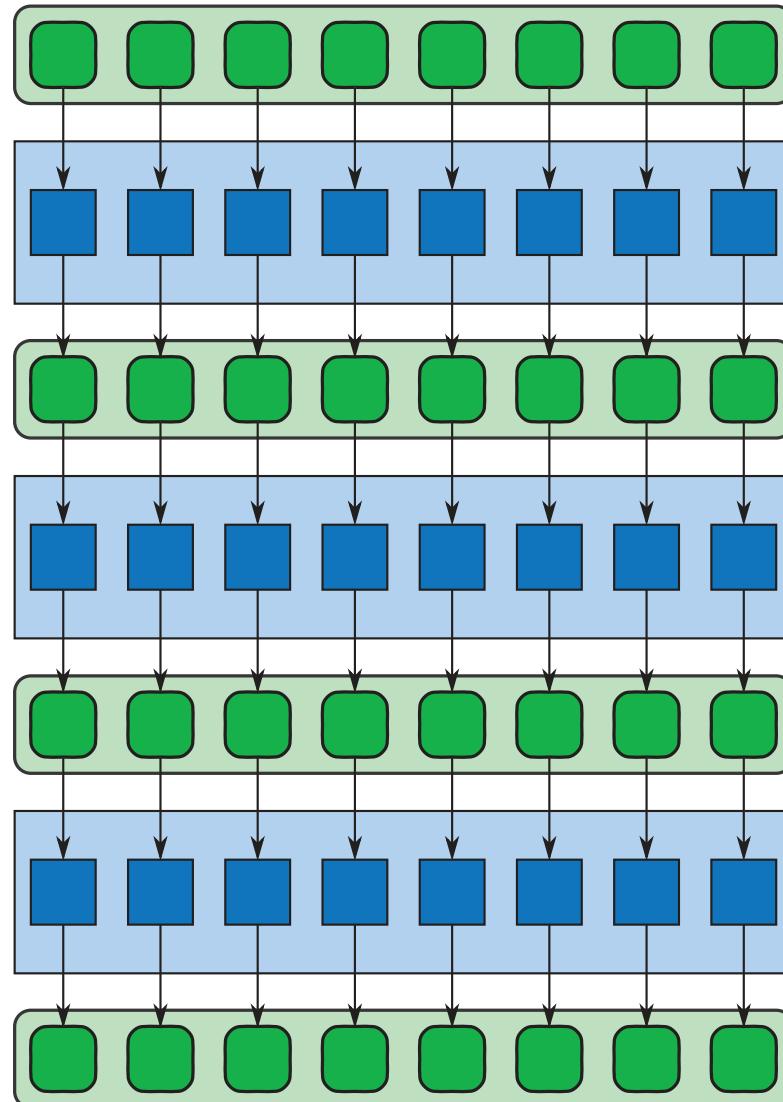
But, sometimes it makes sense to map over multiple collections at once...

Map with 2 Inputs, 1 Output



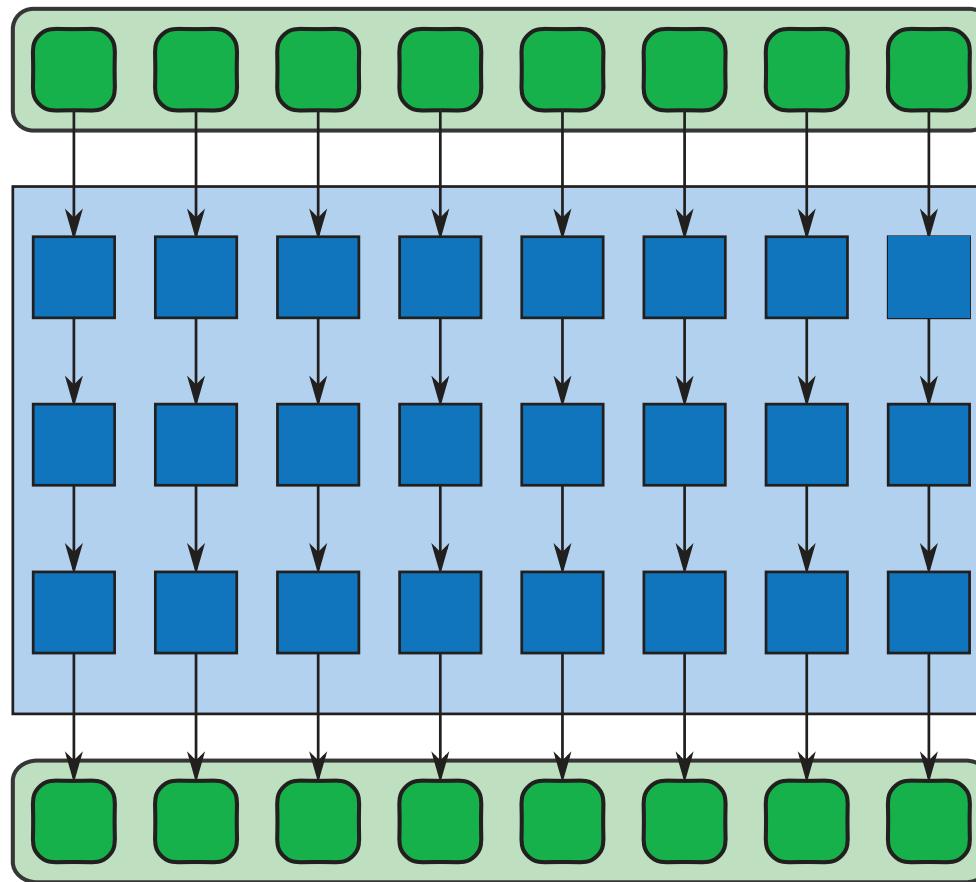
```
int twoToOne ( int x[11], int y[11] ) {  
    return x+y;  
}
```

Optimization – Sequences of Maps



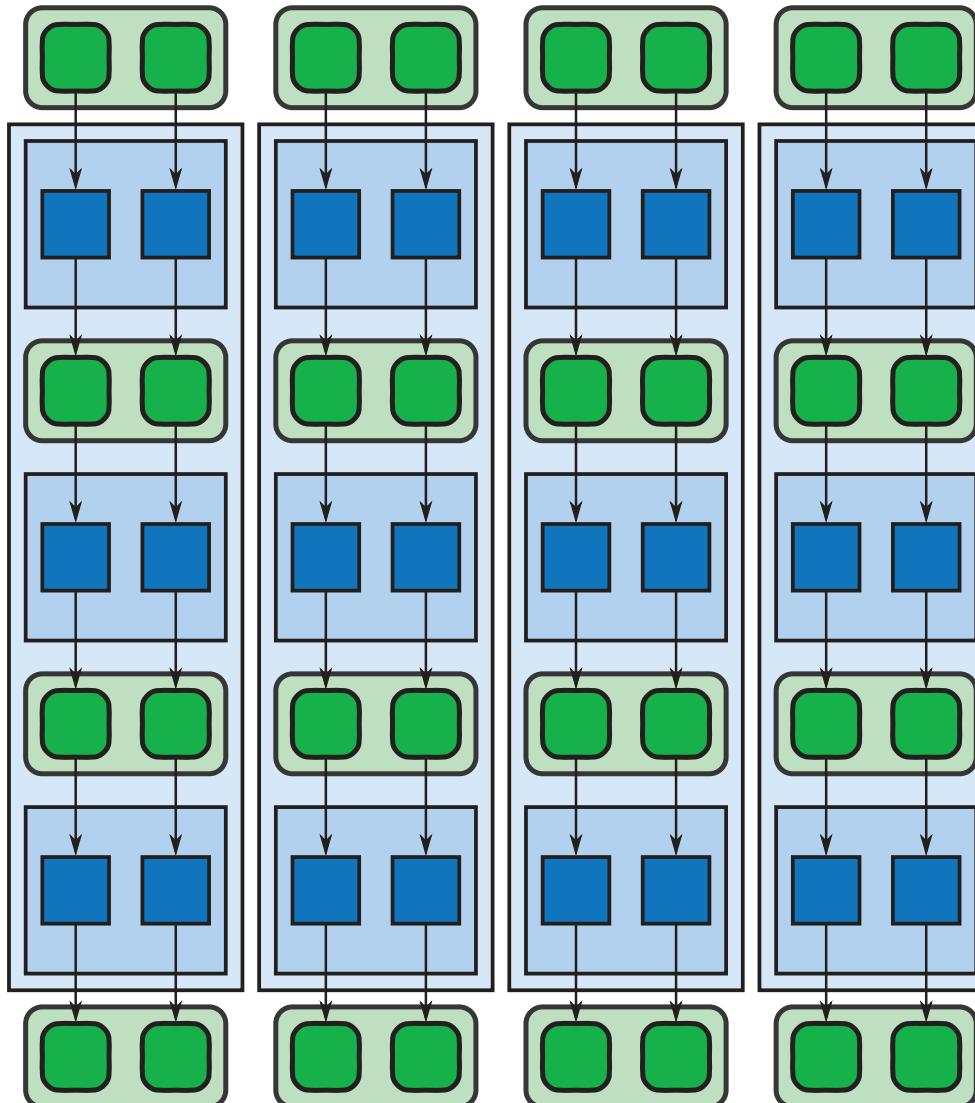
- Often several map operations occur in sequence
 - Vector math consists of many small operations such as additions and multiplications applied as maps
- A naïve implementation may write each intermediate result to memory, wasting memory BW and likely overwhelming the cache

Optimization – Code Fusion



- Can sometimes “fuse” together the operations to perform them at once
- Adds arithmetic intensity, reduces memory/cache usage
- Ideally, operations can be performed using registers alone

Optimization – Cache Fusion



- Sometimes impractical to fuse together the map operations
- Can instead break the work into blocks, giving each CPU one block at a time
- Hopefully, operations use cache alone

Related Patterns

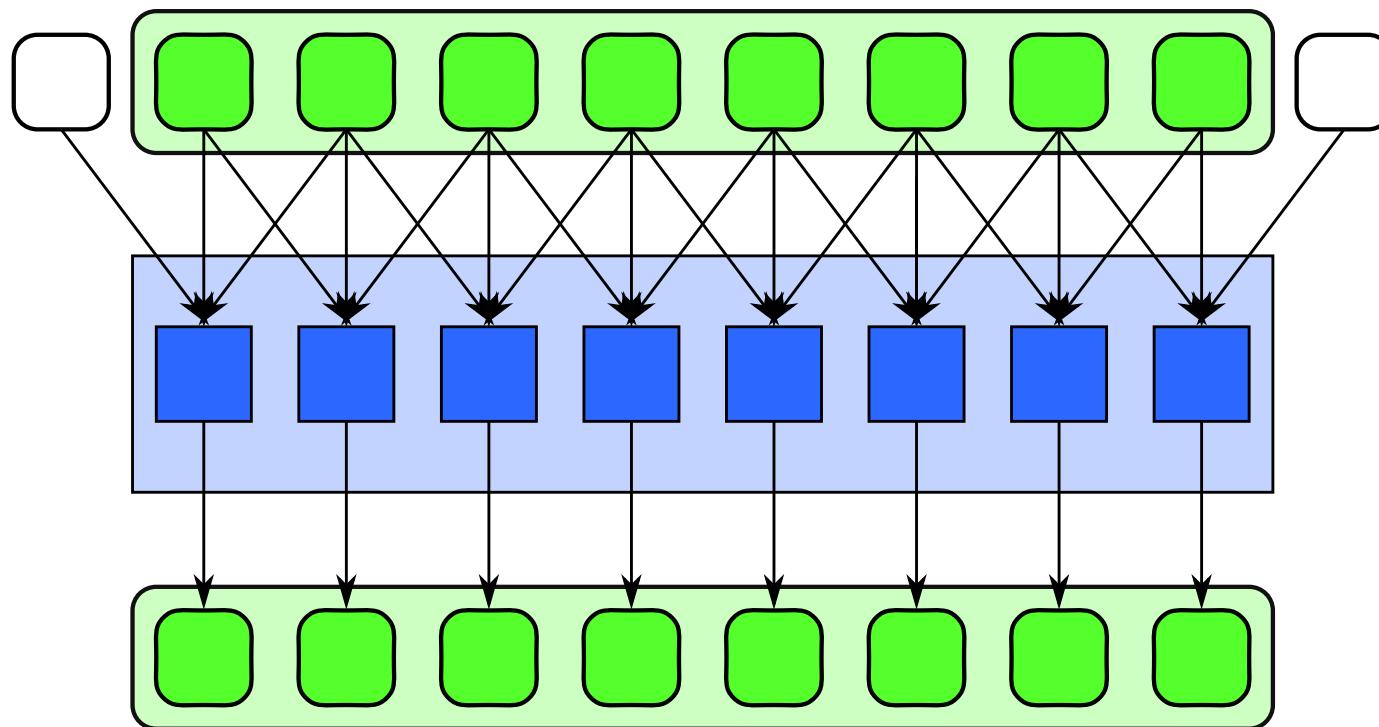
Three patterns related to map are discussed here:

- Stencil
- Workpile
- Divide-and-Conquer

More detail presented in a later lecture

Stencil

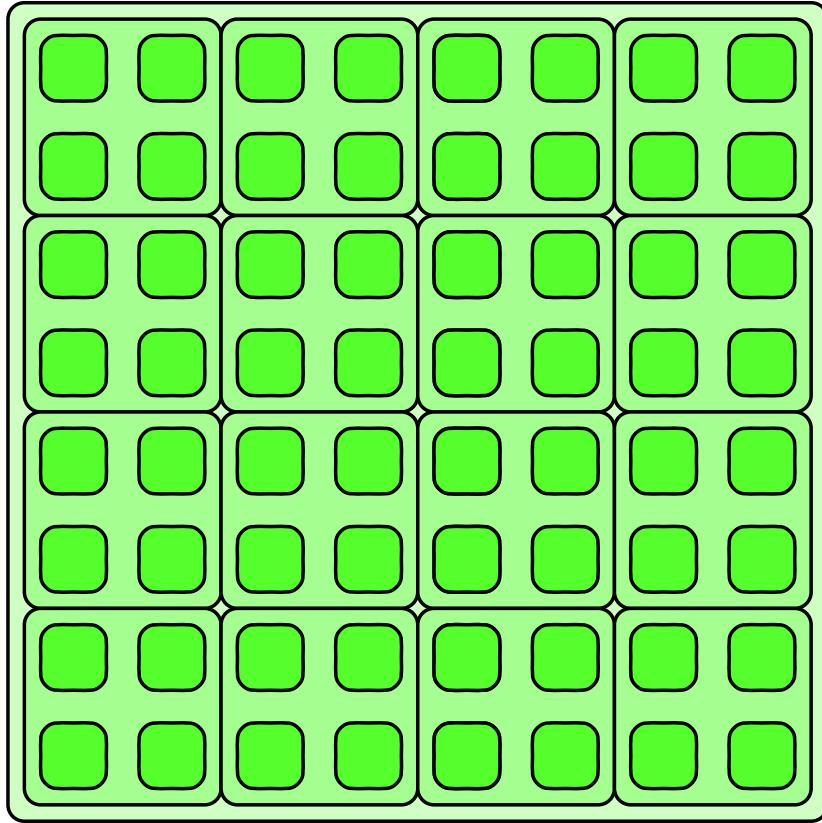
- Each instance of the map function accesses neighbors of its input, offset from its usual input
- Common in imaging and PDE solvers



Workpile

- Work items can be added to the map while it is in progress, from inside map function instances
- Work grows and is consumed by the map
- Workpile pattern terminates when no more work is available

Divide-and-Conquer

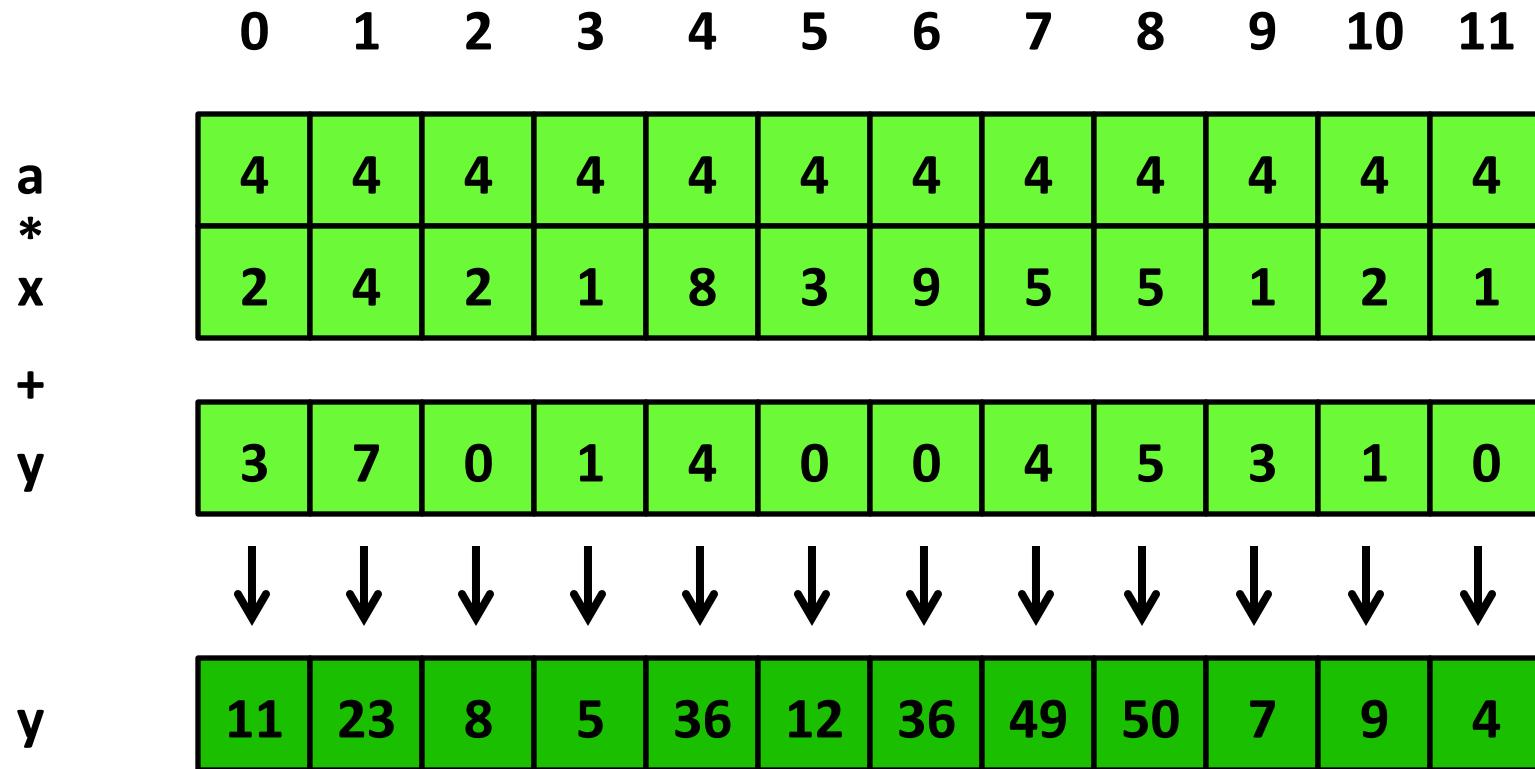


- Applies if a problem can be divided into smaller subproblems recursively until a base case is reached that can be solved serially

Example: Scaled Vector Addition (SAXPY)

- $y \leftarrow ax + y$
 - Scales vector x by a and adds it to vector y
 - Result is stored in input vector y
- Comes from the BLAS (Basic Linear Algebra Subprograms) library
- **Every element in vector x and vector y are independent**

What does $y \leftarrow ax + y$ look like?



Visual: $y \leftarrow ax + y$

	0	1	2	3	4	5	6	7	8	9	10	11
a	4	4	4	4	4	4	4	4	4	4	4	4
*	2	4	2	1	8	3	9	5	5	1	2	1
x	3	7	0	1	4	0	0	4	5	3	1	0
+												
y	11	23	8	5	36	12	36	49	50	7	9	4
y	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓

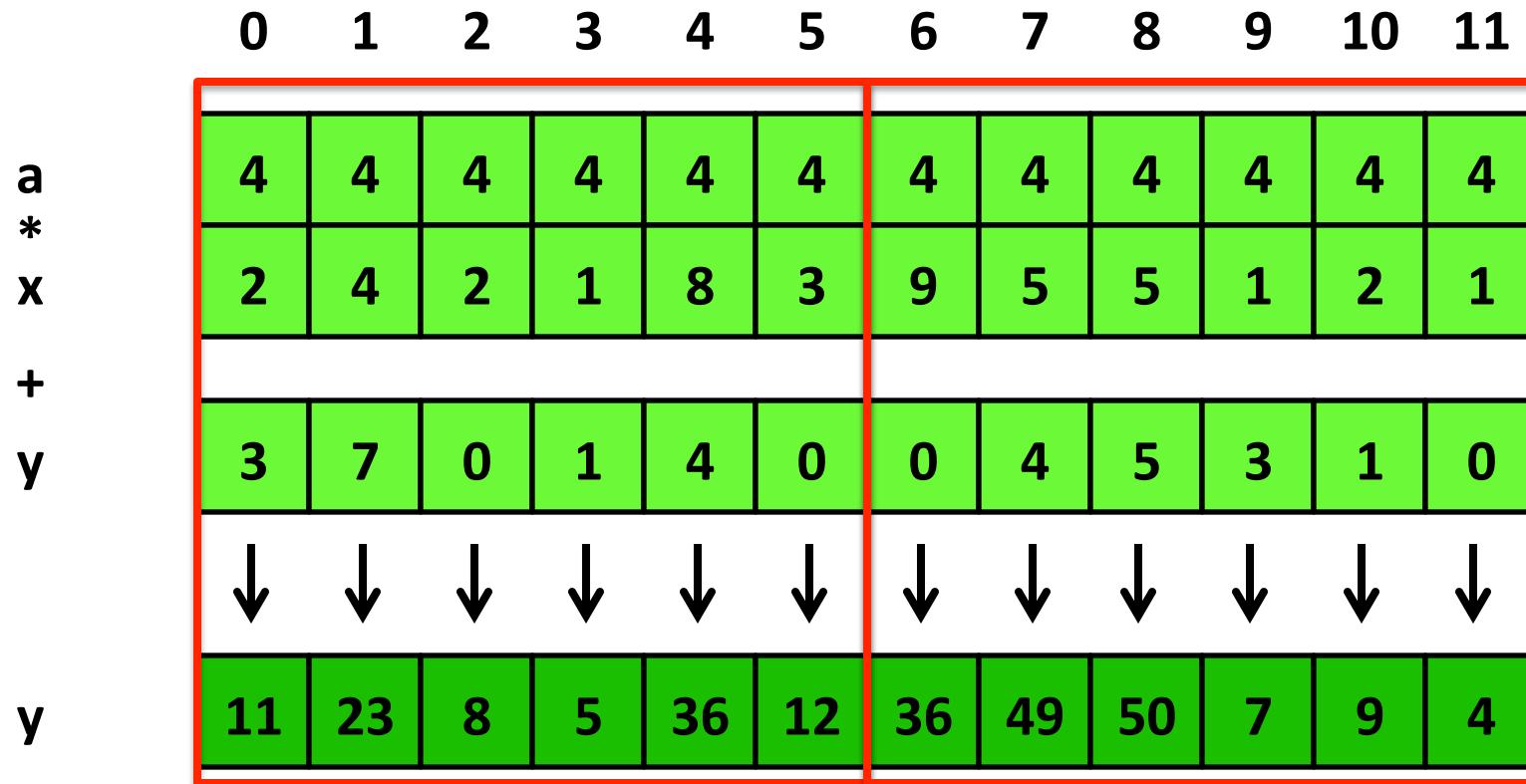
Twelve processors used → one for each element in the vector

Visual: $y \leftarrow ax + y$

	0	1	2	3	4	5	6	7	8	9	10	11
a	4	4	4	4	4	4	4	4	4	4	4	4
*	2	4	2	1	8	3	9	5	5	1	2	1
x												
+												
y	3	7	0	1	4	0	0	4	5	3	1	0
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
y	11	23	8	5	36	12	36	49	50	7	9	4

Six processors used → one for every two elements in the vector

Visual: $y \leftarrow ax + y$



Two processors used → one for every six elements in the vector

Serial SAXPY Implementation

```
1 void saxpy_serial(
2     size_t n,           // the number of elements in the vectors
3     float a,            // scale factor
4     const float x[],   // the first input vector
5     float y[]           // the output vector and second input vector
6 ) {
7     for (size_t i = 0; i < n; ++i)
8         y[i] = a * x[i] + y[i];
9 }
```

TBB SAXPY Implementation

```
1 void saxpy_tbb(
2     int n,          // the number of elements in the vectors
3     float a,        // scale factor
4     float x[],     // the first input vector
5     float y[]      // the output vector and second input vector
6 ) {
7     tbb::parallel_for(
8         tbb::blocked_range<int>(0, n),
9         [&](tbb::blocked_range<int> r) {
10             for (size_t i = r.begin(); i != r.end(); ++i)
11                 y[i] = a * x[i] + y[i];
12         }
13     );
14 }
```

Cilk Plus SAXPY Implementation

```
1 void saxpy_cilk(
2     int n,          // the number of elements in the vectors
3     float a,        // scale factor
4     float x[],     // the first input vector
5     float y[]      // the output vector and second input vector
6 ) {
7     cilk_for (int i = 0; i < n; ++i)
8         y[i] = a * x[i] + y[i];
9 }
```

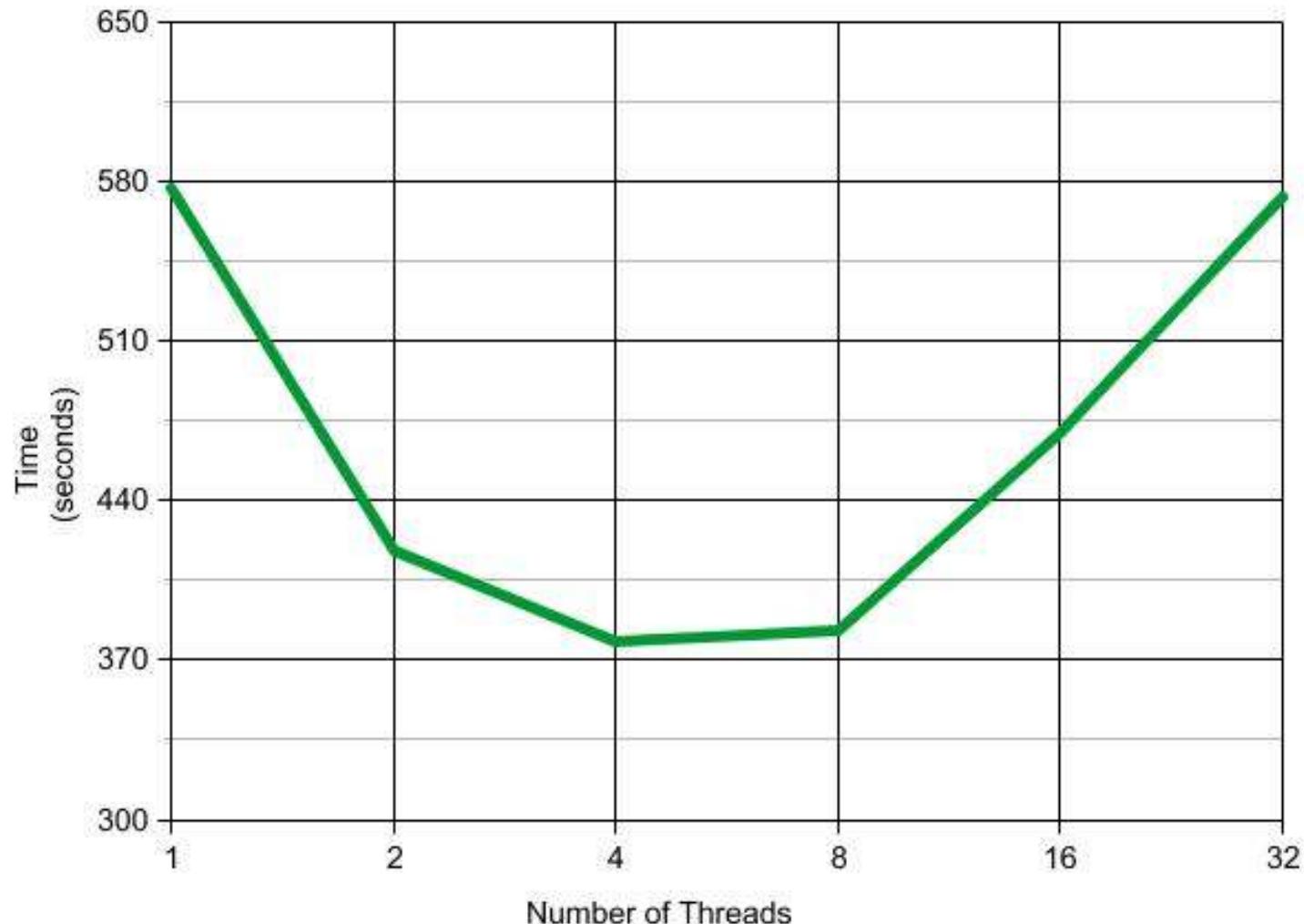
OpenMP SAXPY Implementation

```
1 void saxpy_openmp(
2     int n,          // the number of elements in the vectors
3     float a,        // scale factor
4     float x[],     // the first input vector
5     float y[])     // the output vector and second input vector
6 ) {
7 #pragma omp parallel for
8     for (int i = 0; i < n; ++i)
9         y[i] = a * x[i] + y[i];
10 }
```

OpenMP SAXPY Performance

SAXPY on a NUC

Vector size = 500,000,000





Collectives Pattern

Parallel Computing

CIS 410/510

Department of Computer and Information Science



UNIVERSITY OF OREGON

Outline

- What are Collectives?
- Reduce Pattern
- Scan Pattern
- Sorting

Collectives

- **Collective** operations deal with a *collection* of data as a whole, rather than as separate elements
- Collective patterns include:
 - **Reduce**
 - **Scan**
 - **Partition**
 - **Scatter**
 - **Gather**

Collectives

- **Collective** operations deal with a *collection* of data as a whole, rather than as separate elements
- Collective patterns include:
 - **Reduce**
 - **Scan**
 - **Partition**
 - **Scatter**
 - **Gather**

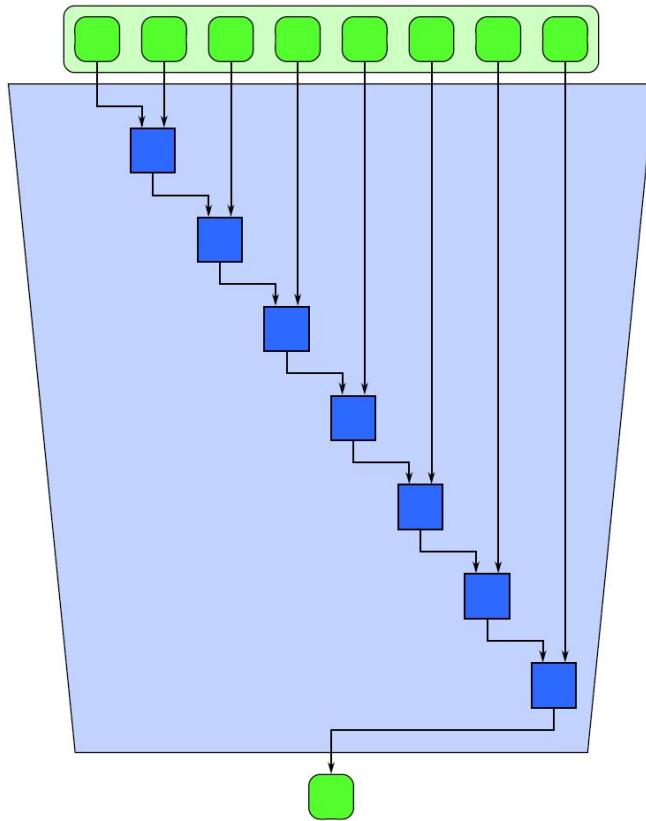
Reduce and Scan will be covered in this lecture

Reduce

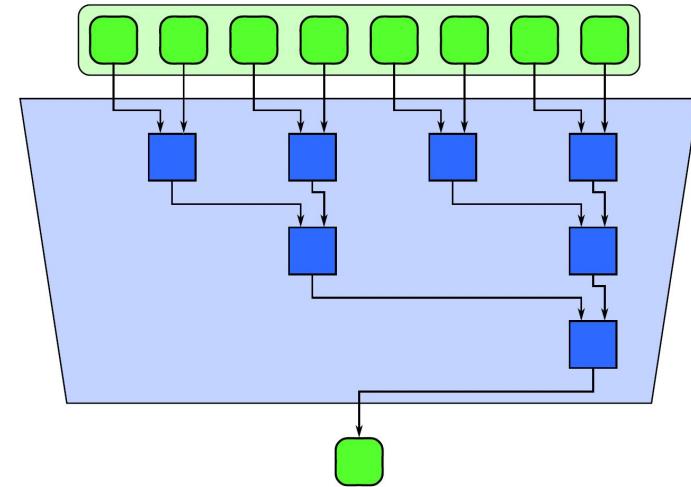
- **Reduce** is used to combine a collection of elements into one summary value
- A combiner function combines elements pairwise
- A combiner function only needs to be *associative* to be parallelizable
- Example combiner functions:
 - Addition
 - Multiplication
 - Maximum / Minimum

Reduce

Serial Reduction

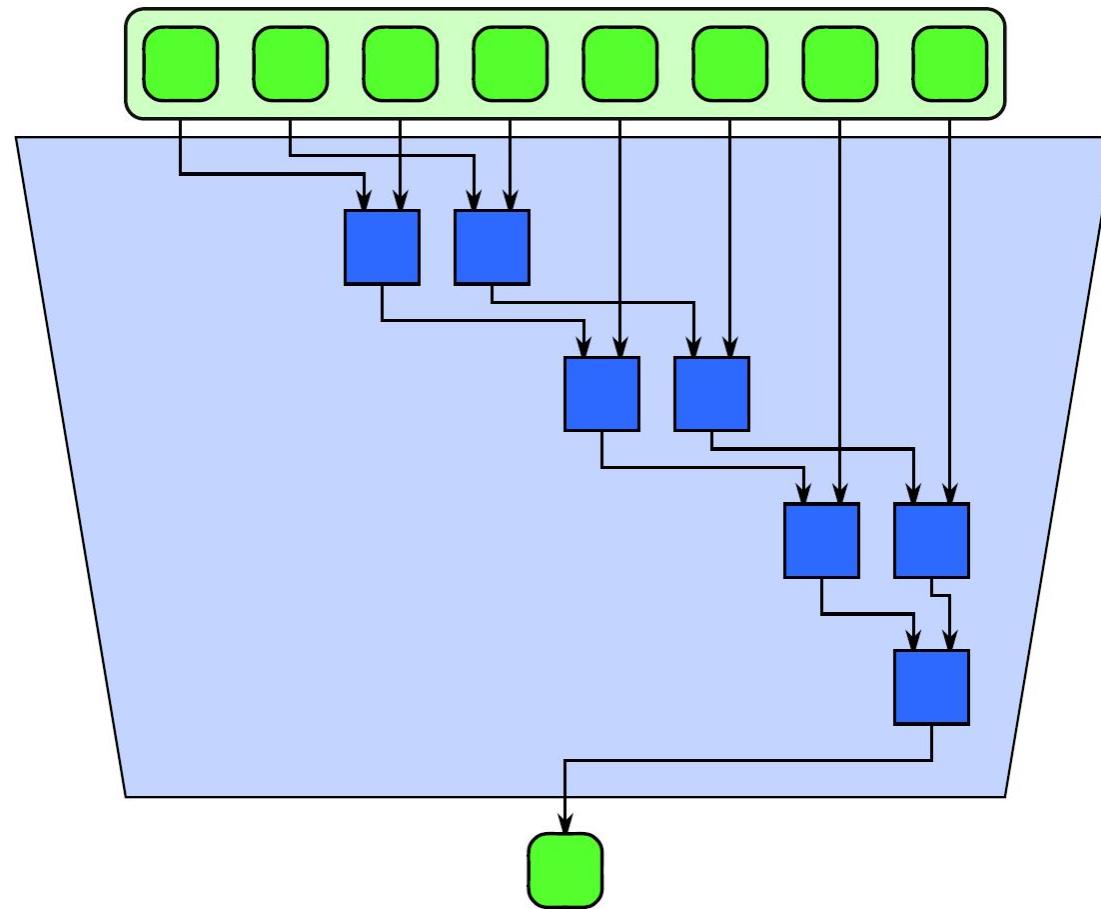


Parallel Reduction



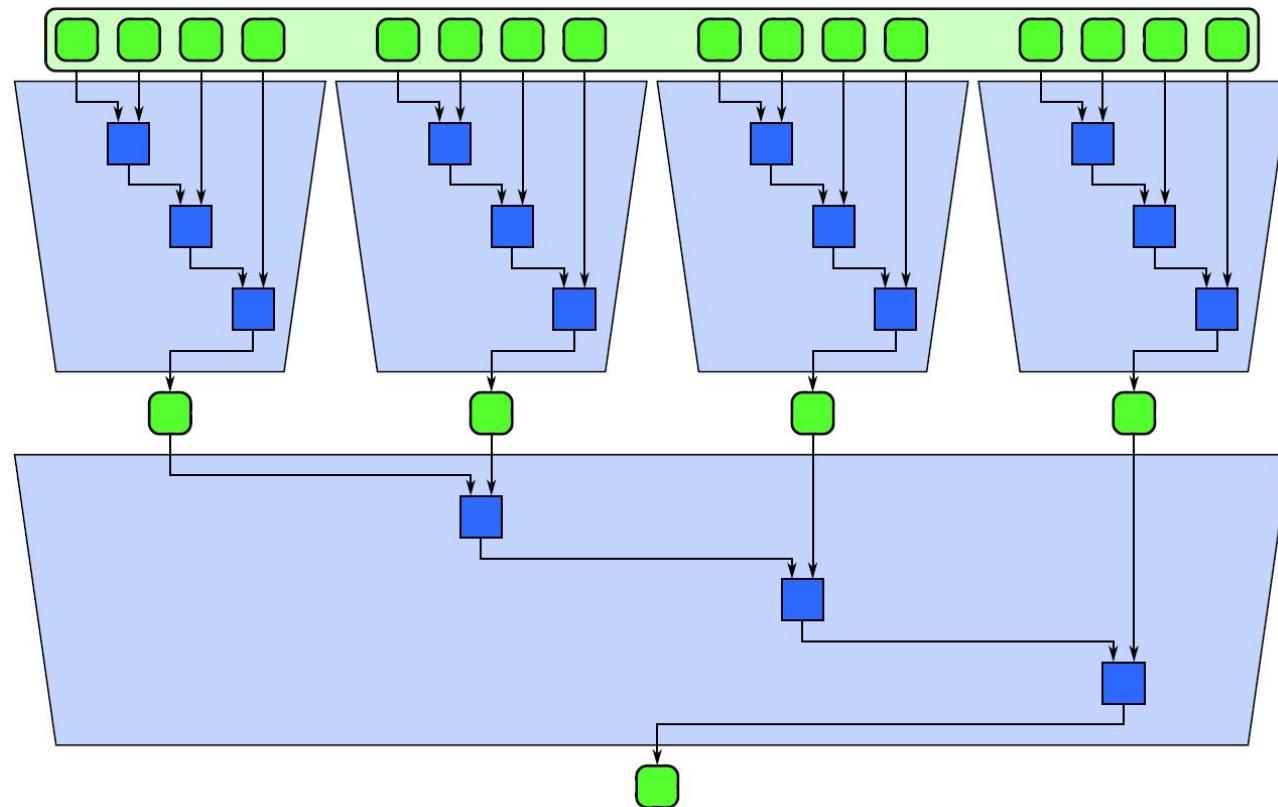
Reduce

□ Vectorization

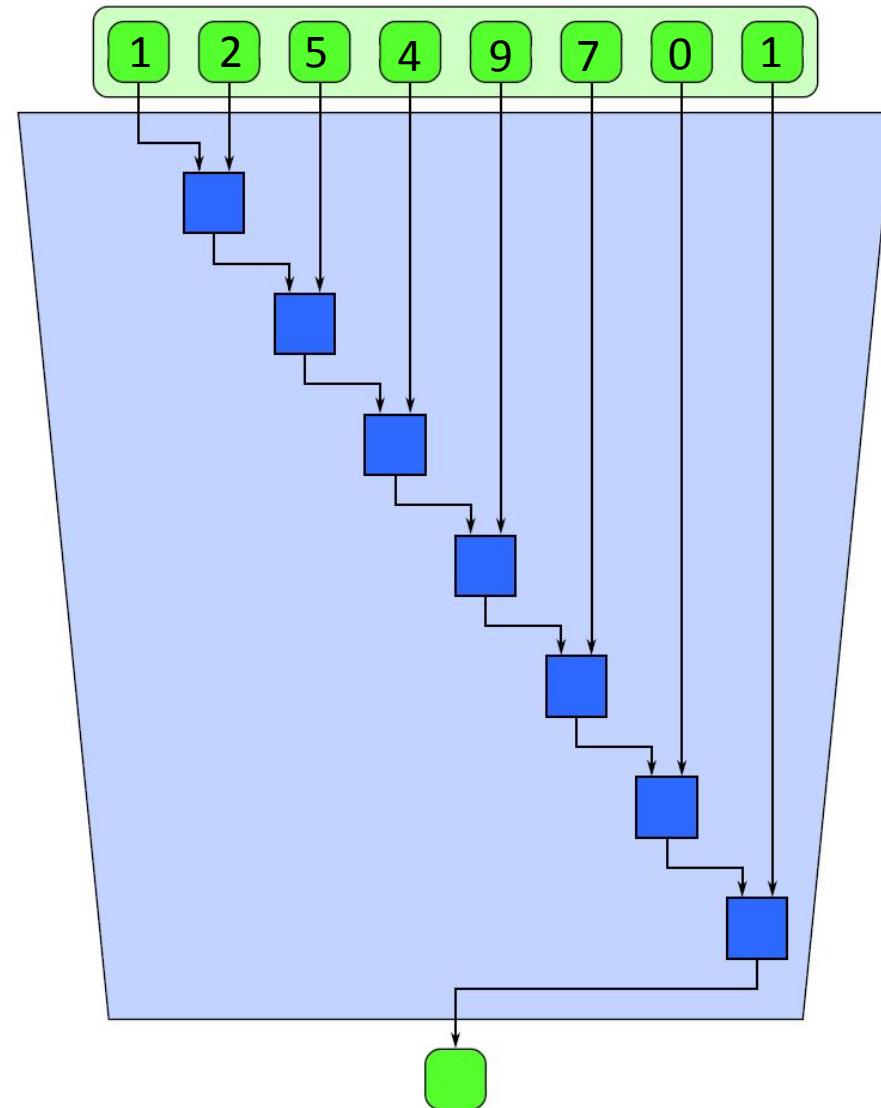


Reduce

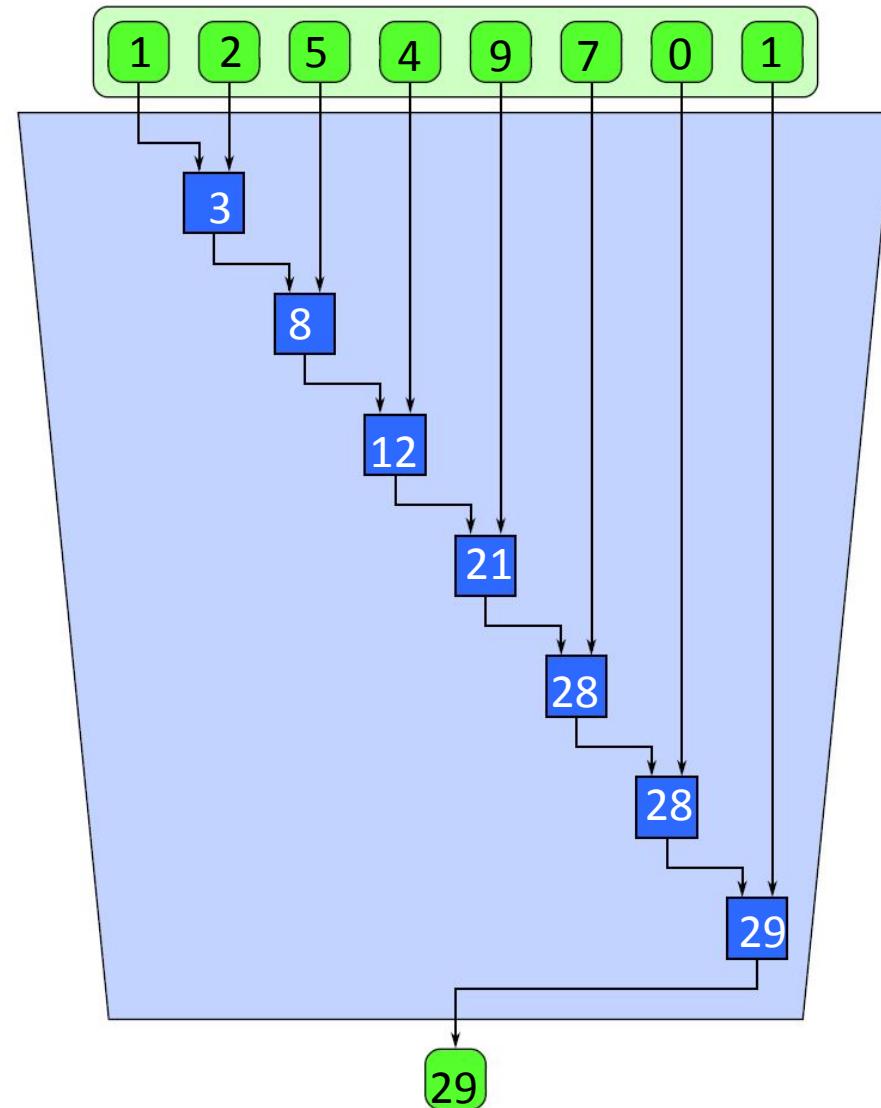
- **Tiling** is used to break chunks of work up for workers to reduce serially



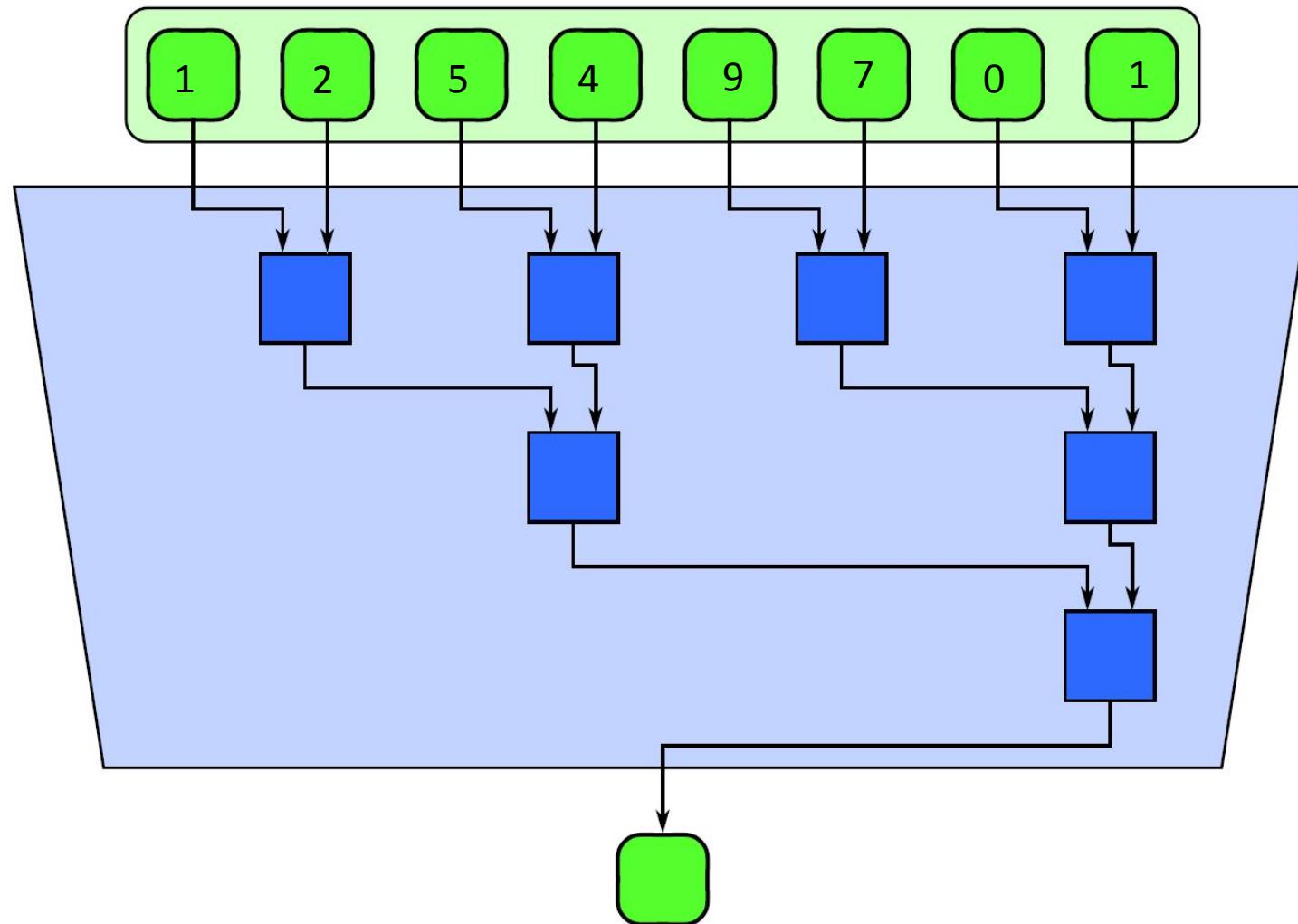
Reduce – Add Example



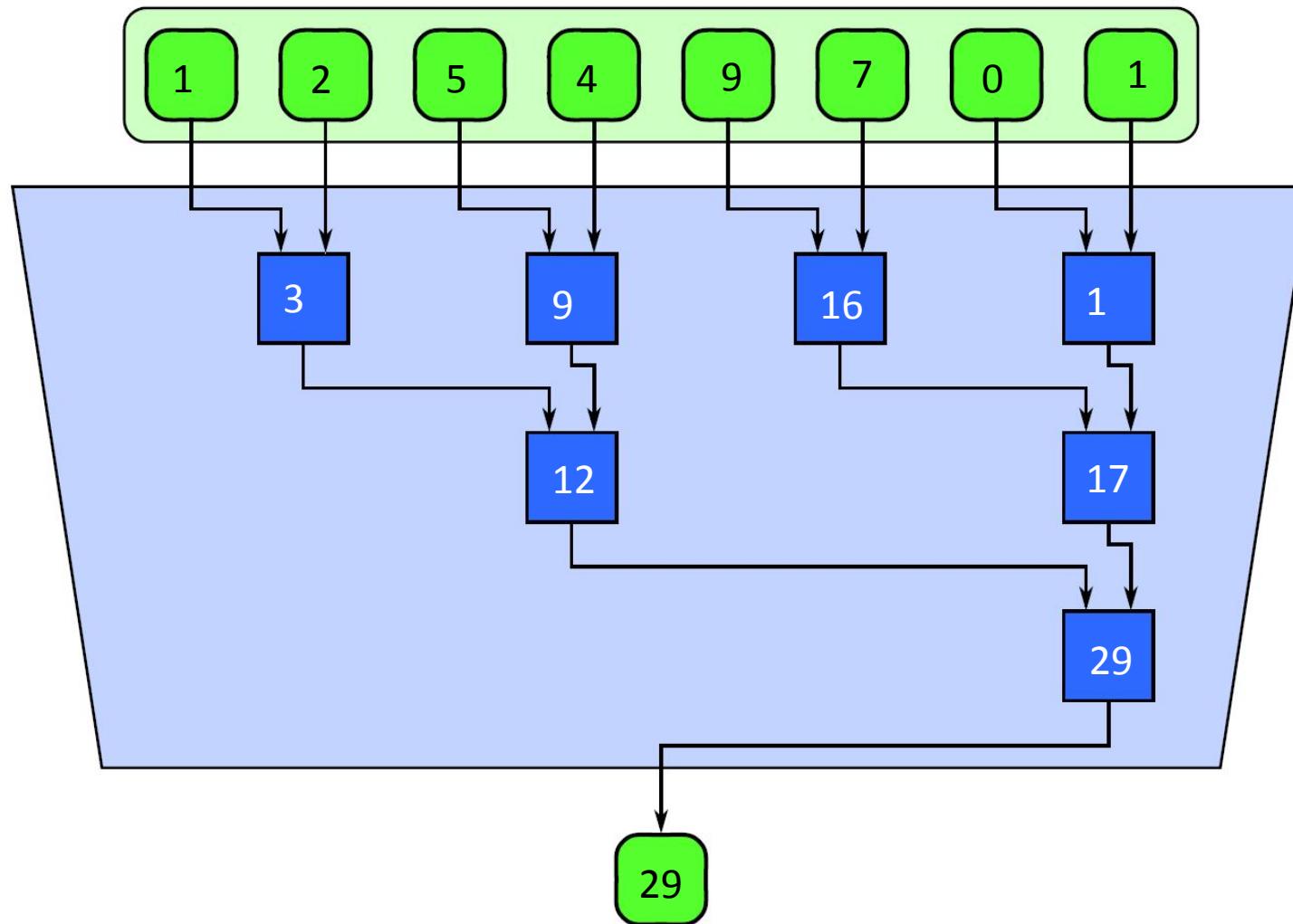
Reduce – Add Example



Reduce – Add Example

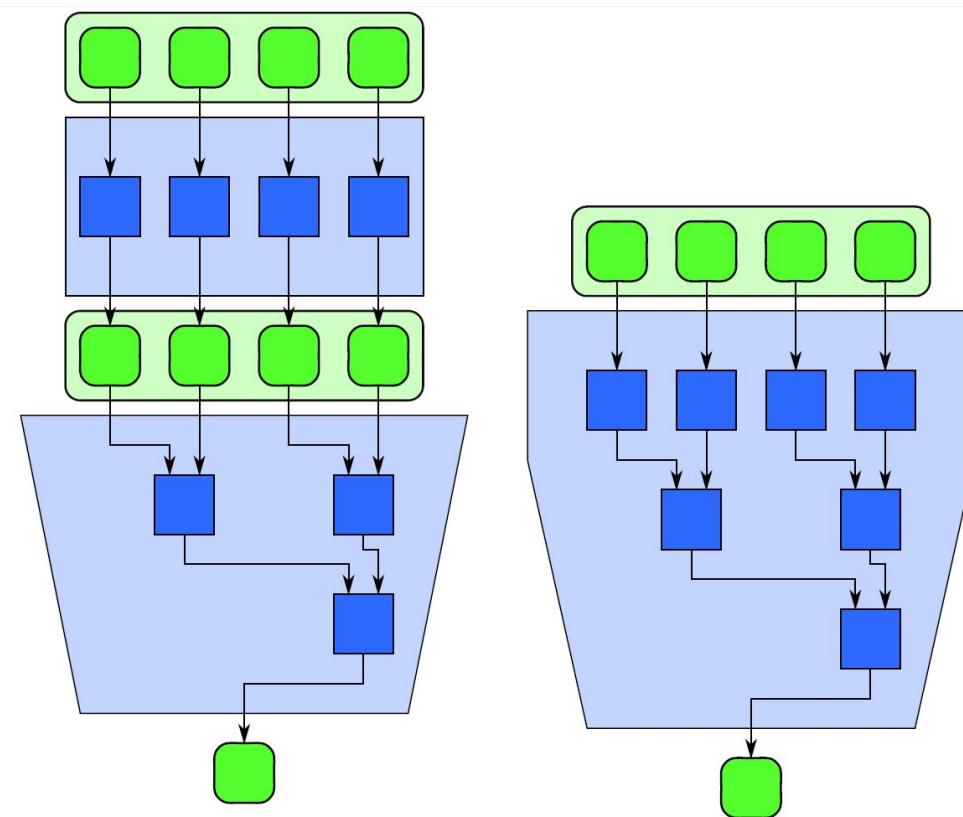


Reduce – Add Example



Reduce

- We can “fuse” the map and reduce patterns



Reduce

- Precision can become a problem with reductions on floating point data
- Different orderings of floating point data can change the reduction value

Reduce Example: Dot Product

- 2 vectors of same length
- Map (*) to multiply the components
- Then reduce with (+) to get the final answer

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i b_i.$$

Also: $\vec{a} \cdot \vec{b} = |\vec{a}| \cos(\theta) |\vec{b}|$

Dot Product – Example Uses

- Essential operation in physics, graphics, video games,...
- Gaming analogy: in Mario Kart, there are “boost pads” on the ground that increase your speed
 - red vector is your speed (x and y direction)
 - blue vector is the orientation of the boost pad (x and y direction). Larger numbers are more power.

How much boost will you get? For the analogy, imagine the pad multiplies your speed:

- If you come in going 0, you'll get nothing
- If you cross the pad perpendicularly, you'll get 0 [just like the banana obliteration, it will give you 0x boost in the perpendicular direction]



$$\text{Total} = \text{speed}_x \cdot \text{boost}_x + \text{speed}_y \cdot \text{boost}_y$$

Ref: <http://betterexplained.com/articles/vector-calculus-understanding-the-dot-product/>

Scan

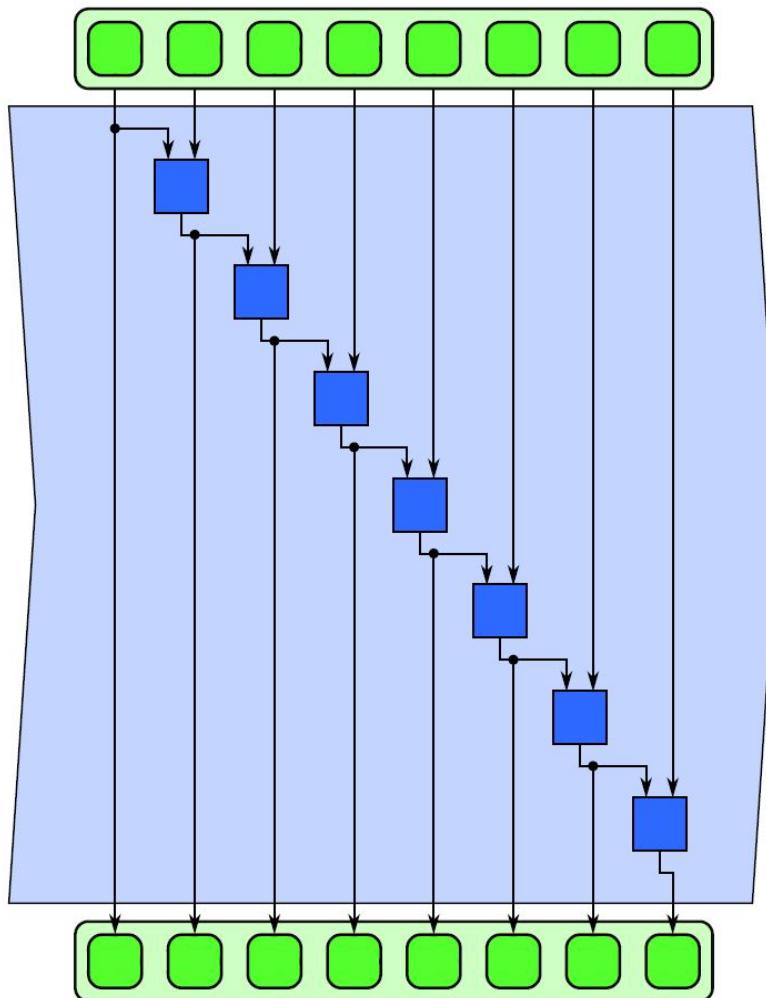
- The **scan** pattern produces partial reductions of input sequence, generates new sequence
- Trickier to parallelize than reduce
- Inclusive scan vs. exclusive scan
 - Inclusive scan: includes current element in partial reduction
 - Exclusive scan: excludes current element in partial reduction, partial reduction is of all prior elements prior to current element

Scan – Example Uses

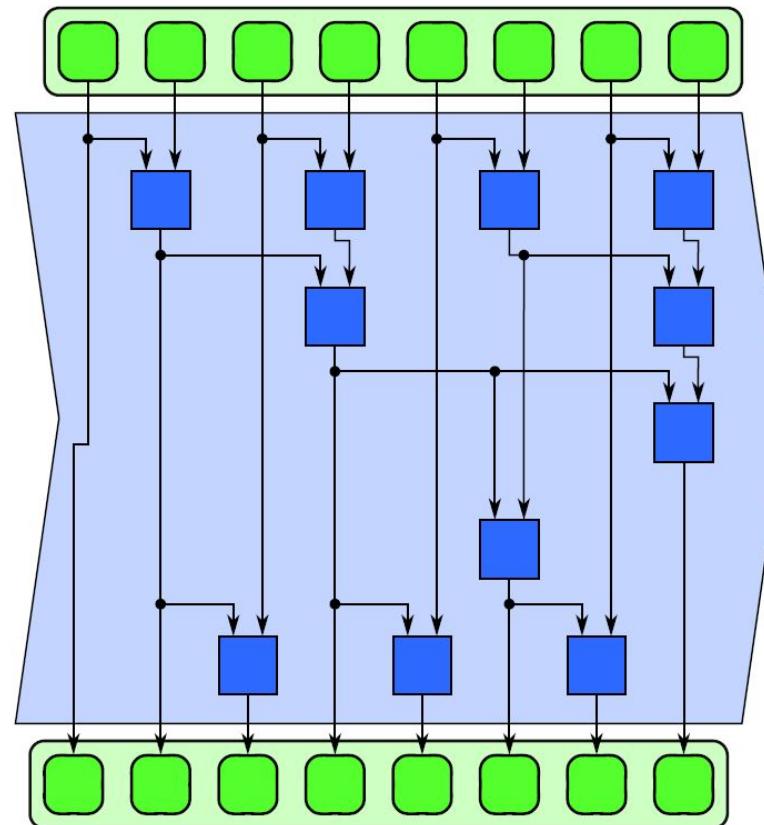
- Lexical comparison of strings – e.g., determine that “strategy” should appear before “stratification” in a dictionary
- Add multi-precision numbers (those that cannot be represented in a single machine *word*)
- Evaluate polynomials
- Implement radix sort or quicksort
- Delete marked elements in an array
- Dynamically allocate processors
- Lexical analysis – parsing programs into tokens
- Searching for regular expressions
- Labeling components in 2-D images
- Some tree algorithms – e.g., finding the depth of every vertex in a tree

Scan

Serial Scan

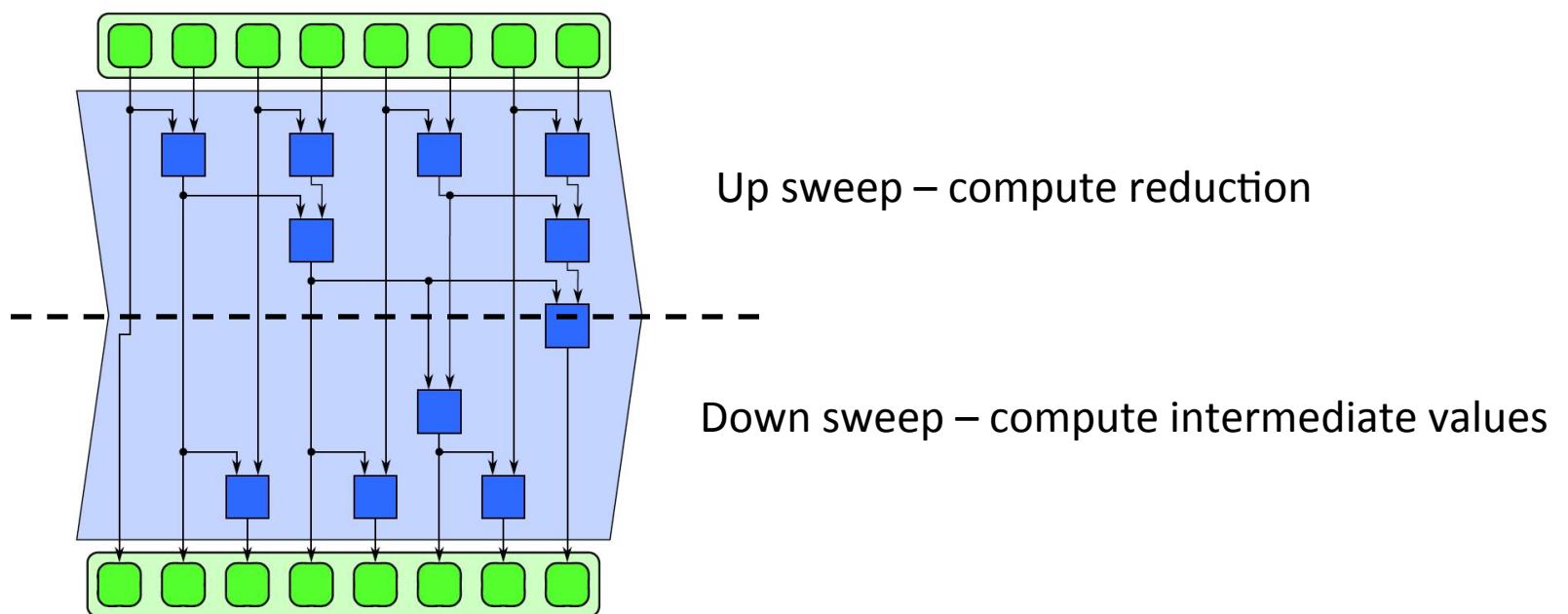


Parallel Scan

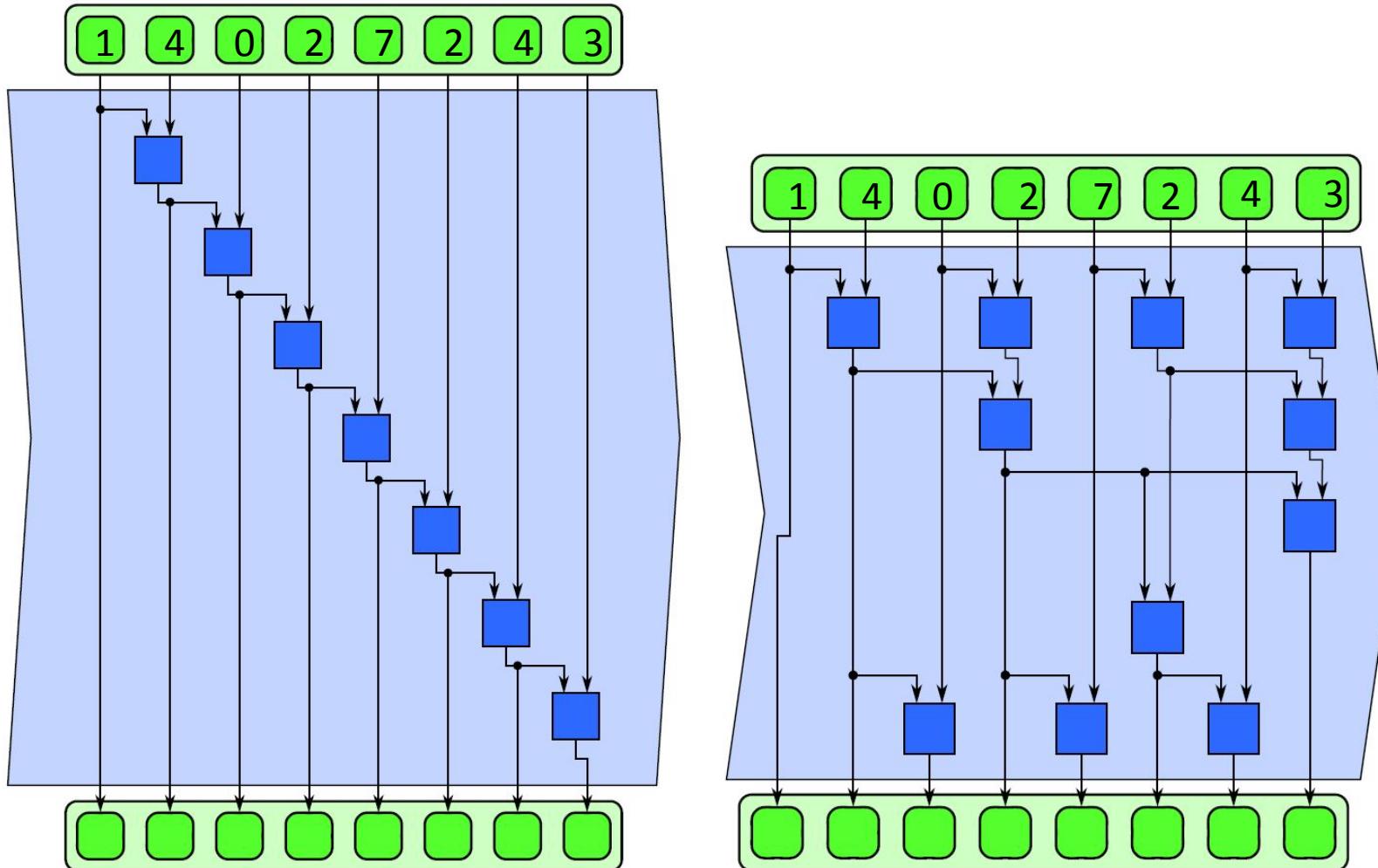


Scan

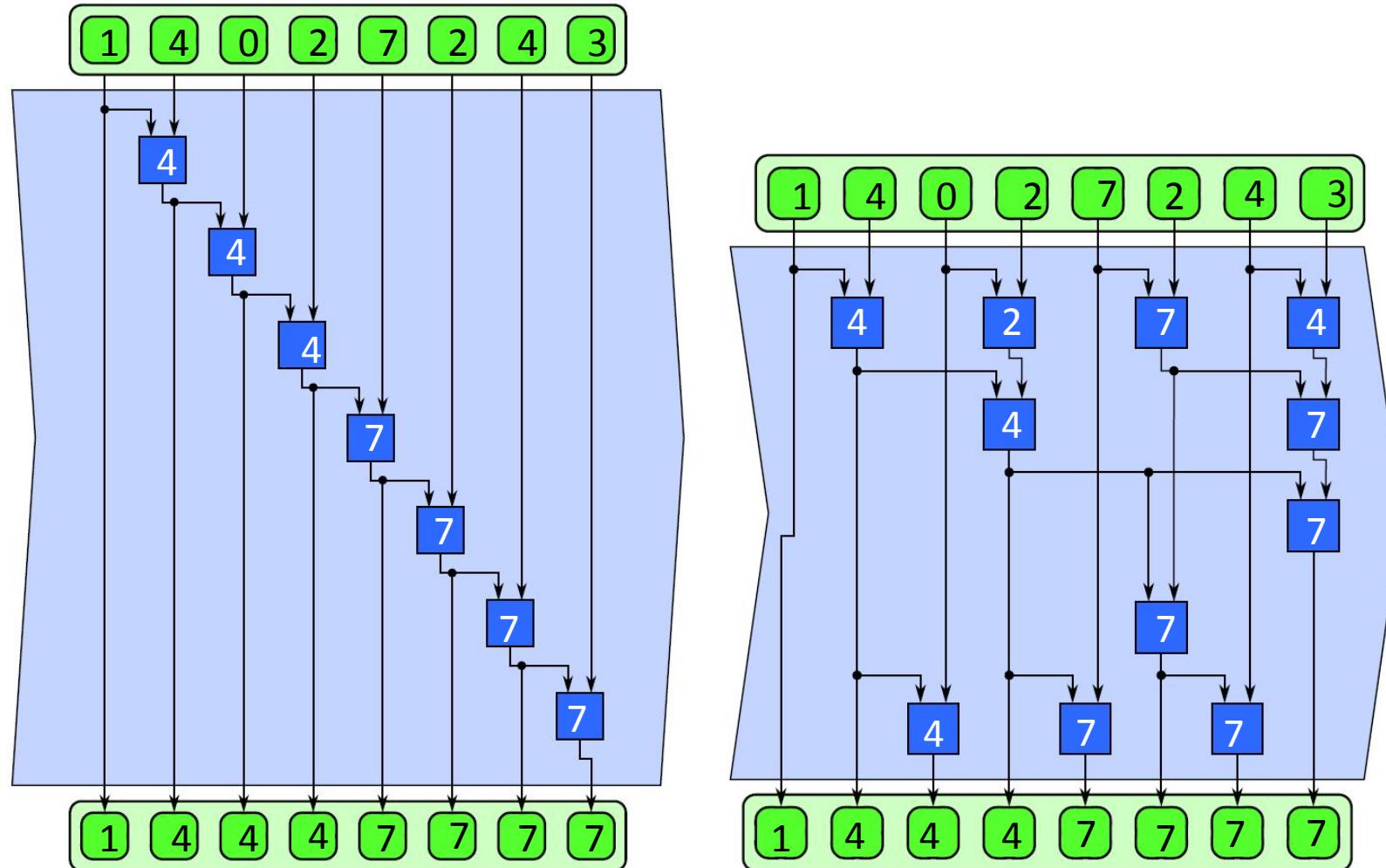
- One algorithm for parallelizing scan is to perform an “up sweep” and a “down sweep”
- Reduce the input on the up sweep
- The down sweep produces the intermediate results



Scan – Maximum Example



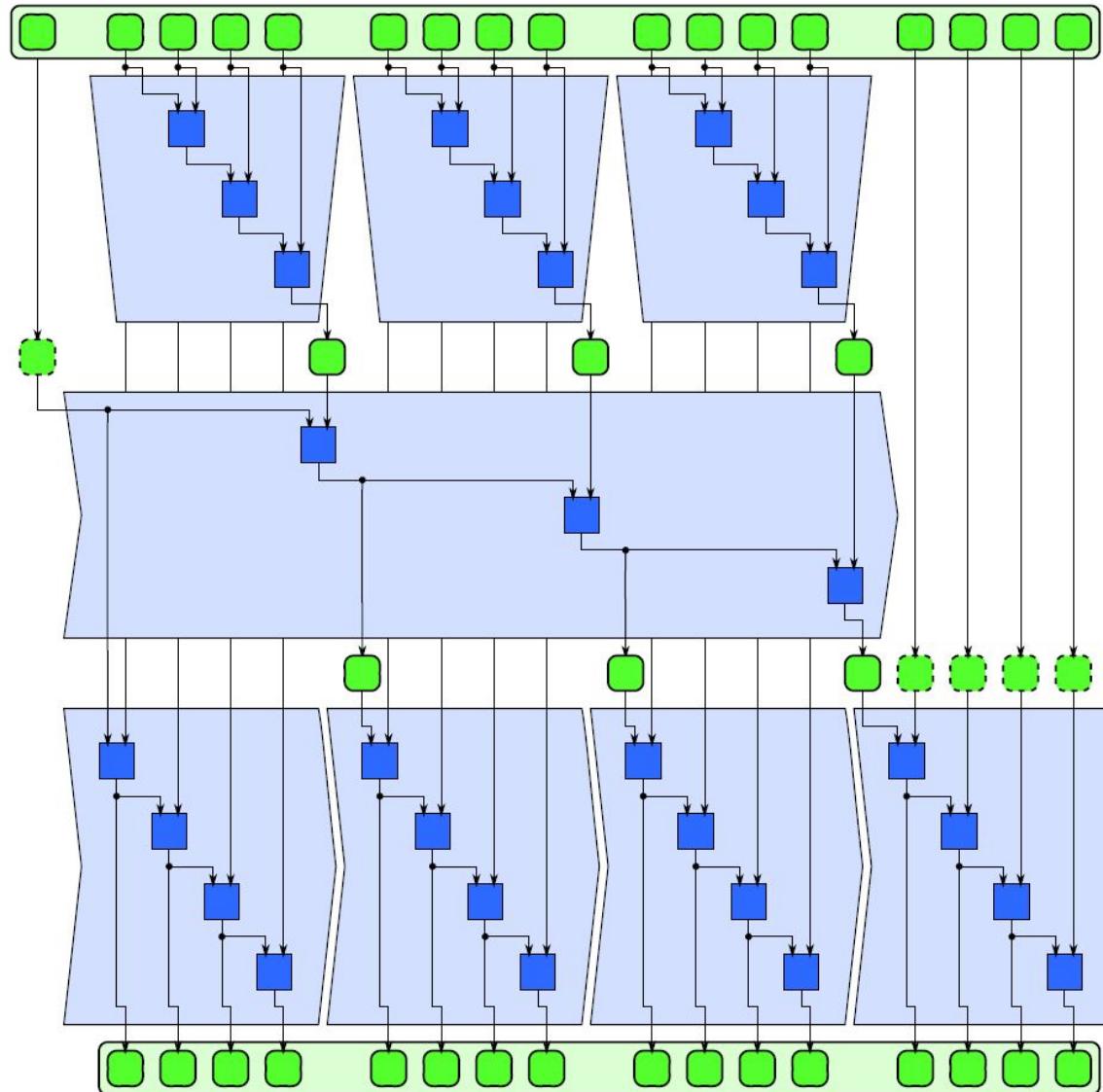
Scan – Maximum Example



Scan

- Three phase scan with tiling

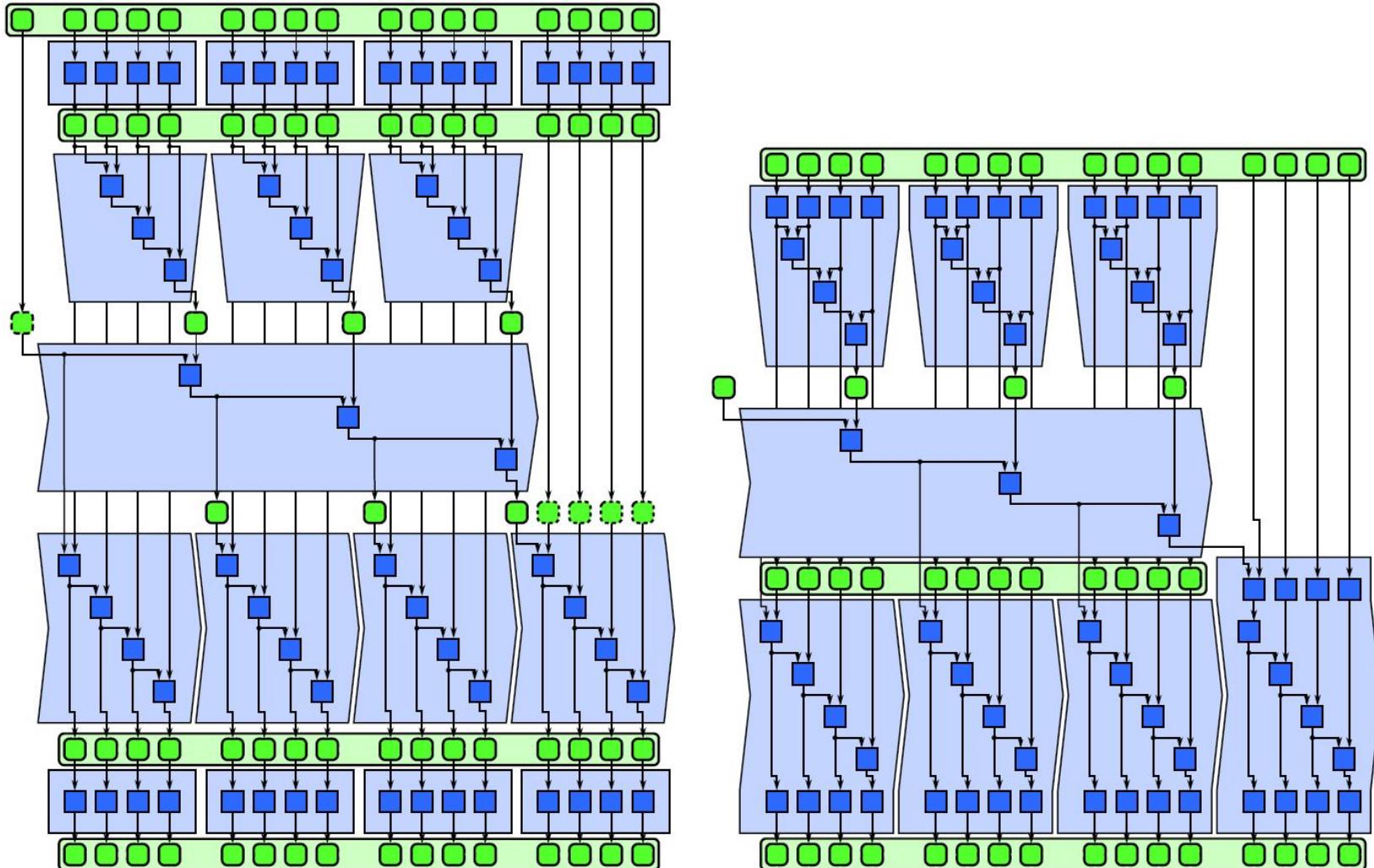
Scan



Scan

- Just like reduce, we can also fuse the **map** pattern with the **scan** pattern

Scan



Merge Sort as a reduction

- We can sort an array via a pair of a map and a reduce
- Map each element into a vector containing just that element
 - \diamond is the merge operation: $[1,3,5,7] \diamond [2,6,15] = [1,2,3,5,6,7,15]$
 - $[]$ is the empty list
- How fast is this?

Right Biased Sort

Start with [14,3,4,8,7,52,1]

Map to [[14],[3],[4],[8],[7],[52],[1]]

Reduce:

$$\begin{aligned} & [14] \diamond ([3] \diamond ([4] \diamond ([8] \diamond ([7] \diamond ([52] \diamond [1]))))) \\ &= [14] \diamond ([3] \diamond ([4] \diamond ([8] \diamond ([7] \diamond [1,52])))) \\ &= [14] \diamond ([3] \diamond ([4] \diamond ([8] \diamond [1,7,52]))) \\ &= [14] \diamond ([3] \diamond ([4] \diamond [1,7,8,52])) \\ &= [14] \diamond ([3] \diamond [1,4,7,8,52]) \\ &= [14] \diamond [1,3,4,7,8,52] \\ &= [1,3,4,7,8,14,52] \end{aligned}$$

Right Biased Sort Cont

- How long did that take?
- We did $O(n)$ merges...but each one took $O(n)$ time
- $O(n^2)$
- We wanted merge sort, but instead we got insertion sort!

Tree Shape Sort

Start with [14,3,4,8,7,52,1]

Map to [[14],[3],[4],[8],[7],[52],[1]]

Reduce:

$$\begin{aligned} &(([14] \diamond [3]) \diamond ([4] \diamond [8])) \diamond(([7] \diamond [52]) \diamond [1]) \\ &= ([3,14] \diamond [4,8]) \diamond ([7,52] \diamond [1]) \\ &= [3,4,8,14] \diamond [1,7,52] \\ &= [1,3,4,7,8,14,52] \end{aligned}$$

Tree Shaped Sort Performance

- Even if we only had a single processor this is better
 - We do $O(\log n)$ merges
 - Each one is $O(n)$
 - So $O(n * \log(n))$
- But opportunity for parallelism is not so great
 - $O(n)$ assuming sequential merge
 - Takeaway: the shape of reduction matters!



Data Reorganization Pattern

Parallel Computing

CIS 410/510

Department of Computer and Information Science



UNIVERSITY OF OREGON

Outline

- Gather Pattern
 - Shifts, Zip, Unzip
- Scatter Pattern
 - Collision Rules: atomic, permutation, merge, priority
- Pack Pattern
 - Split, Unsplit, Bin
 - Fusing Map and Pack
 - Expand
- Partitioning Data
- AoS vs. SoA
- Example Implementation: AoS vs. SoA

Data Movement

- Performance is often more limited by data movement than by computation
 - Transferring data across memory layers is costly
 - ◆ locality is important to minimize data access times
 - ◆ data organization and layout can impact this
 - Transferring data across networks can take many cycles
 - ◆ attempting to minimize the # messages and overhead is important
 - Data movement also costs more in power
- For “data intensive” application, it is a good idea to design the data movement first
 - Design the computation around the data movements
 - Applications such as search and sorting are all about data movement and reorganization

Parallel Data Reorganization

- Remember we are looking to do things in parallel
- How to be faster than the sequential algorithm?
- Similar consistency issues arise as when dealing with computation parallelism
- Here we are concerned more with parallel data movement and management issues
- Might involve the creation of additional data structures (e.g., for holding intermediate data)

Gather Pattern

- Gather pattern creates a (source) collection of data by reading from another (input) data collection
 - Given a collection of (ordered) indices
 - Read data from the source collection at each index
 - Write data to the output collection in index order
- Transfers from source collection to output collection
 - Element type of output collection is the same as the source
 - Shape of the output collection is that of the index collection
 - ◆ same dimensionality
- Can be considered a combination of map and random serial read operations
 - Essentially does a number of random reads in parallel

Gather: Serial Implementation

```
1 template<typename Data, typename Idx>
2 void gather(
3     size_t n, //number of elements in data collection
4     size_t m, //number of elements in index collection
5     Data a[], //input data collection (n elements)
6     Data A[], //output data collection (m elements)
7     Idx idx[] //input index collection (m elements)
8 ) {
9     for (size_t i = 0; i < m; ++i) {
10         size_t j = idx[i]; //get ith index
11         assert(0 <= j && j < n); //check array bounds
12         A[i] = a[j]; //perform random read
13     }
14 }
```

Serial implementation of gather in pseudocode

Gather: Serial Implementation

```
1 template<typename Data, typename Idx>
2 void gather(
3     size_t n, //number of elements in data collection
4     size_t m, //number of elements in index collection
5     Data a[], //input data collection (n elements)
6     Data A[], //output data collection (m elements)
7     Idx idx[] //input index collection (m elements)
8 ) {
9     for (size_t i = 0; i < m; ++i) {
10         size_t j = idx[i]; //get ith index
11         assert(0 <= j && j < n); //check array bounds
12         A[i] = a[j]; //perform random read
13     }
14 }
```

Serial implementation of gather in pseudocode

Do you see opportunities for parallelism?

Gather: Serial Implementation

```
1 template<typename Data, typename Idx>
2 void gather(
3     size_t n, //number of elements in data collection
4     size_t m, //number of elements in index collection
5     Data a[], //input data collection (n elements)
6     Data A[], //output data collection (m elements)
7     Idx idx[] //input index collection (m elements)
8 ) {
9     for (size_t i = 0; i < m; ++i) {
10         size_t j = idx[i]; //get ith index
11         assert(0 <= i && i < n); //check array bounds
12         A[i] = a[j]; //perform random read
13     }
14 }
```

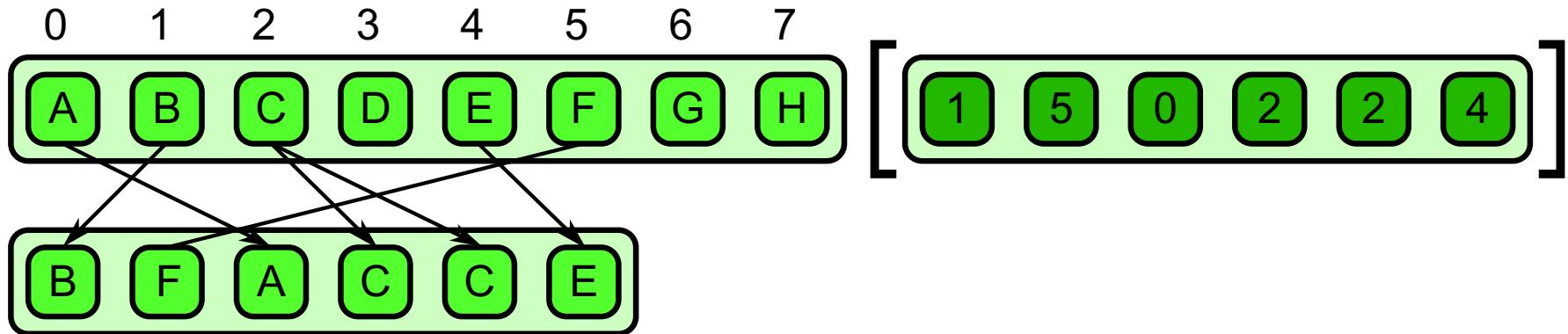
Parallelize over
for loop to
perform random
read

Serial implementation of gather in pseudocode

Are there any conflicts that arise?

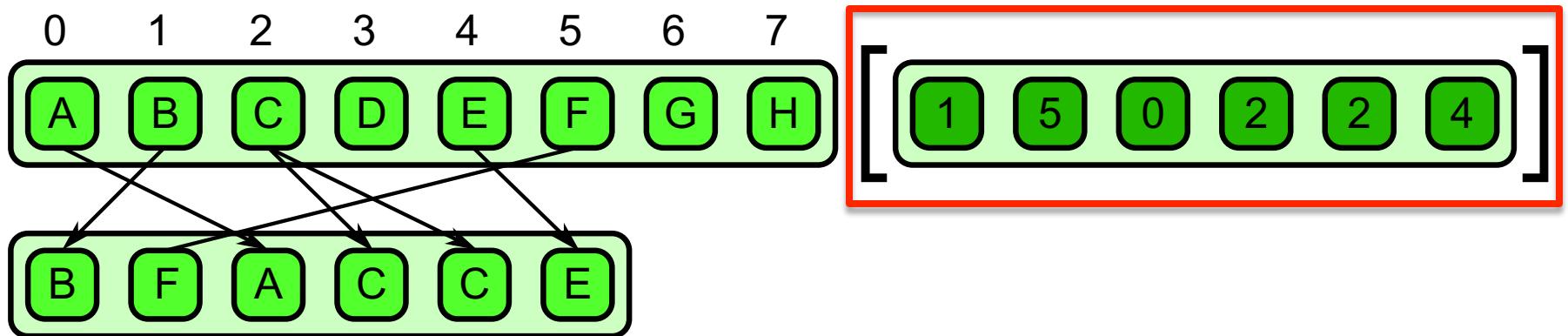
Gather: Defined (parallel perspective)

- Results from the combination of a map with a random read



- Simple pattern, but with many special cases that make the implementation more efficient

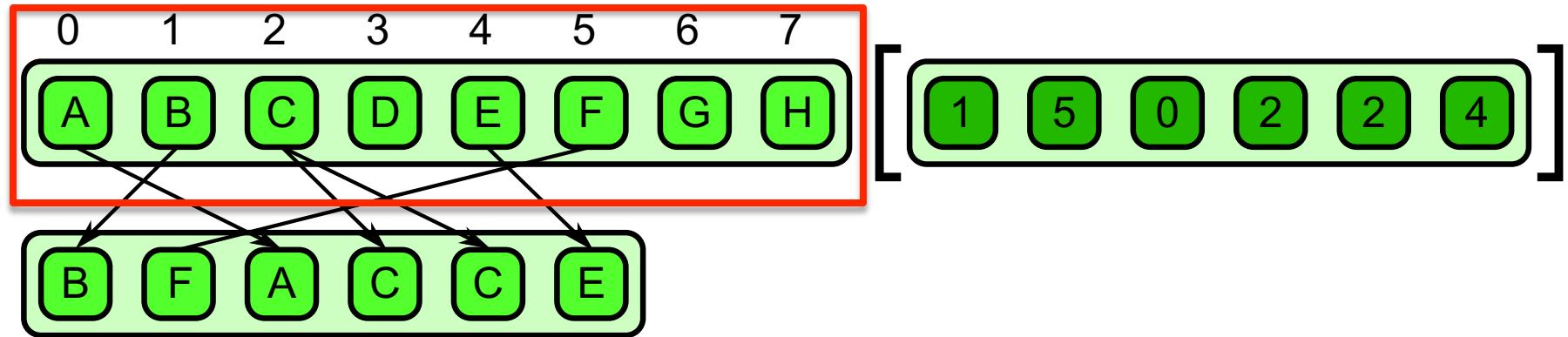
Gather: Defined



Given a **collection of read locations**

- ❑ address or array indices

Gather: Defined

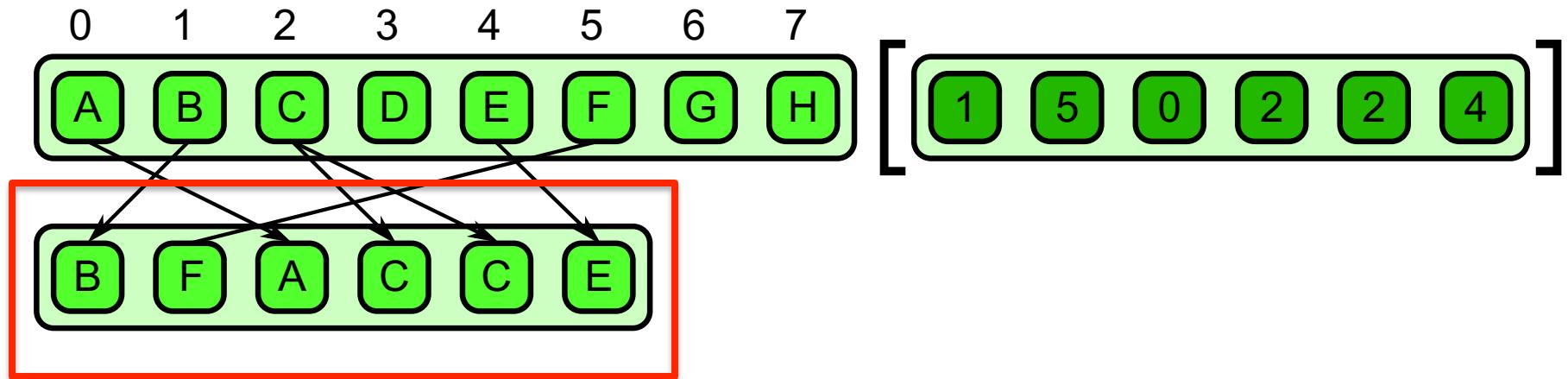


Given a collection of read locations

- ❑ address or array indices

and a **source array**

Gather: Defined



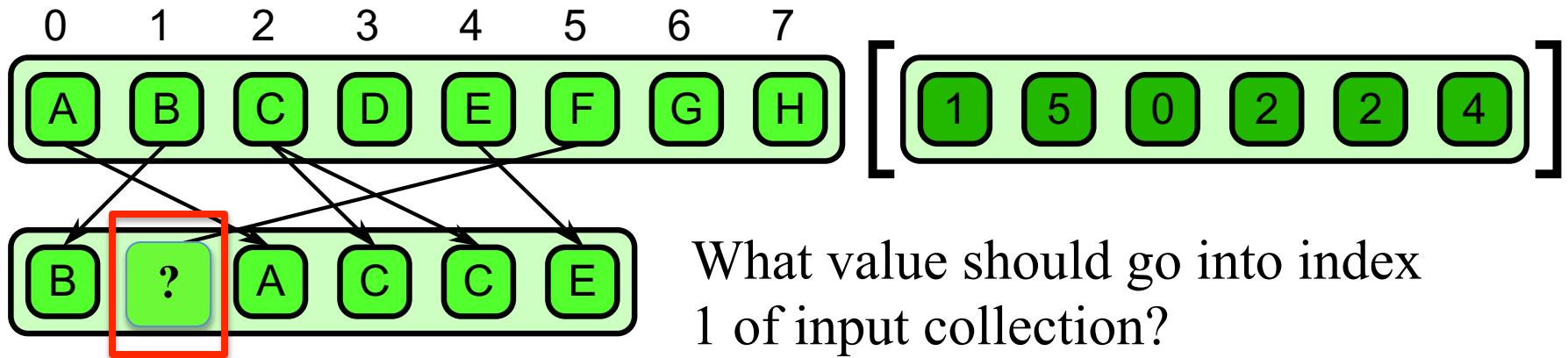
Given a collection of read locations

❑ address or array indices

and a source array

gather all the data from the source array at the given locations and places them into an **output collection**

Gather: Defined



What value should go into index 1 of input collection?

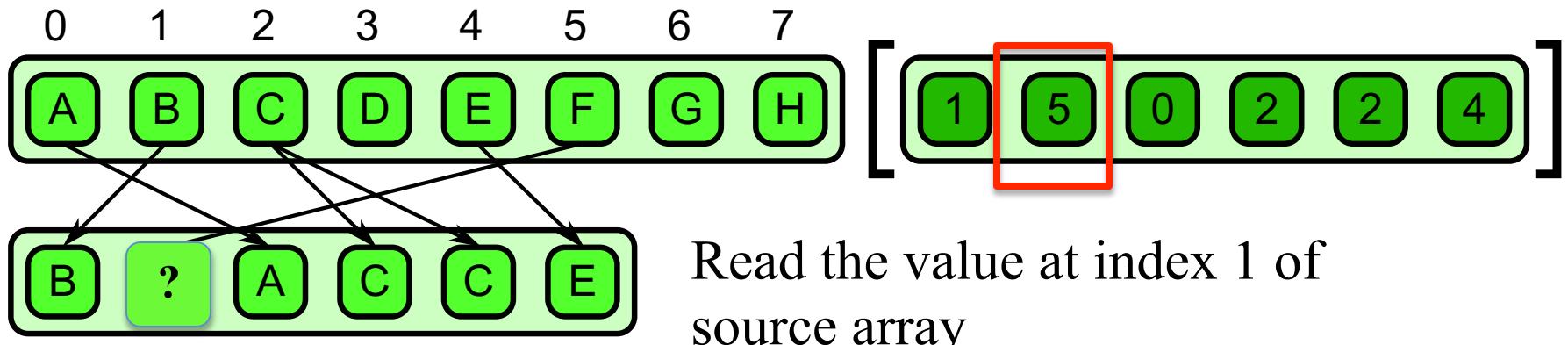
Given a collection of read locations

- ❑ address or array indices

and a source array

gather all the data from the source array at the given locations and places them into an **output collection**

Gather: Defined



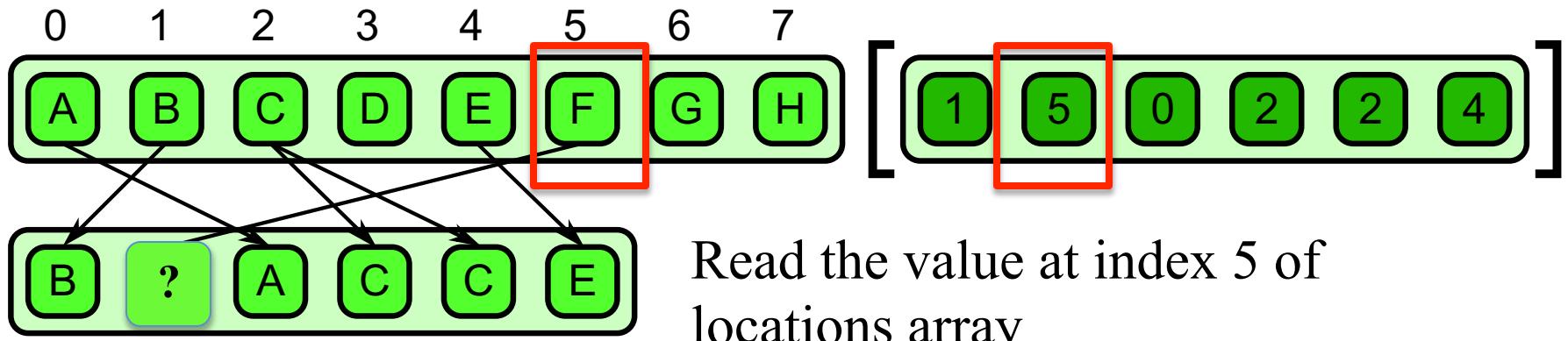
Given a collection of read locations

- ❑ address or array indices

and a source array

gather all the data from the source array at the given locations and places them into an **output collection**

Gather: Defined



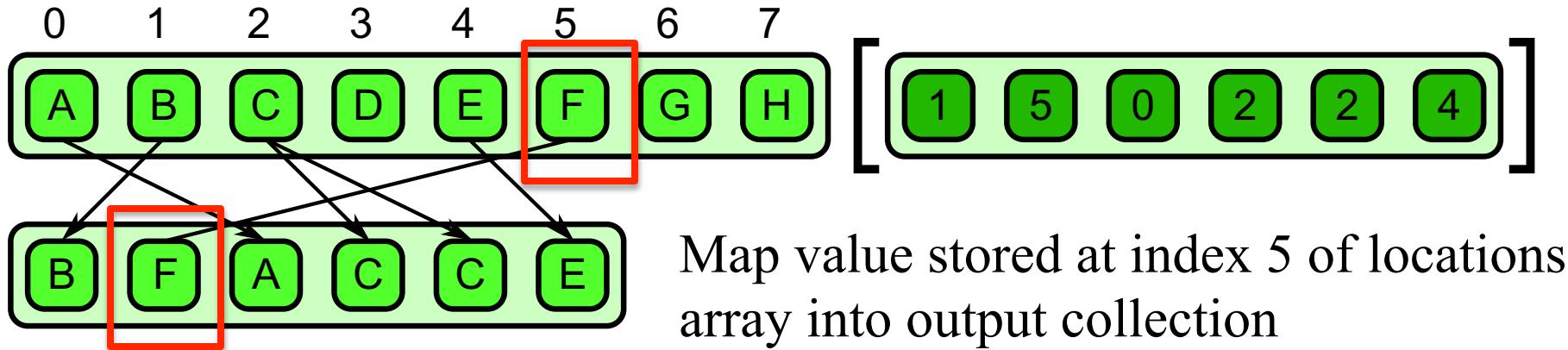
Given a collection of read locations

❑ address or array indices

and a source array

gather all the data from the source array at the given locations and places them into an **output collection**

Gather: Defined



Map value stored at index 5 of locations array into output collection

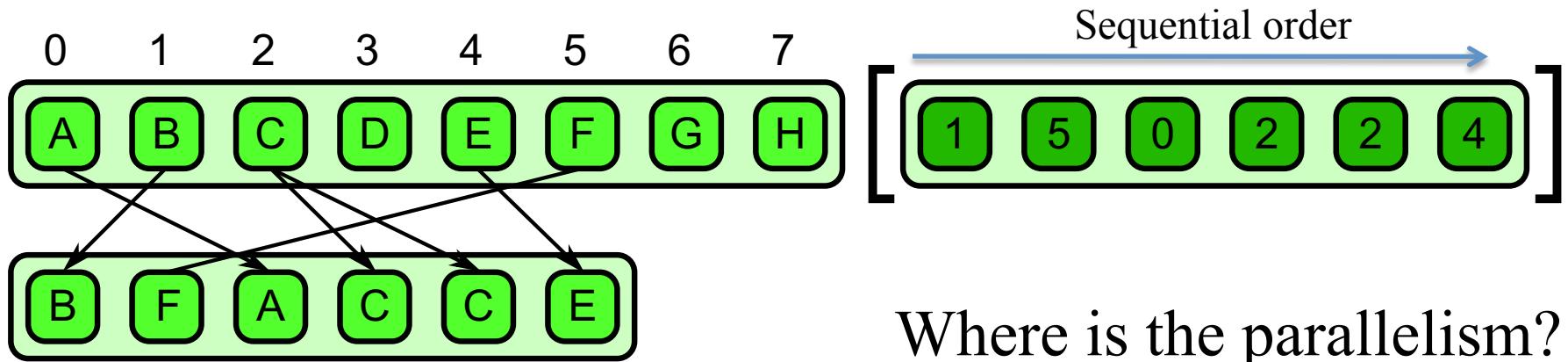
Given a collection of read locations

- ❑ address or array indices

and a source array

gather all the data from the source array at the given locations and places them into an **output collection**

Gather: Defined



Where is the parallelism?

Given a collection of read locations

- ❑ address or array indices

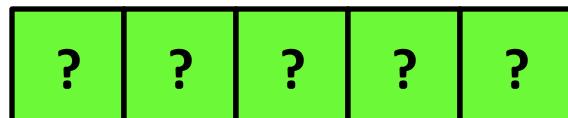
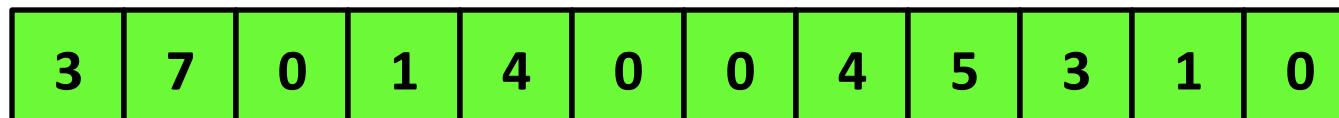
and a source array

gather all the data from the source array at the given locations and places them into an **output collection**

Quiz 1

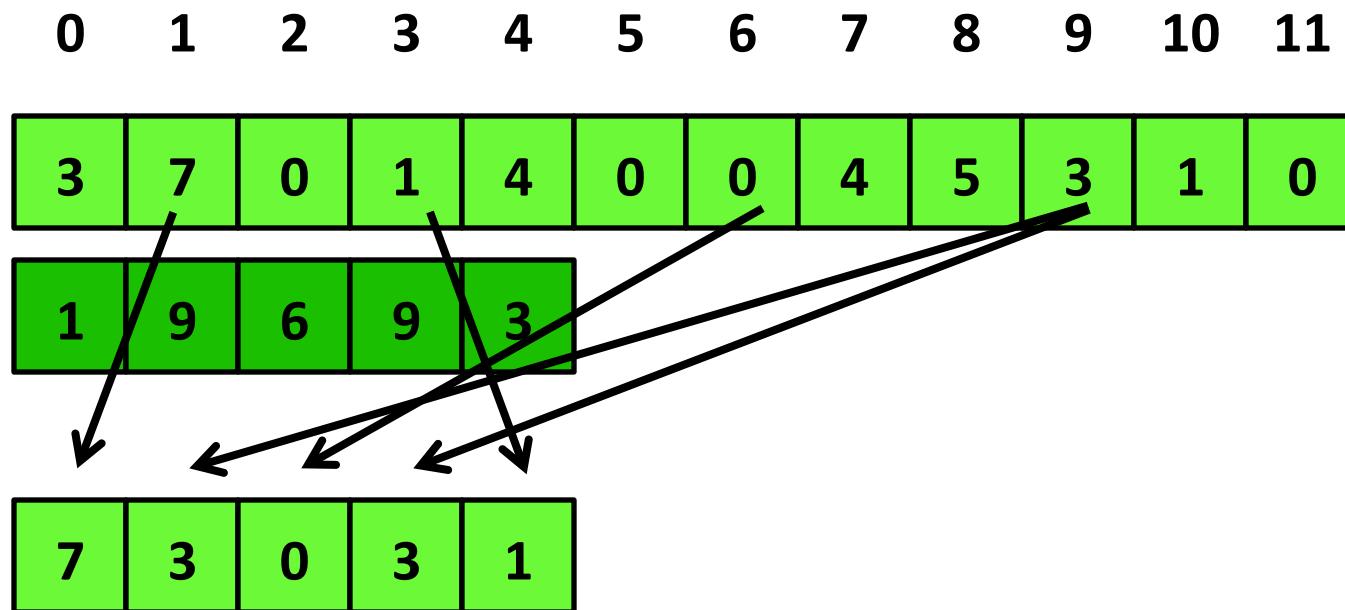
Given the following locations and source array, use a gather to determine what values should go into the output collection:

0 1 2 3 4 5 6 7 8 9 10 11

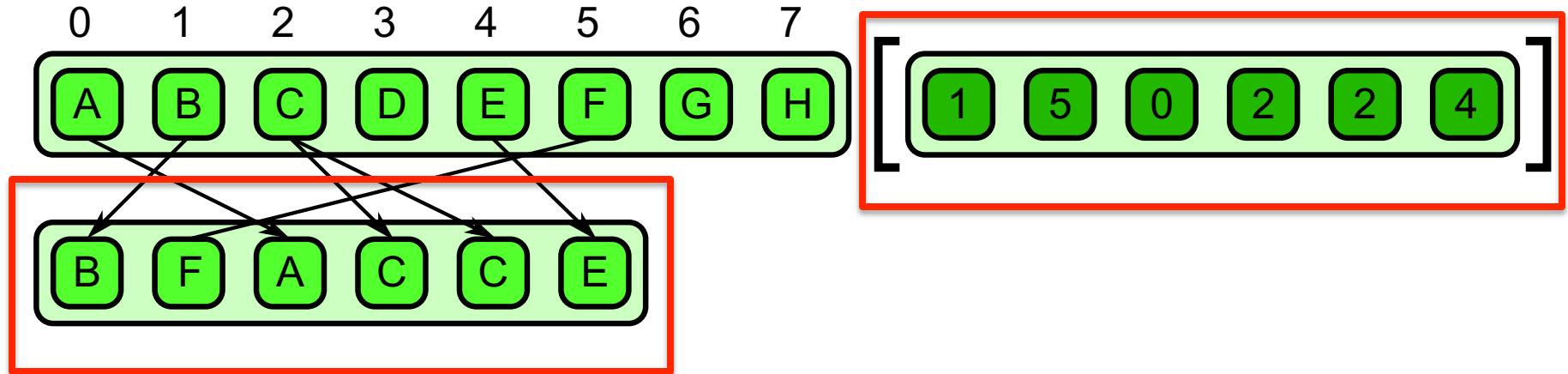


Quiz 1

Given the following locations and source array, use a gather to determine what values should go into the output collection:

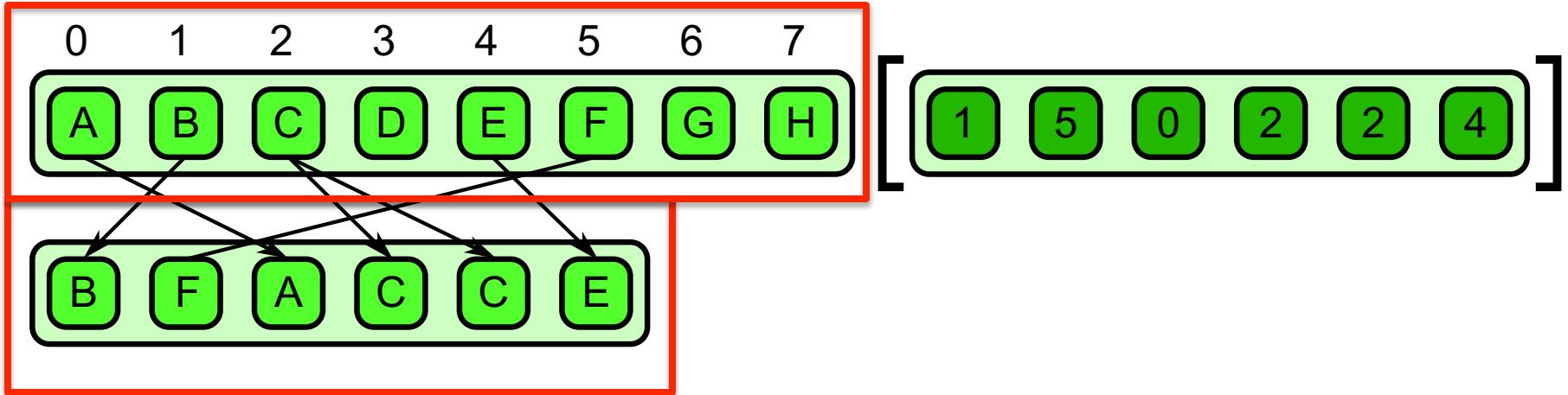


Gather: Array Size



- Output data collection has the same number of elements as the number of indices in the index collection
 - Same dimensionality

Gather: Array Size

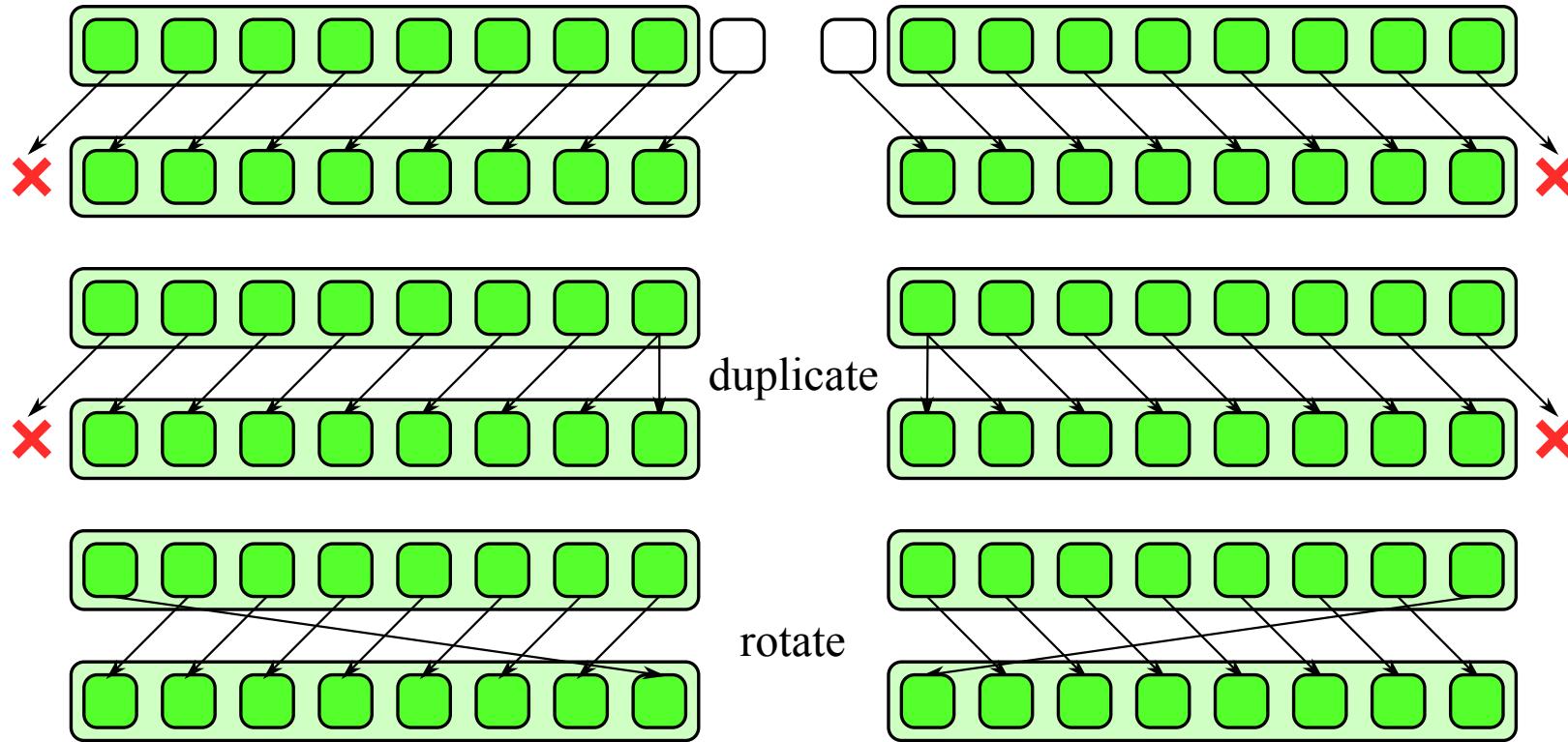


- Output data collection has the same number of elements as the number of indices in the index collection
- Elements of the output collection are the same type as the input data collection

Outline

- Gather Pattern
 - Shifts, Zip, Unzip
- Scatter Pattern
 - Collision Rules: atomic, permutation, merge, priority
- Pack Pattern
 - Split, Unsplit, Bin
 - Fusing Map and Pack
 - Expand
- Partitioning Data
- AoS vs. SoA
- Example Implementation: AoS vs. SoA

Special Case of Gather: Shifts



- Moves data to the left or right in memory
- Data accesses are offset by fixed distances

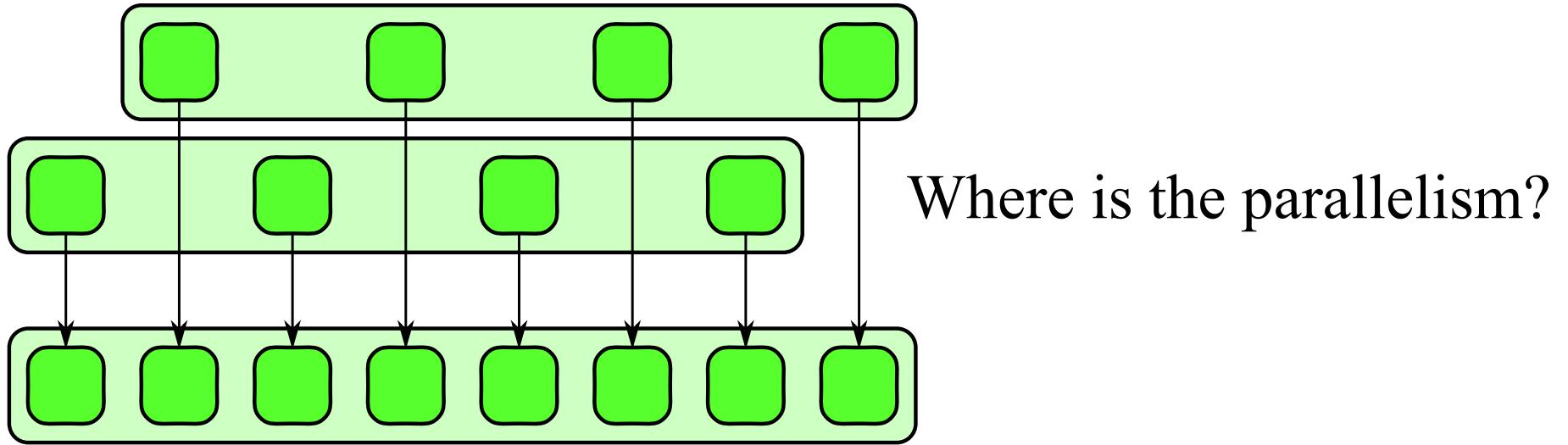
More about Shifts

- Regular data movement
- Variants from how boundary conditions handled
 - Requires “out of bounds” data at edge of the array
 - Options: default value, duplicate, rotate
- Shifts can be handled efficiently with vector instructions because of regularity
 - Shift multiple data elements at the same time
- Shifts can also take advantage of good data locality

Table of Contents

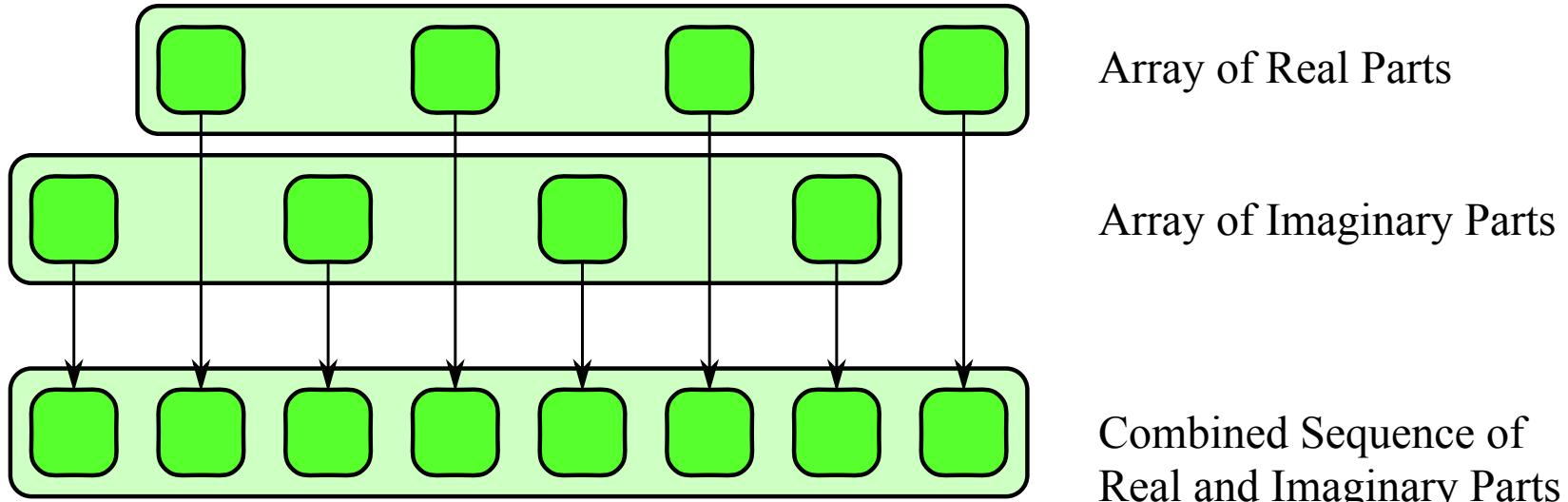
- Gather Pattern
 - Shifts, **Zip**, Unzip
- Scatter Pattern
 - Collision Rules: atomic, permutation, merge, priority
- Pack Pattern
 - Split, Unsplit, Bin
 - Fusing Map and Pack
 - Expand
- Partitioning Data
- AoS vs. SoA
- Example Implementation: AoS vs. SoA

Special Case of Gather: Zip



- Function is to interleaves data (like a zipper)

Zip Example



- Given two separate arrays of real parts and imaginary parts
- Use `zip` to combine them into a sequence of real and imaginary pairs

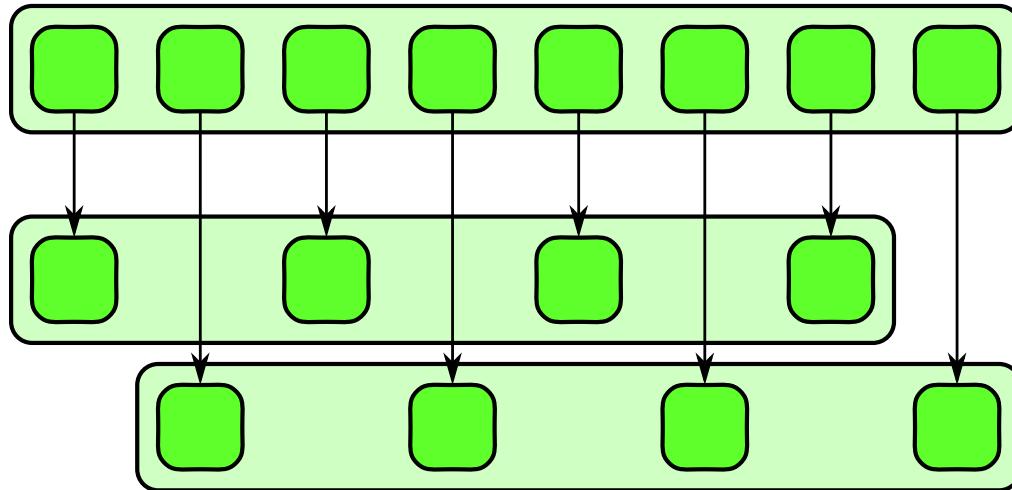
More about Zip

- Can be generalized to more elements
- Can zip data of unlike types

Table of Contents

- Gather Pattern
 - Shifts, Zip, **Unzip**
- Scatter Pattern
 - Collision Rules: atomic, permutation, merge, priority
- Pack Pattern
 - Split, Unsplit, Bin
 - Fusing Map and Pack
 - Expand
- Partitioning Data
- AoS vs. SoA
- Example Implementation: AoS vs. SoA

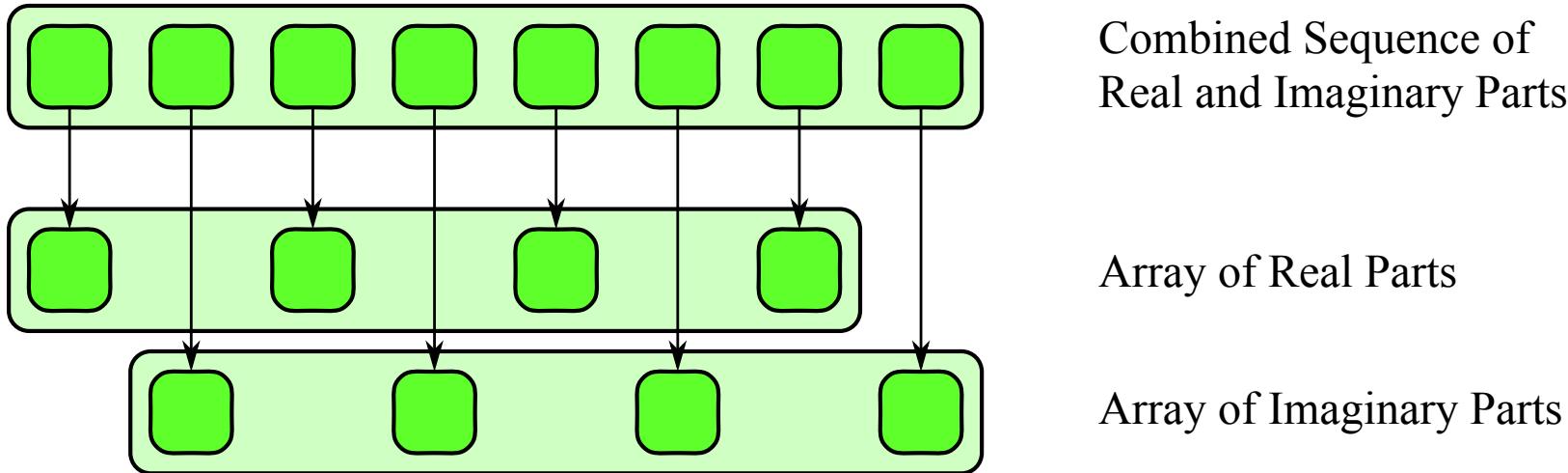
Special Case of Gather: Unzip



Where is the parallelism?

- Reverses a zip
- Extracts sub-arrays at certain offsets and strides from an input array

Unzip Example



- Given a sequence of complex numbers organized as pairs
- Use `unzip` to extract real and imaginary parts into separate arrays

Contents

- Gather Pattern
 - Shifts, Zip, Unzip
- Scatter Pattern
 - Collision Rules: atomic, permutation, merge, priority
- Pack Pattern
 - Split, Unsplit, Bin
 - Fusing Map and Pack
 - Expand
- Partitioning Data
- AoS vs. SoA
- Example Implementation: AoS vs. SoA

Gather vs. Scatter

Gather

- Combination of map with random **reads**
- Read locations provided as input

Scatter

- Combination of map with random **writes**
- Write locations provided as input
- Race conditions ... Why?

Scatter: Serial Implementation

```
1 template<typename Data, typename Idx>
2 void scatter(
3     size_t n, //number of elements in output data collection
4     size_t m, //number of elements in input data and index collection
5     Data a[], //input data collection (m elements)
6     Data A[], //output data collection (n elements)
7     Idx idx[] //input index collection (m elements)
8 ) {
9     for (size_t i = 0; i < m; ++i) {
10         size_t j = idx[i]; //get ith index
11         assert(0 <= j && j < n); //check output array bounds
12         A[j] = a[i]; //perform random write
13     }
14 }
```

Serial implementation of scatter in pseudocode

Scatter: Serial Implementation

```
1 template<typename Data, typename Idx>
2 void scatter(
3     size_t n, //number of elements in output data collection
4     size_t m, //number of elements in input data and index collection
5     Data a[], //input data collection (m elements)
6     Data A[], //output data collection (n elements)
7     Idx idx[] //input index collection (m elements)
8 ) {
9     for (size_t i = 0; i < m; ++i) {
10         size_t j = idx[i]; //get ith index
11         assert(0 <= j && j < n); //check output array bounds
12         A[j] = a[i]; //perform random write
13     }
14 }
```

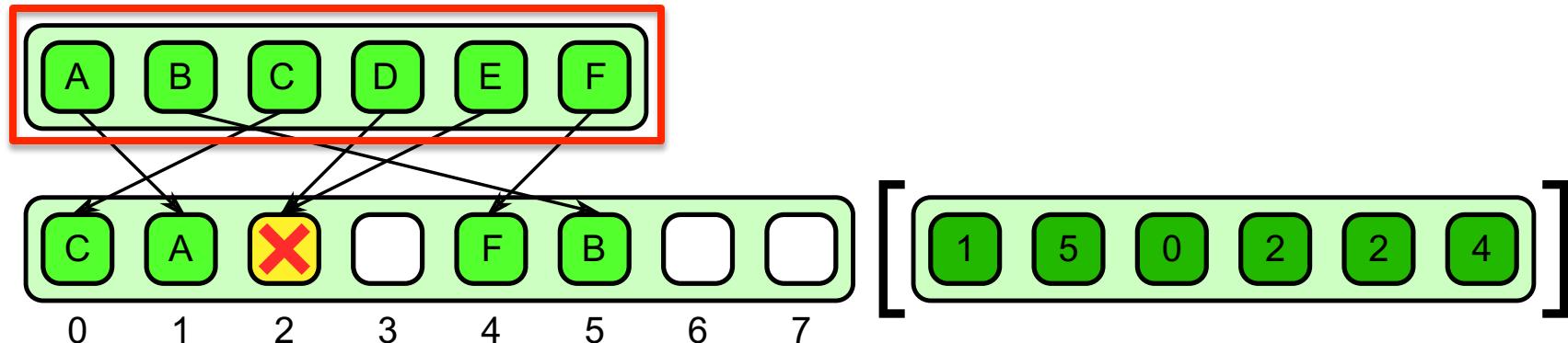
Parallelize over
for loop to
perform random
write

Serial implementation of scatter in pseudocode

Scatter: Defined

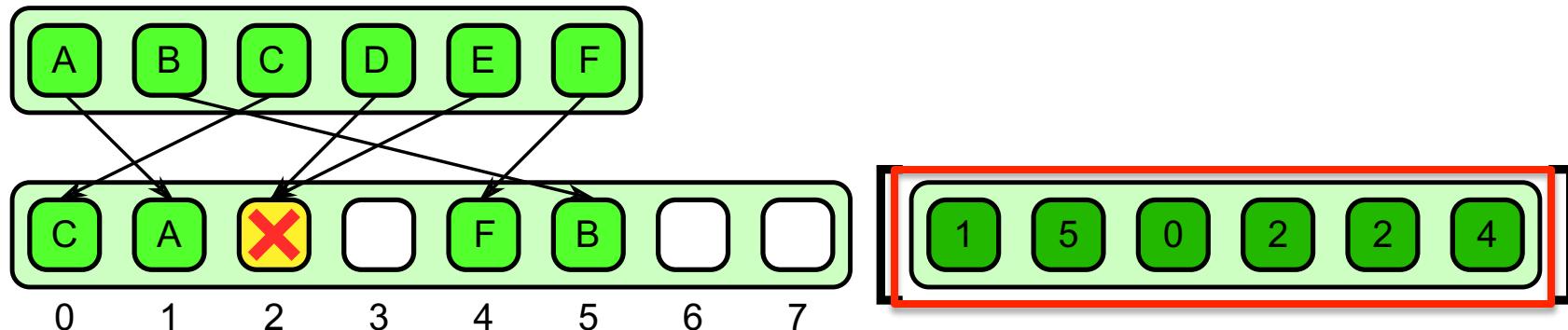
- Results from the combination of a map with a random write
- Writes to the same location are possible
- Parallel writes to the same location are **collisions**

Scatter: Defined



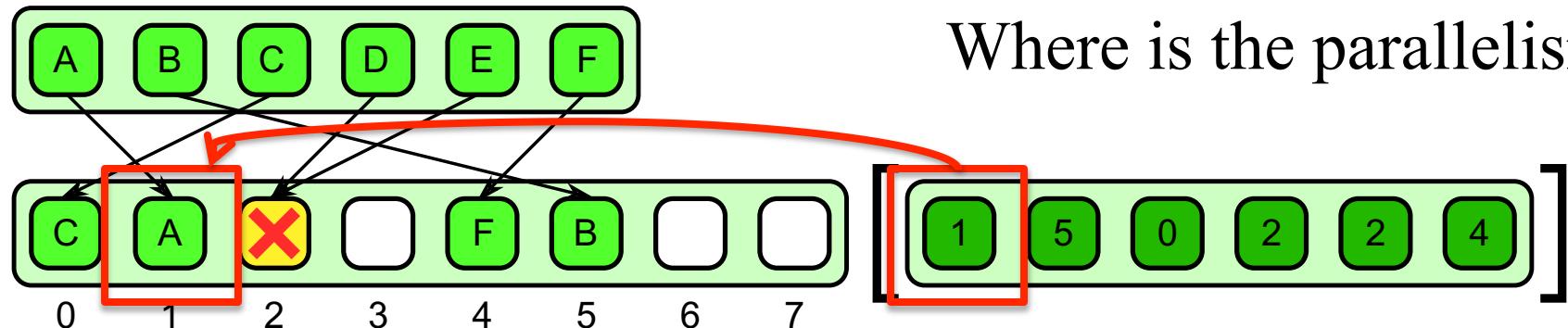
Given a collection of **input data**

Scatter: Defined



Given a collection of input data
and a collection of **write locations**

Scatter: Defined



Where is the parallelism?

Given a collection of input data
and a collection of write locations
scatter data to the **output collection**

Problems?

Does the output collection have to be larger in size?

Quiz 2

Given the following locations and source array, what values should go into the input collection:

0 1 2 3 4 5 6 7 8 9 10 11

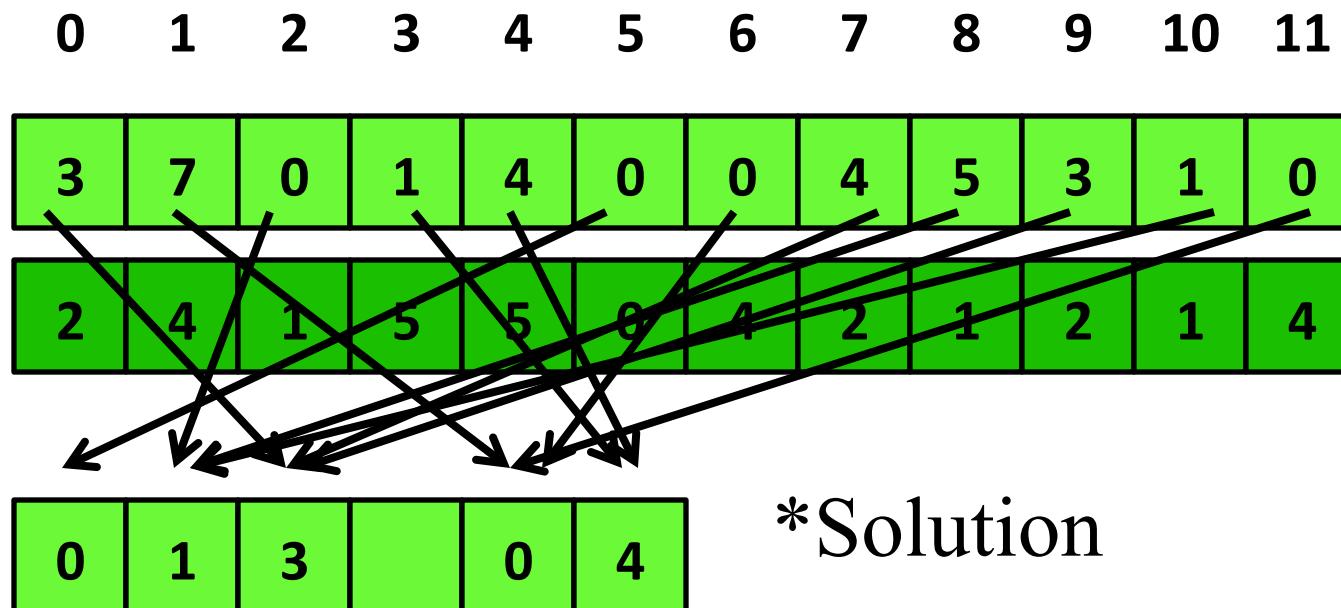
3	7	0	1	4	0	0	4	5	3	1	0
---	---	---	---	---	---	---	---	---	---	---	---

2	4	1	5	5	0	4	2	1	2	1	4
---	---	---	---	---	---	---	---	---	---	---	---

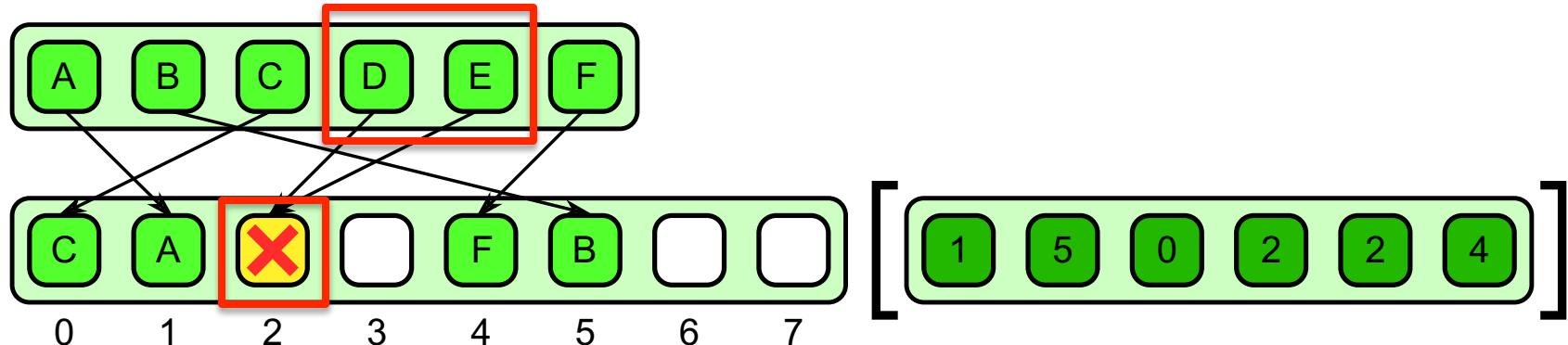
?	?	?	?	?	?
---	---	---	---	---	---

Quiz 2

Given the following locations and source array, what values should go into the input collection:



Scatter: Race Conditions



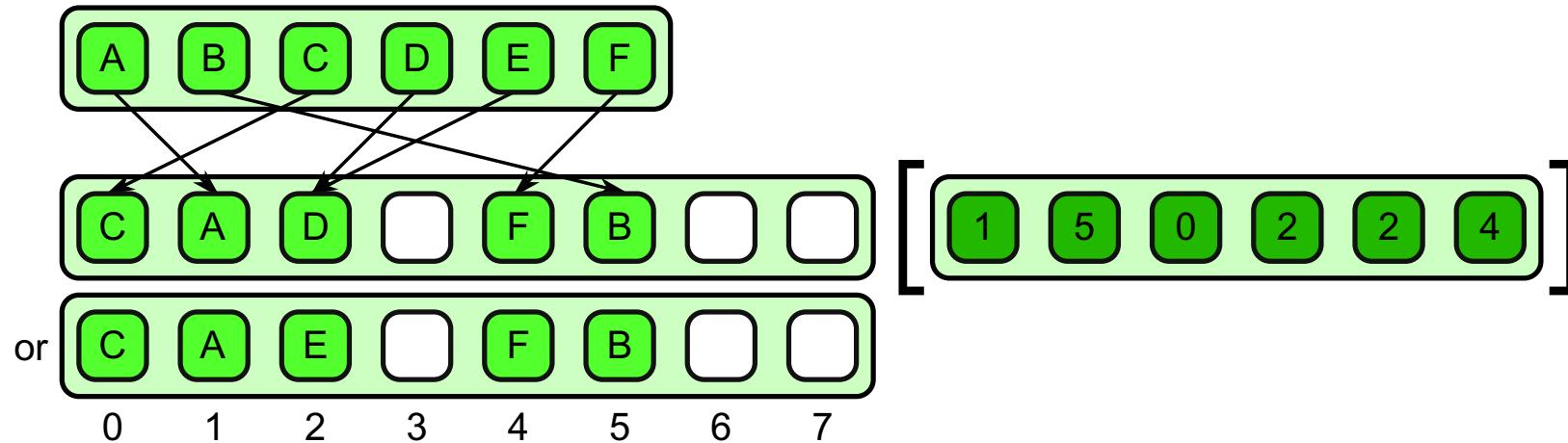
Given a collection of input data
and a collection of write locations
scatter data to the output collection

Race Condition: Two (or more) values being written to the same location in output collection. Result is undefined unless enforce rules. **Need rules to resolve collisions!**

Table of Contents

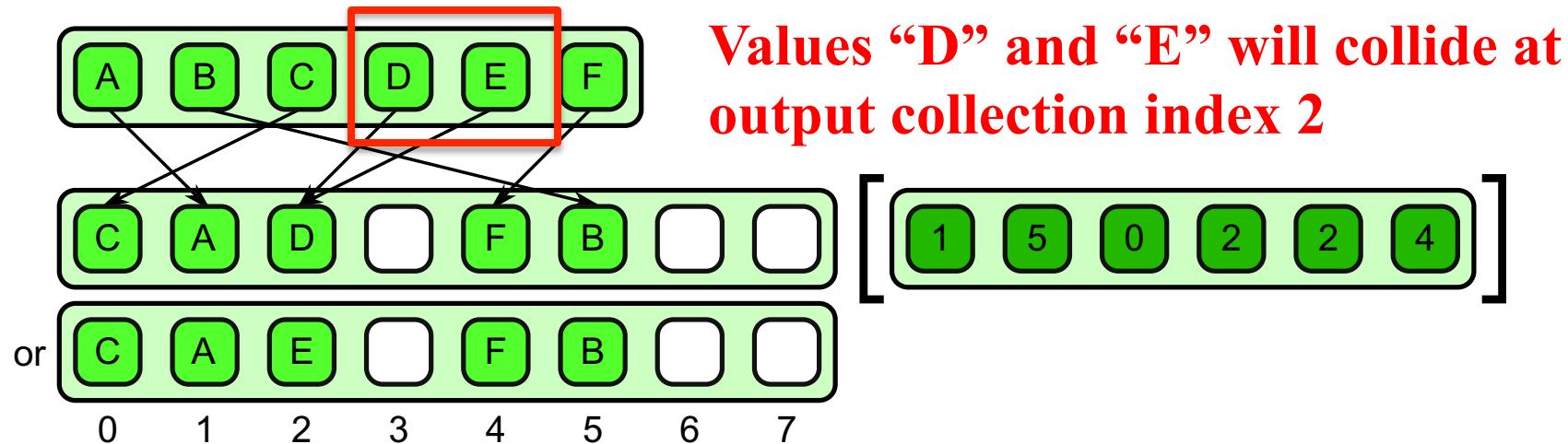
- Gather Pattern
 - Shifts, Zip, Unzip
- Scatter Pattern
 - Collision Rules: atomic, permutation, merge, priority
- Pack Pattern
 - Split, Unsplit, Bin
 - Fusing Map and Pack
 - Expand
- Partitioning Data
- AoS vs. SoA
- Example Implementation: AoS vs. SoA

Collision Resolution: Atomic Scatter



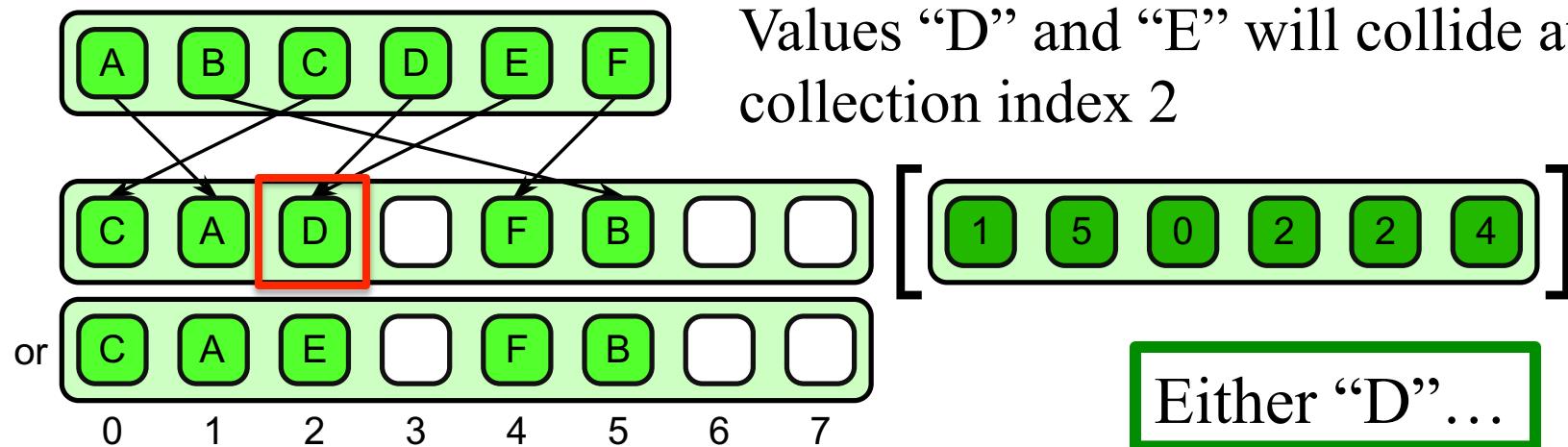
- **Non-deterministic** approach
- Upon collision, one and only one of the values written to a location will be written in its entirety

Collision Resolution: Atomic Scatter



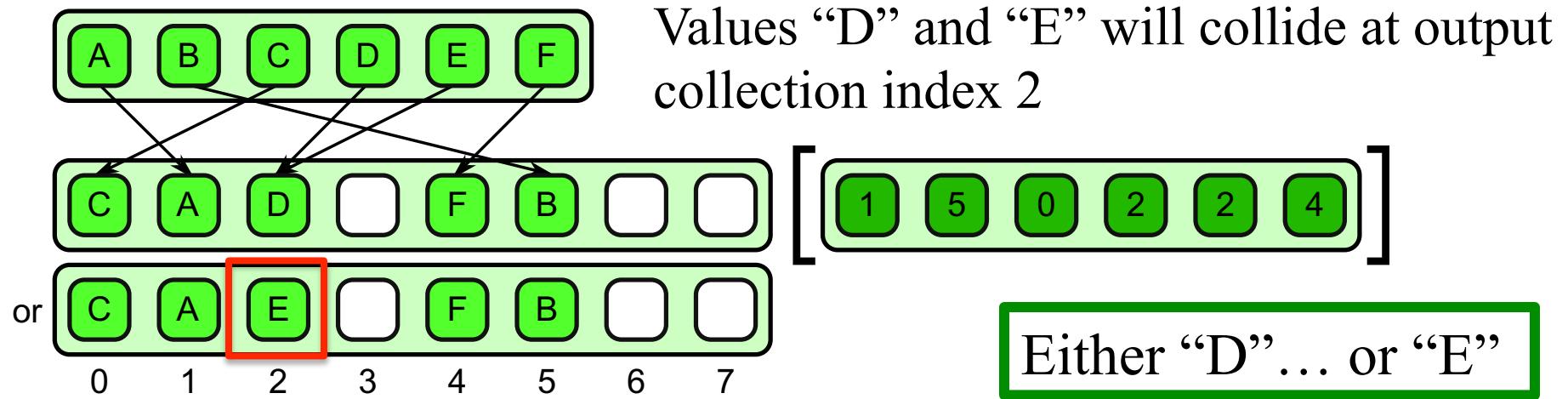
- Non-deterministic approach
- Upon collision, one and only one of the values written to a location will be written in its entirety

Collision Resolution: Atomic Scatter



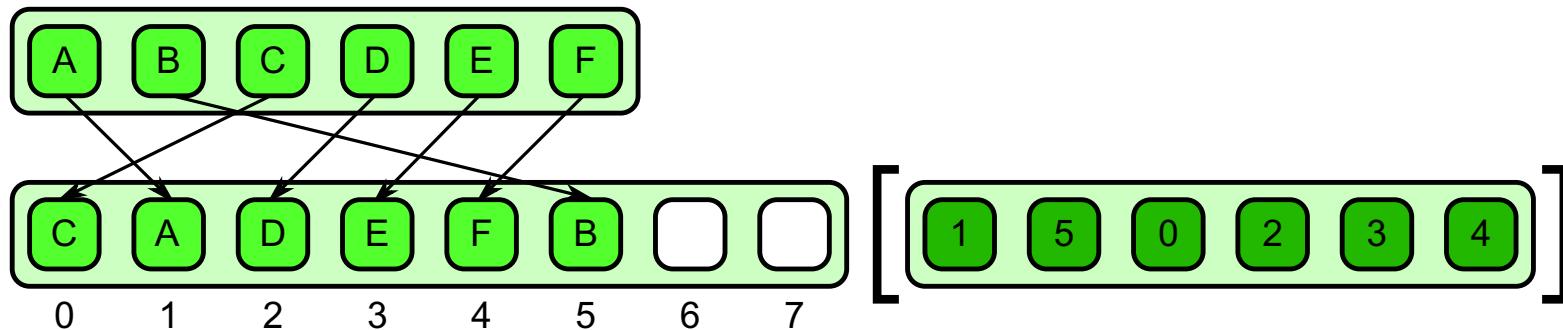
- Non-deterministic approach
- Upon collision, one and only one of the values written to a location will be written in its entirety
- No rule determines which of the input items will be retained

Collision Resolution: Atomic Scatter



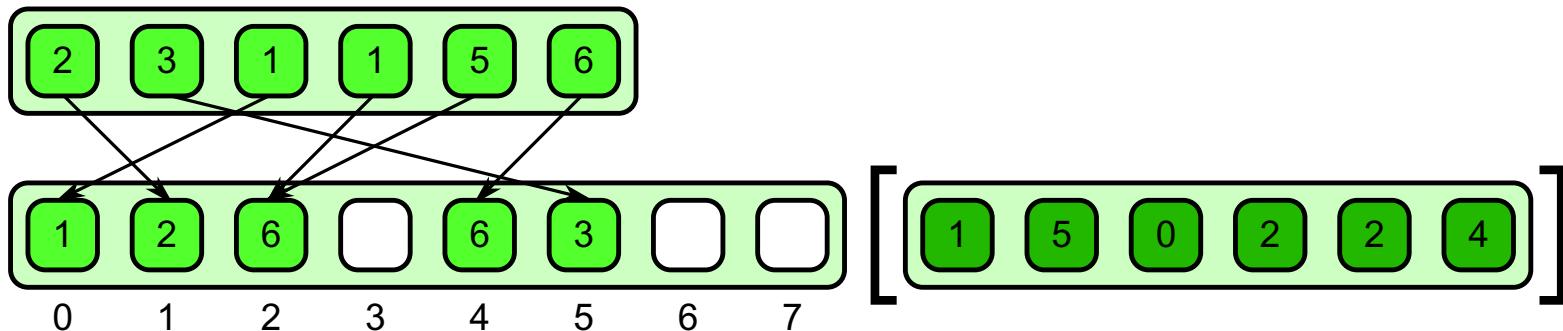
- Non-deterministic approach
- Upon collision, one and only one of the values written to a location will be written in its entirety
- No rule determines which of the input items will be retained

Collision Resolution: Permutation Scatter



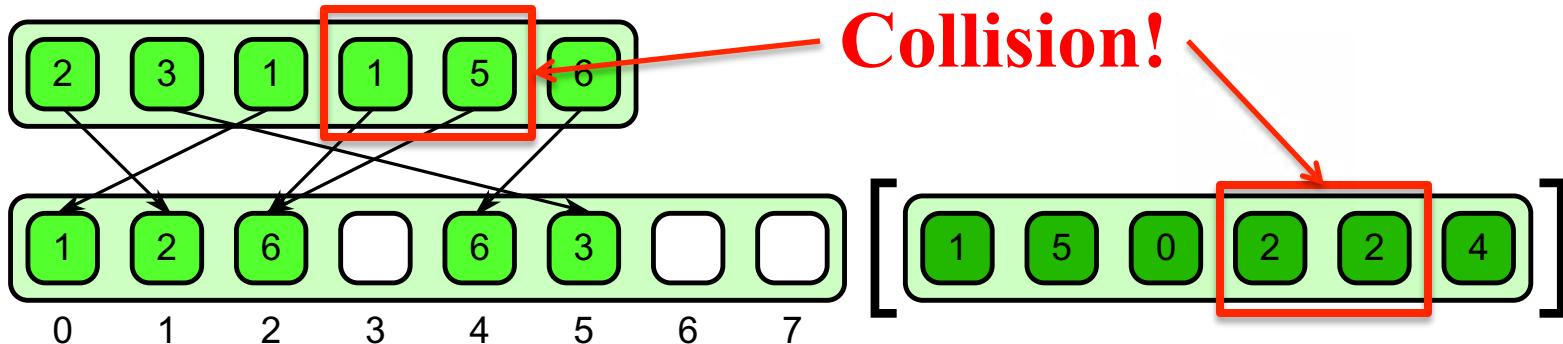
- Pattern simply states that collisions are **illegal**
 - Output is a permutation of the input
- Check for collisions in advance
 - turn scatter into gather
- Examples
 - FFT scrambling, matrix/image transpose, unpacking

Collision Resolution: Merge Scatter



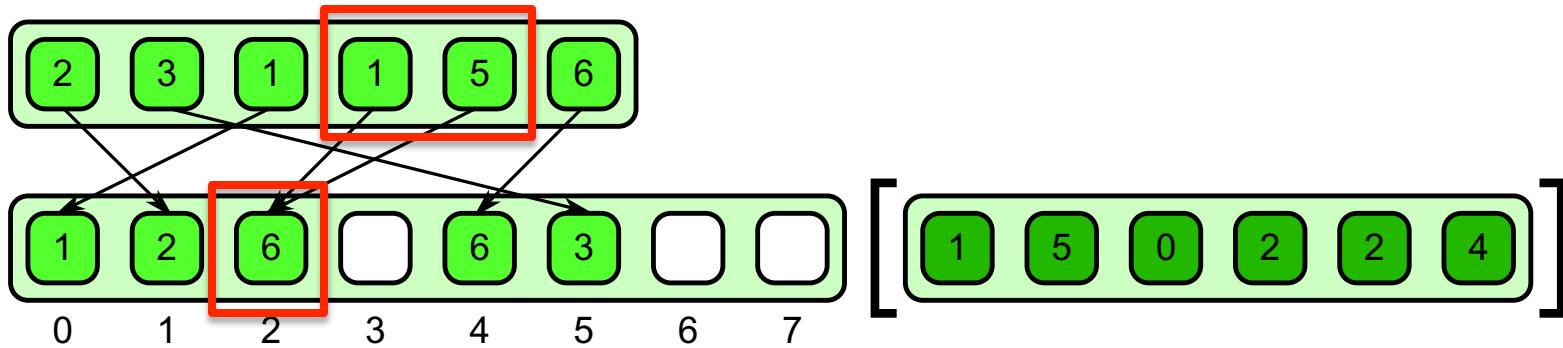
- Associative and commutative operators are provided to merge elements in case of a collision

Collision Resolution: Merge Scatter



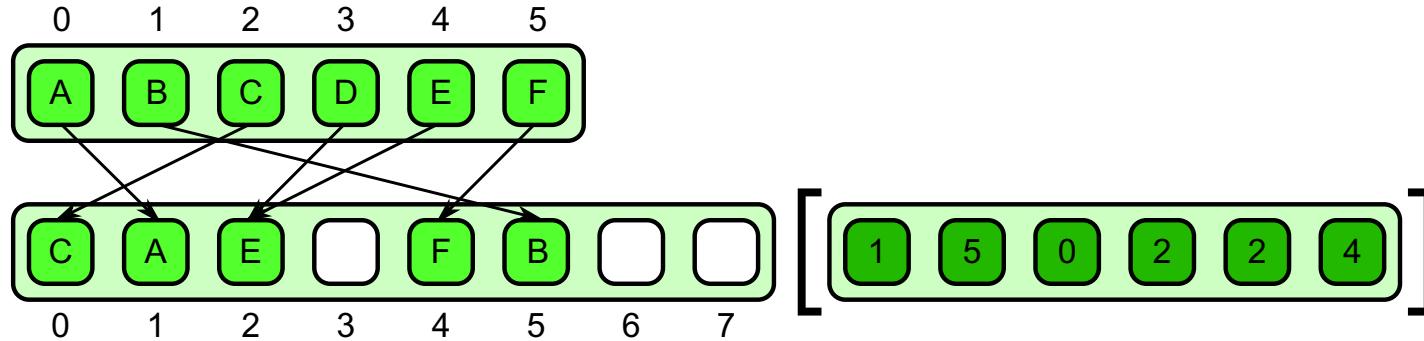
- ❑ Associative and commutative operators are provided to merge elements in case of a collision

Collision Resolution: Merge Scatter



- Associative and commutative operators are provided to merge elements in case of a collision
- Use addition as the merge operator
- Both associative and commutative properties are required since scatters to a particular location could occur in any order

Collision Resolution: Priority Scatter



- Every element in the input array is assigned a priority based on its position
- Priority is used to decide which element is written in case of a collision
- Example
 - 3D graphics rendering

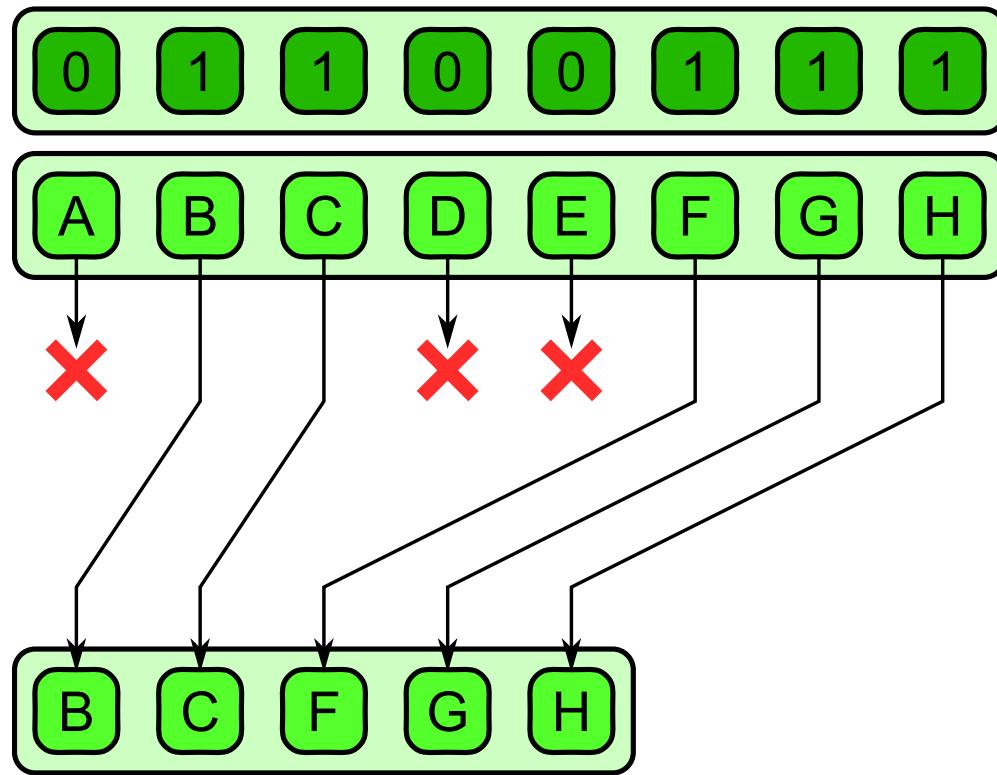
Converting Scatter to Gather

- Scatter is a more expensive than gather
 - Writing has cache line consequences
 - May cause additional reading due to cache conflicts
 - **False sharing** is a problem that arises
 - ◆ writes from different cores go to the same cache line
- Can avoid problems if addresses are known “in advance”
 - Allows optimizations to be applied
 - Convert addresses for a scatter into those for a gather
 - Useful if the same pattern of scatter address will be used repeatedly so the cost is amortized

Table of Contents

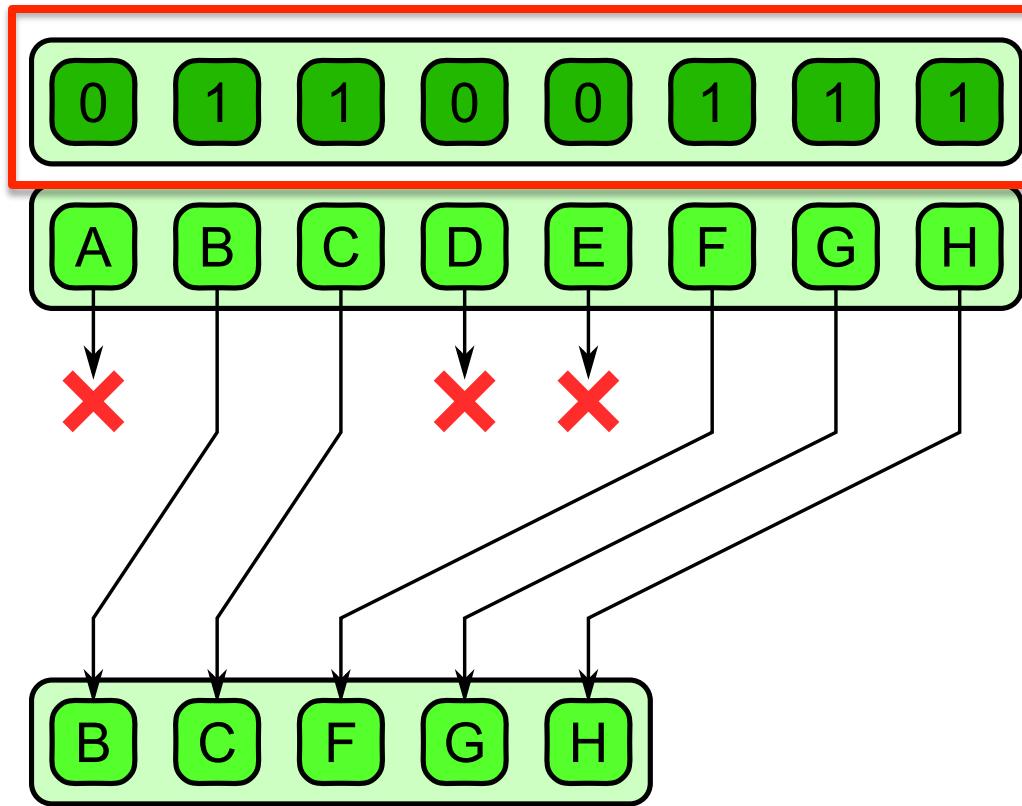
- Gather Pattern
 - Shifts, Zip, Unzip
- Scatter Pattern
 - Collision Rules: atomic, permutation, merge, priority
- Pack Pattern
 - Split, Unsplit, Bin
 - Fusing Map and Pack
 - Expand
- Partitioning Data
- AoS vs. SoA
- Example Implementation: AoS vs. SoA

Pack: Defined



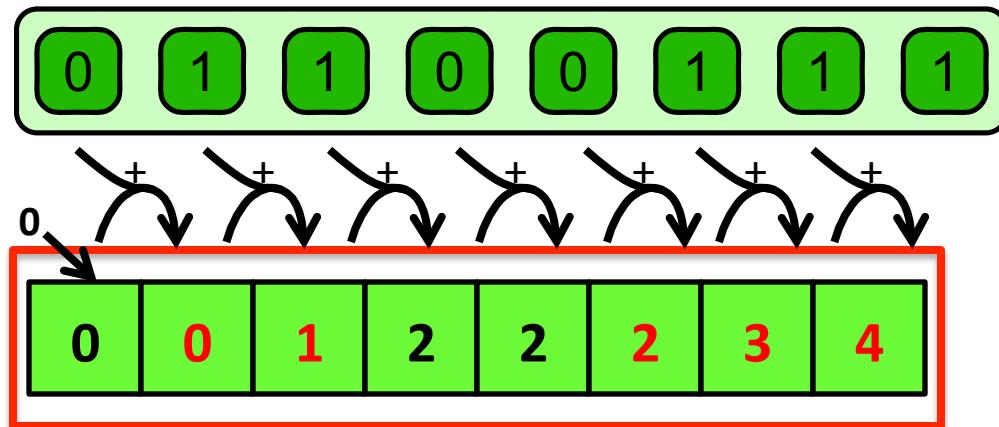
- Used to eliminate unused elements from a collection
- Retained elements are moved so they are contiguous in memory
- Goal is to improve the performance ...
How?

Pack Algorithm



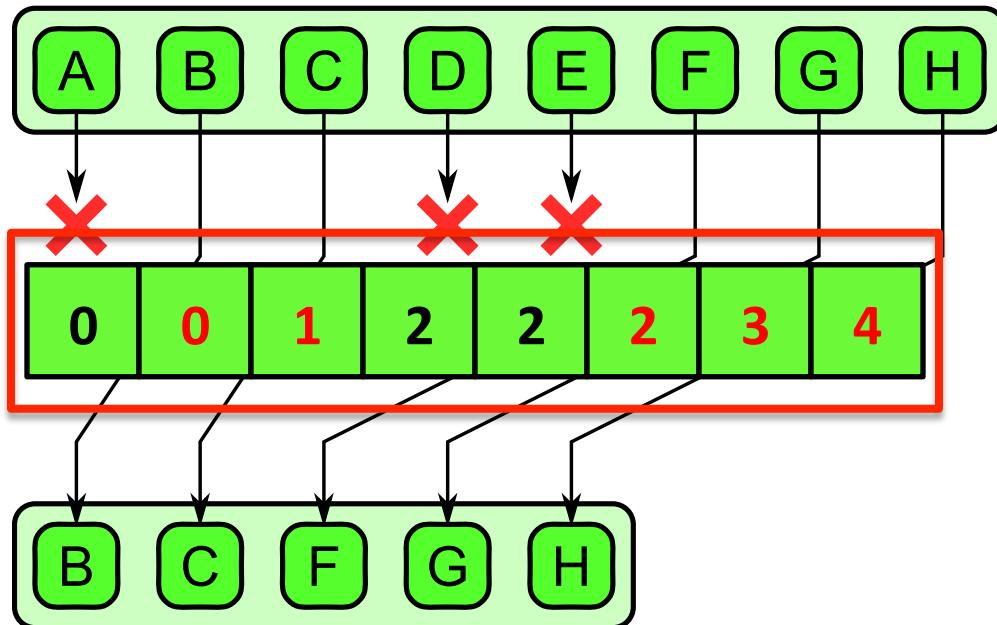
1. Convert input array of Booleans into integer 0's and 1's

Pack Algorithm



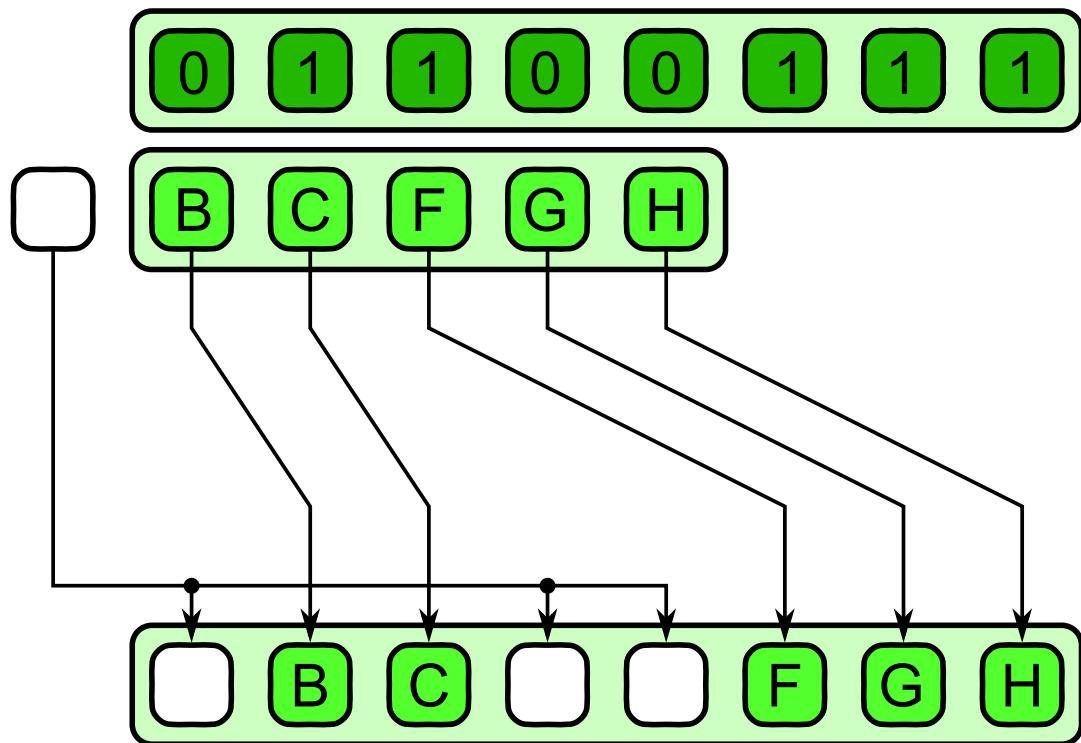
1. Convert input array of Booleans into integer 0's and 1's
2. Exclusive scan of this array with the addition operation

Pack Algorithm



1. Convert input array of Booleans into integer 0's and 1's
2. Exclusive scan of this array with the addition operation
3. **Write values to output array based on offsets**

Unpack: Defined

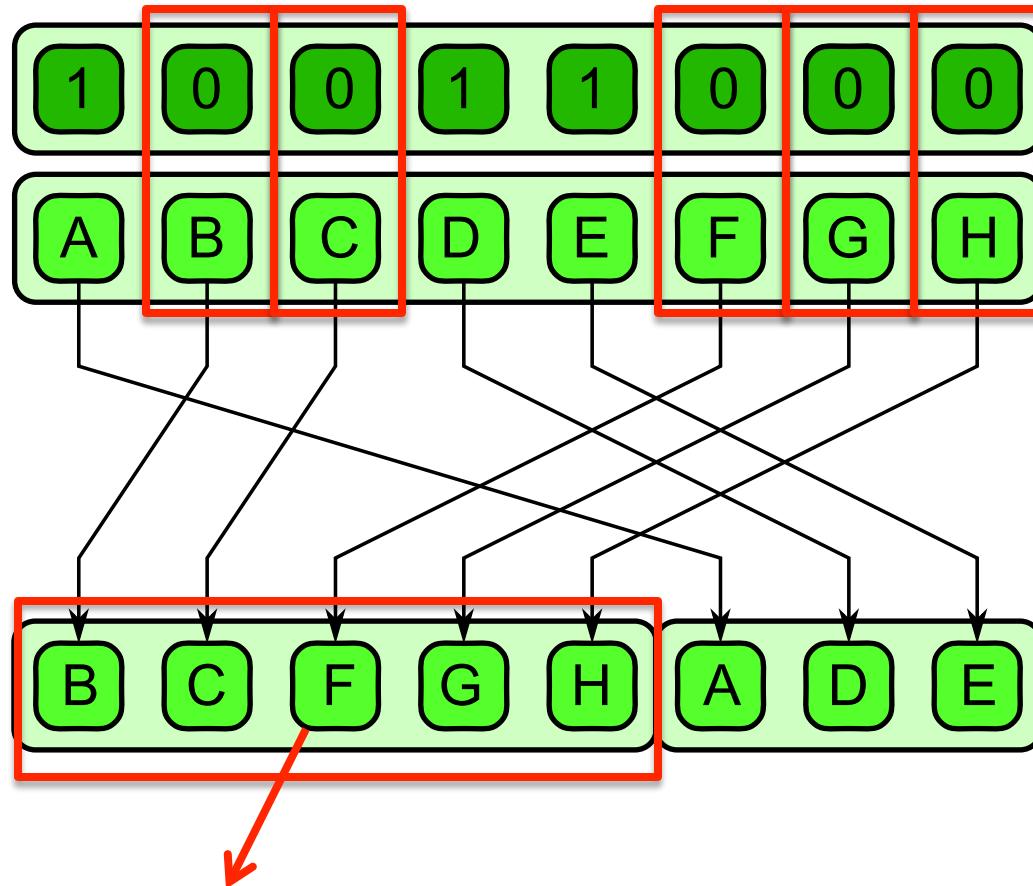


- Inverse of pack operation
- Given the same data on which elements were kept and which were discarded, spread elements back in their original locations

Table of Contents

- Gather Pattern
 - Shifts, Zip, Unzip
- Scatter Pattern
 - Collision Rules: atomic, permutation, merge, priority
- Pack Pattern
 - Split, Unsplit, Bin
 - Fusing Map and Pack
 - Expand
- Partitioning Data
- AoS vs. SoA
- Example Implementation: AoS vs. SoA

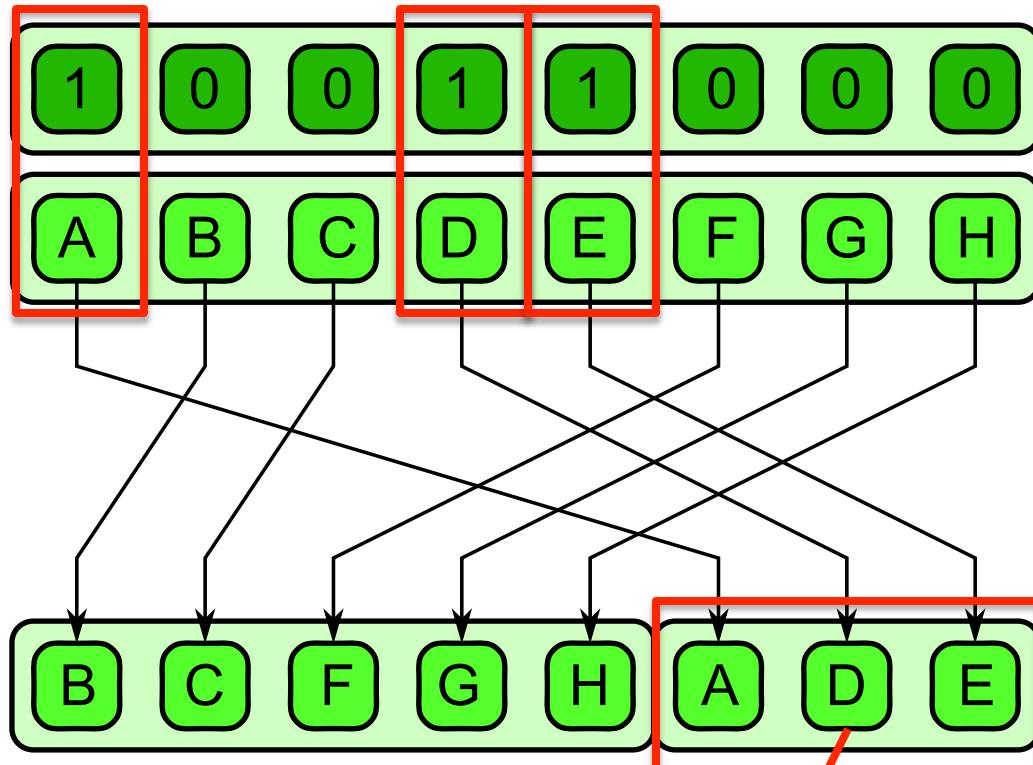
Generalization of Pack: Split



Upper half of output collection: values equal to 0

- Generalization of pack pattern
- Elements are moved to upper or lower half of output collection based on some state
- Does not lose information like pack

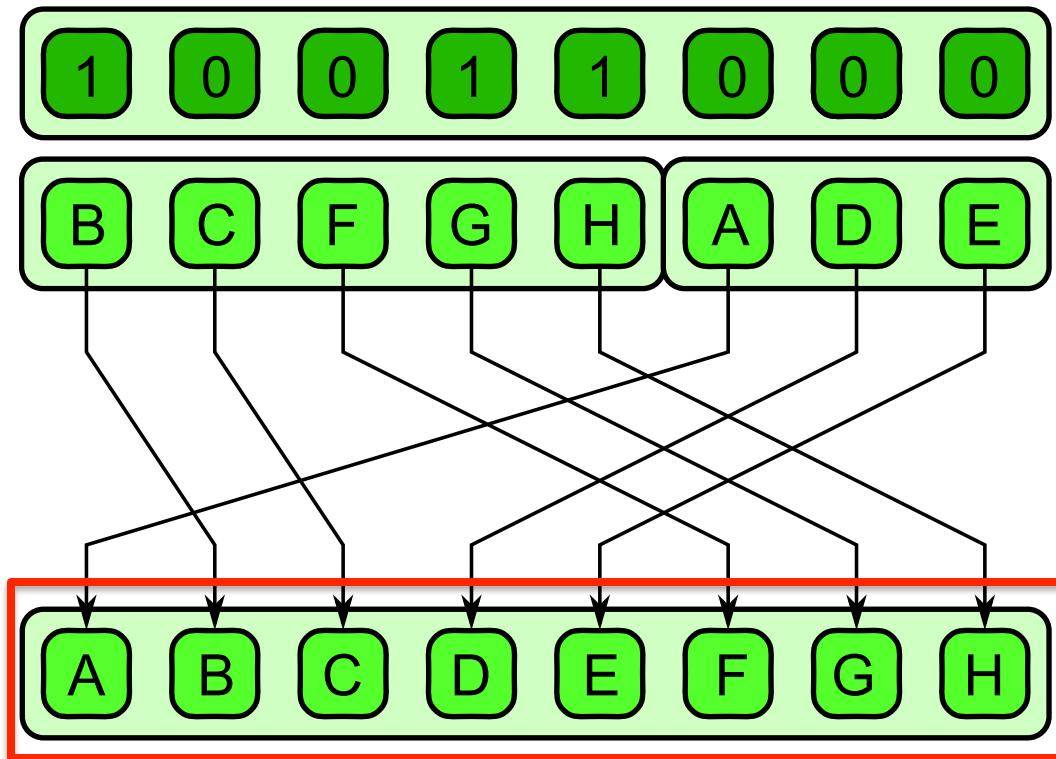
Generalization of Pack: Split



Lower half of output collection: values equal to 1

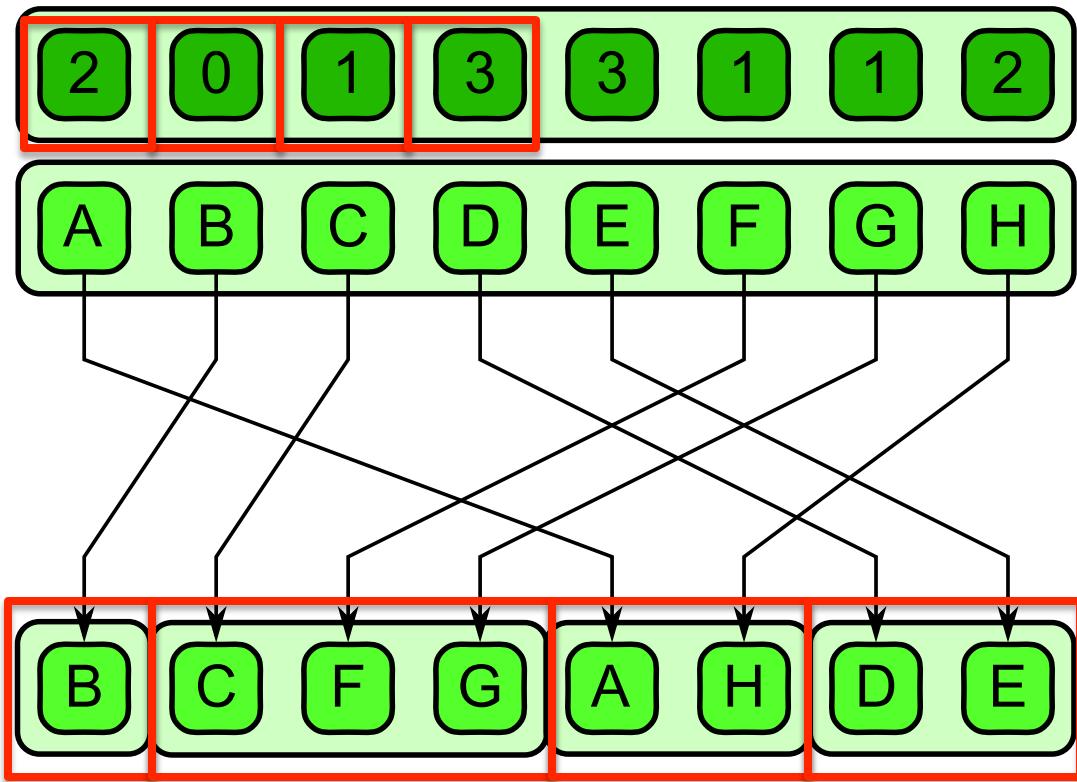
- Generalization of pack pattern
- Elements are moved to upper or lower half of output collection based on some state
- Does not lose information like pack

Generalization of Pack: Unsplit



- Inverse of split
- Creates **output collection** based on original input collection

Generalization of Pack: Bin



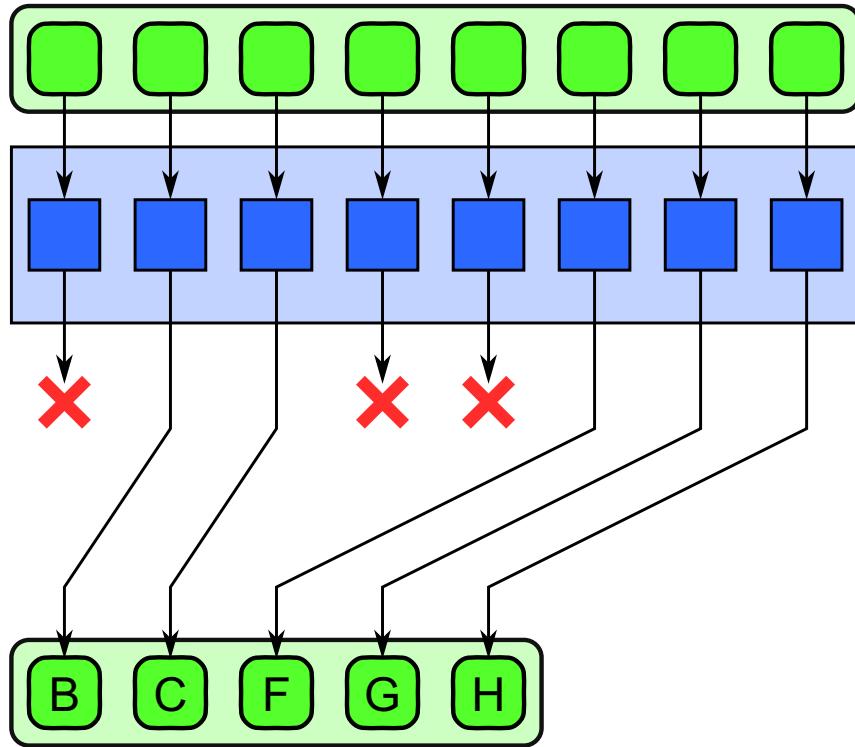
- Generalized split to support more categories (>2)
- Examples
 - Radix sort
 - Pattern classification

4 different categories = 4 bins

Table of Contents

- Gather Pattern
 - Shifts, Zip, Unzip
- Scatter Pattern
 - Collision Rules: atomic, permutation, merge, priority
- Pack Pattern
 - Split, Unsplit, Bin
 - Fusing Map and Pack
 - Expand
- Partitioning Data
- AoS vs. SoA
- Example Implementation: AoS vs. SoA

Fusion of Map and Pack

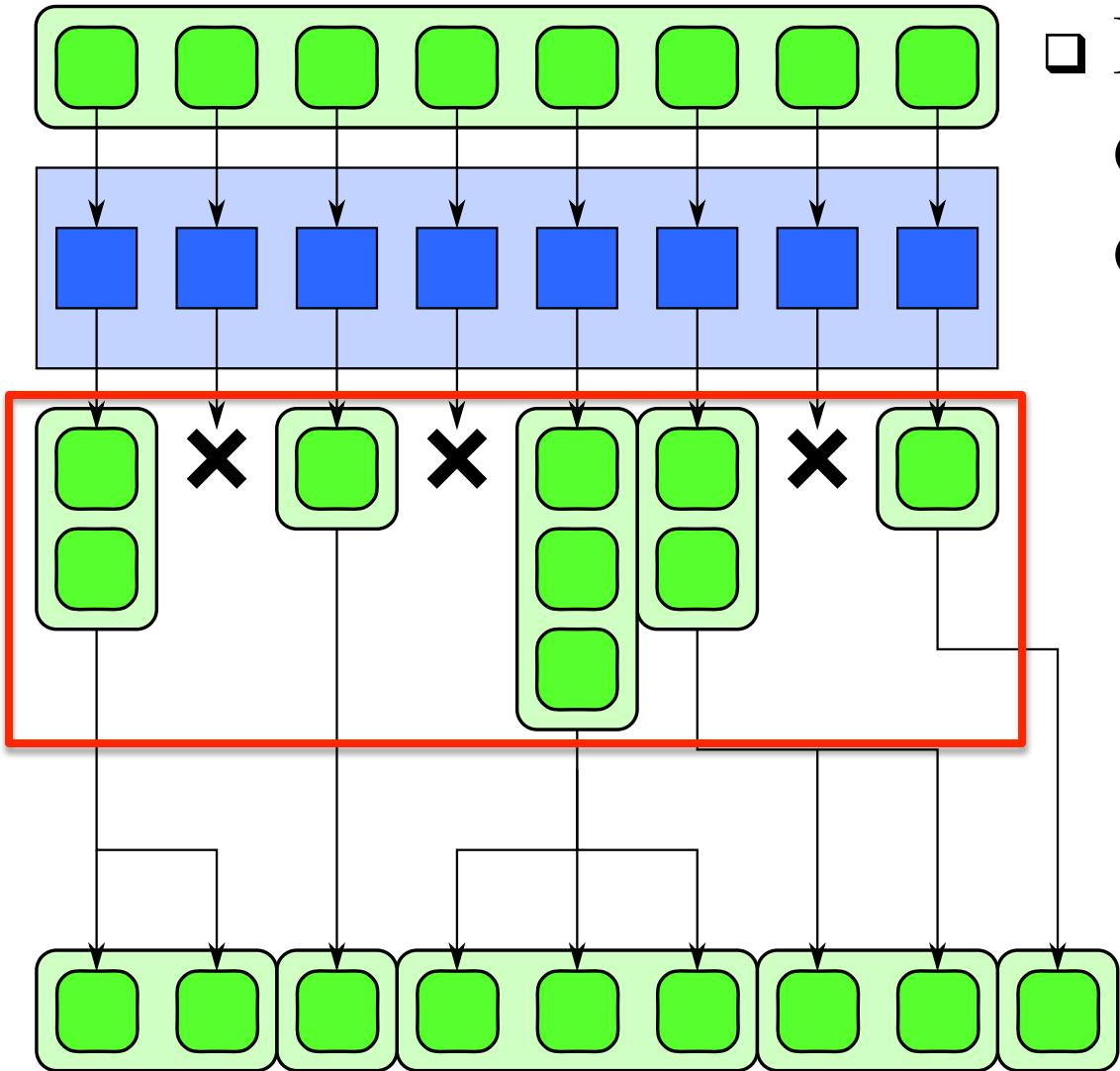


- Advantageous if most of the elements of a map are discarded
- **Map** checks pairs for collision
- **Pack** stores only actual collisions
- Output BW ~ results reported, not number of pairs tested
- Each element can output 0 or 1 element

Table of Contents

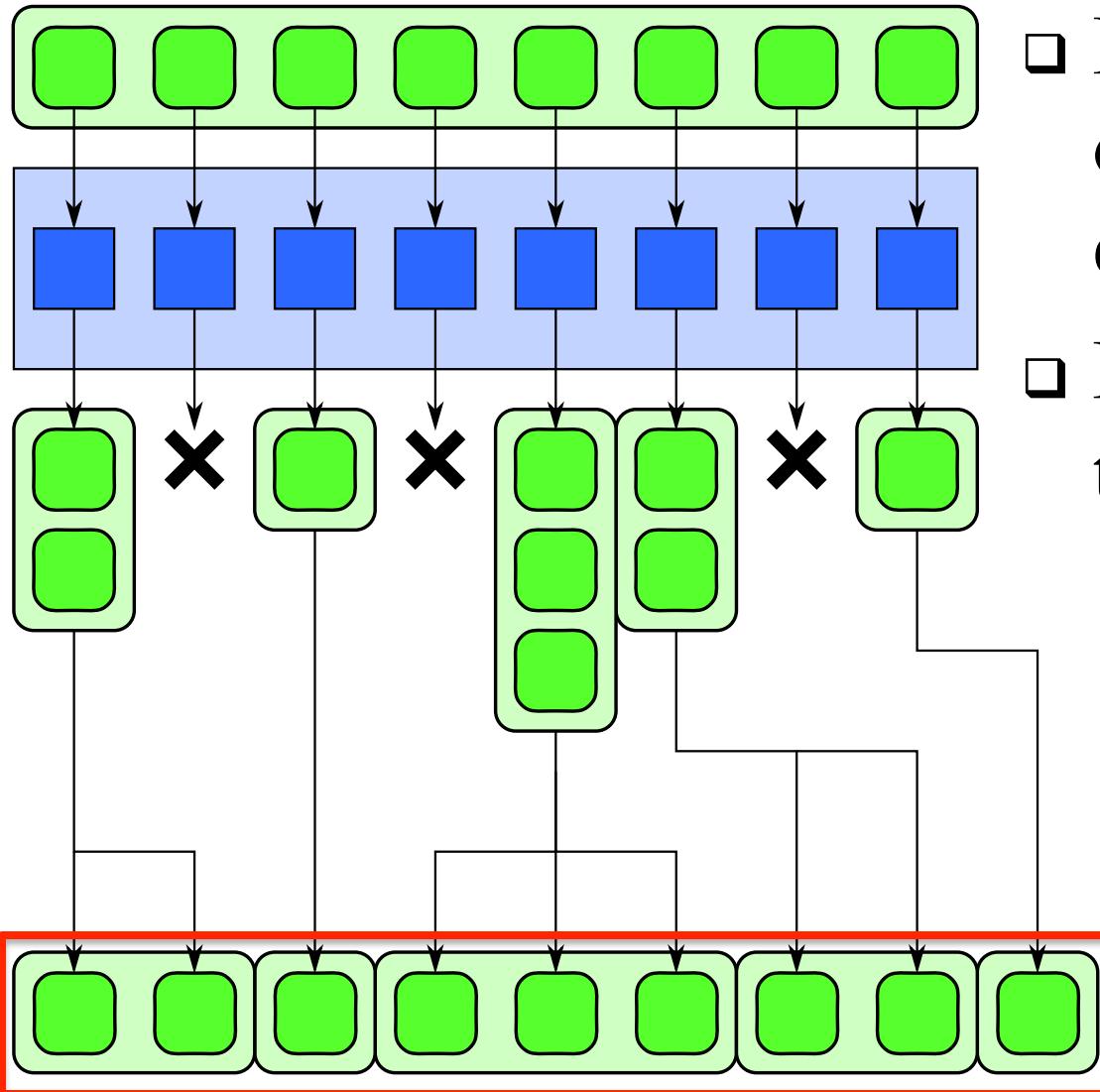
- Gather Pattern
 - Shifts, Zip, Unzip
- Scatter Pattern
 - Collision Rules: atomic, permutation, merge, priority
- Pack Pattern
 - Split, Unsplit, Bin
 - Fusing Map and Pack
 - **Expand**
- Partitioning Data
- AoS vs. SoA
- Example Implementation: AoS vs. SoA

Generalization of Pack: Expand



- Each element can output any number of elements

Generalization of Pack: Expand



- Each element can output any number of elements
- **Results are fused together in order**

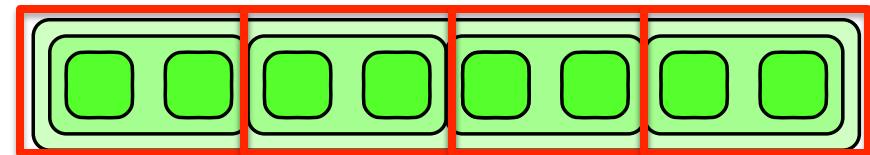
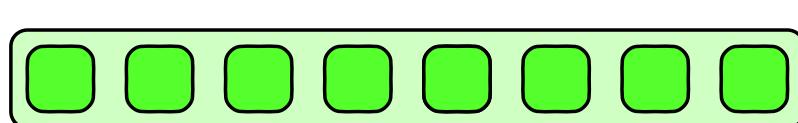
Table of Contents

- Gather Pattern
 - Shifts, Zip, Unzip
- Scatter Pattern
 - Collision Rules: atomic, permutation, merge, priority
- Pack Pattern
 - Split, Unsplit, Bin
 - Fusing Map and Pack
 - Expand
- Partitioning Data
- AoS vs. SoA
- Example Implementation: AoS vs. SoA

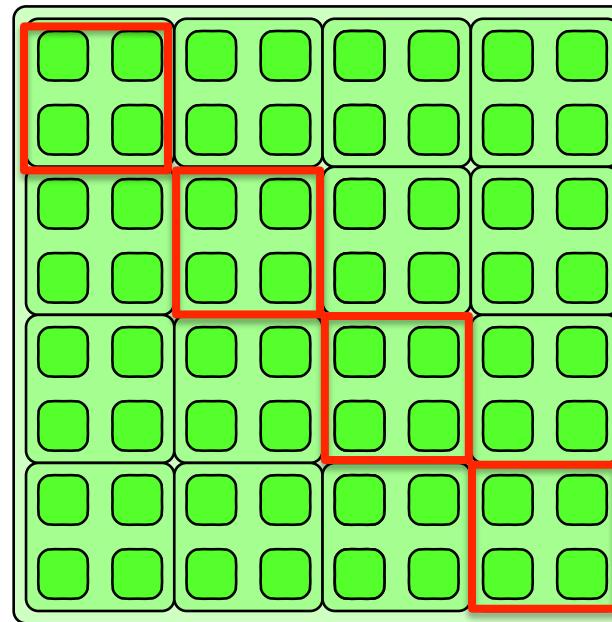
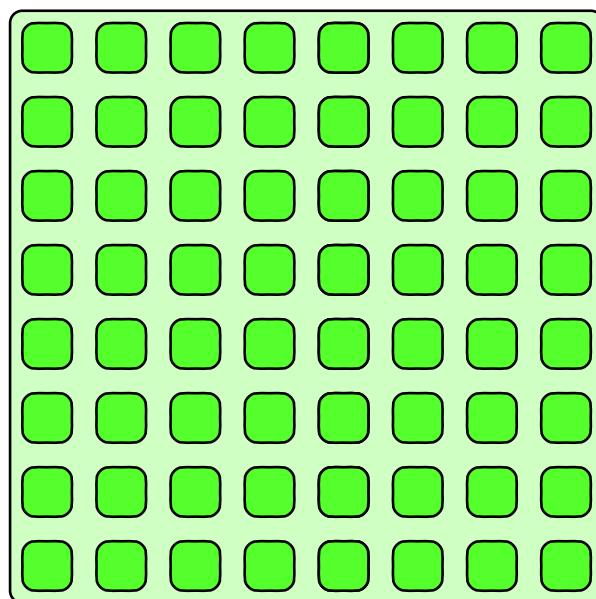
Parallelizing Algorithms

- Common strategy:
 1. Divide up the computational domain into sections
 2. Work on the sections individually
 3. Combine the results
- Methods
 - Divide-and-conquer
 - Fork-join (discussed in Chapter 8)
 - Geometric decomposition
 - Partitions
 - Segments

Partitioning



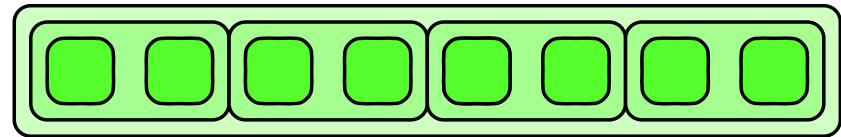
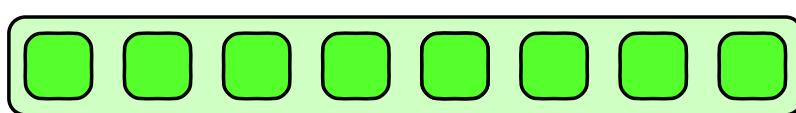
1D



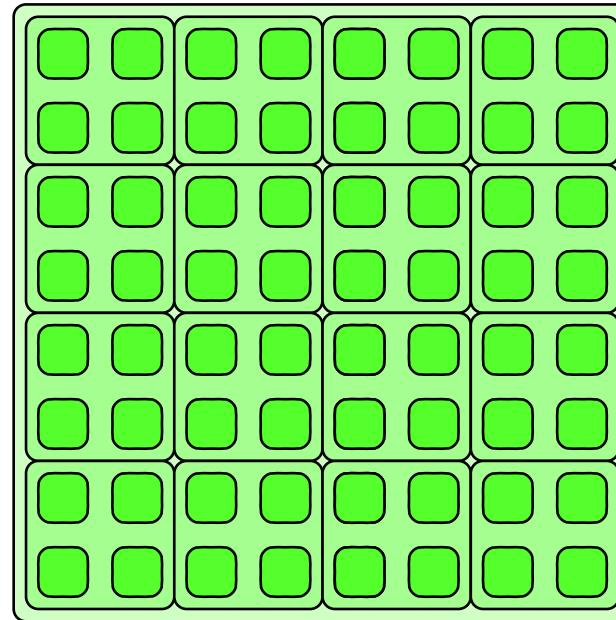
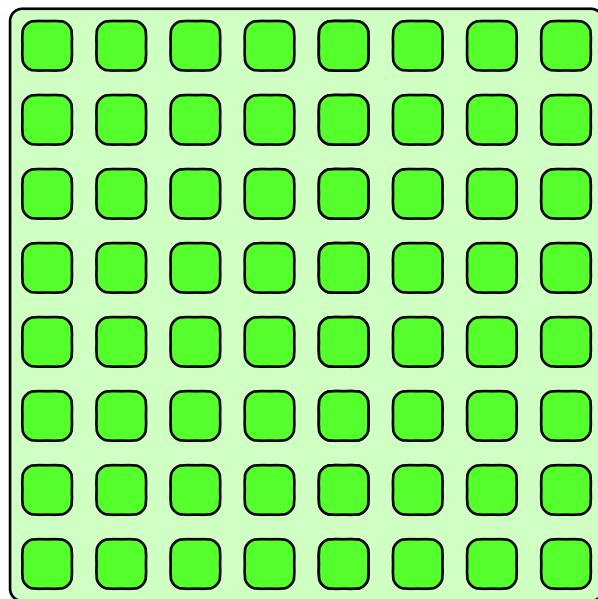
2D

- Data is divided into
 - **non-overlapping**
 - **equal-sized** regions

Partitioning



1D

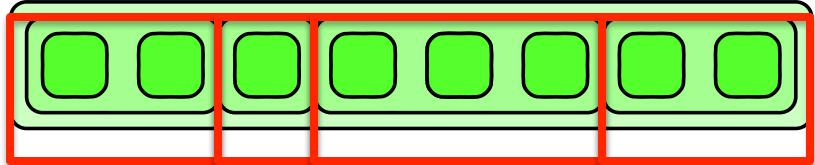
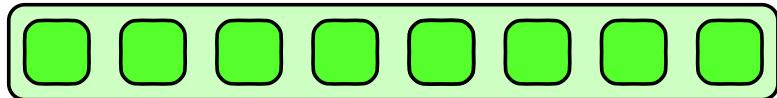


2D

- Data is divided into
 - **non-overlapping** ←
 - **equal-sized** regions

Avoid write conflicts and race conditions

Segmentation

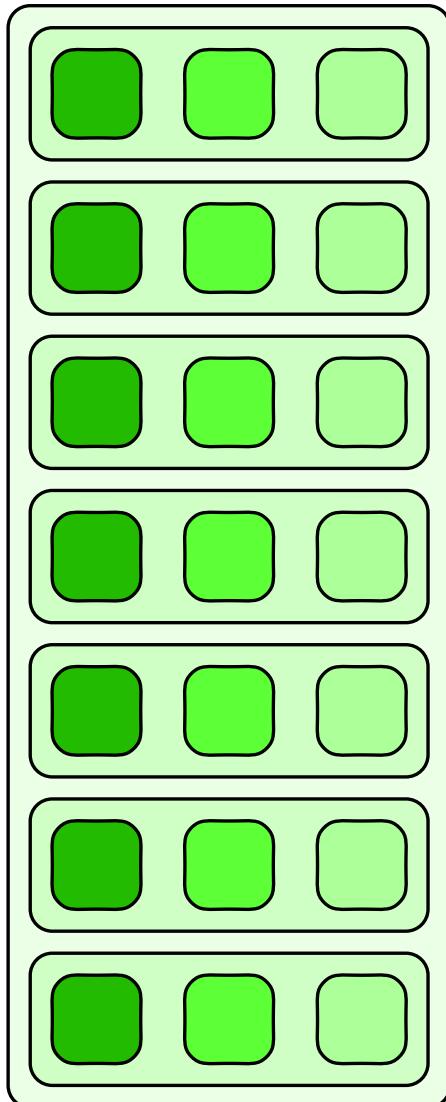


- Data is divided into **non-uniform** non-overlapping regions
- Start of each segment can be marked using:
 - Array of integers
 - Array of Boolean flags

Table of Contents

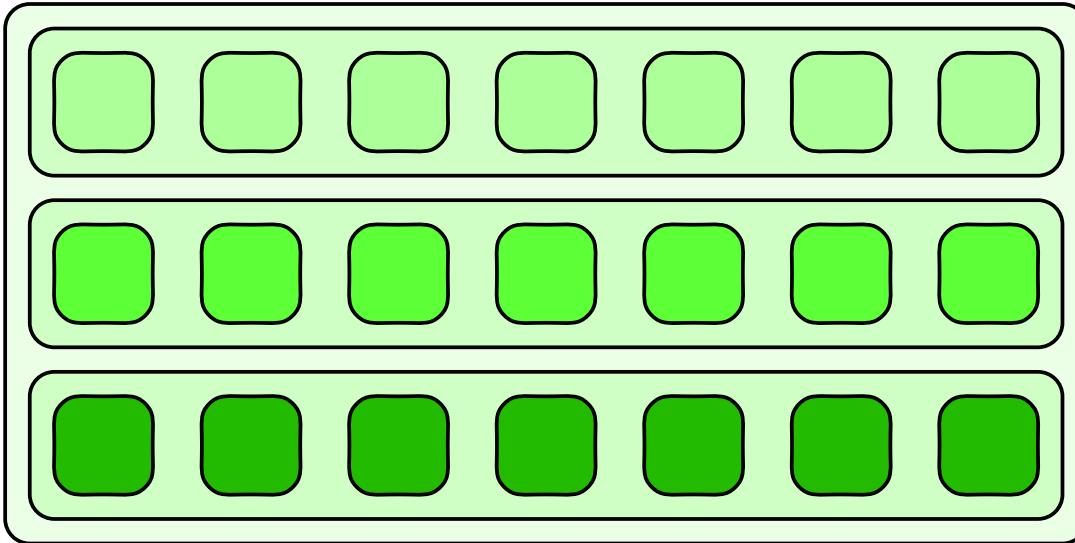
- Gather Pattern
 - Shifts, Zip, Unzip
- Scatter Pattern
 - Collision Rules: atomic, permutation, merge, priority
- Pack Pattern
 - Split, Unsplit, Bin
 - Fusing Map and Pack
 - Expand
- Partitioning Data
- AoS vs. SoA
- Example Implementation: AoS vs. SoA

Array of Structures (AoS)



- May lead to better cache utilization if data is accessed randomly

Structures of Arrays (SoA)



- Typically better for vectorization and avoidance of false sharing

Data Layout Options

Array of Structures (AoS), padding at end



Array of Structures (AoS), padding after each structure



Structure of Arrays (SoA), padding at end



Structure of Arrays (SoA), padding after each component



Table of Contents

- Gather Pattern
 - Shifts, Zip, Unzip
- Scatter Pattern
 - Collision Rules: atomic, permutation, merge, priority
- Pack Pattern
 - Split, Unsplit, Bin
 - Fusing Map and Pack
 - Expand
- Partitioning Data
- AoS vs. SoA
- Example Implementation: AoS vs. SoA

Example Implementation

AoS Code

```
struct node {  
    float x, y, z;  
};  
struct node NODES[1024];  
  
float dist[1024];  
for(i=0;i<1024;i+=16){  
    float x[16],y[16],z[16],d[16];  
    x[:] = NODES[i:16].x;  
    y[:] = NODES[i:16].y;  
    z[:] = NODES[i:16].z;  
    d[:] = sqrtf(x[:] * x[:] + y[:] * y[:] + z[:] * z[:]);  
    dist[i:16] = d[:];  
}
```

SoA Code

```
struct node1 {  
    float x[1024], y[1024], z[1024];  
}  
struct node1 NODES1;  
  
float dist[1024];  
for(i=0;i<1024;i+=16){  
    float x[16],y[16],z[16],d[16];  
    x[:] = NODES1.x[i:16];  
    y[:] = NODES1.y[i:16];  
    z[:] = NODES1.z[i:16];  
    d[:] = sqrtf(x[:] * x[:] + y[:] * y[:] + z[:] * z[:]);  
    dist[i:16] = d[:];  
}
```

AoS Code

```
struct node {  
    float x, y, z;  
};  
struct node NODES[1024];  
  
float dist[1024];  
for(i=0;i<1024;i+=16){  
    float x[16],y[16],z[16],d[16];  
    x[:] = NODES[i:16].x;  
    y[:] = NODES[i:16].y;  
    z[:] = NODES[i:16].z;  
    d[:] = sqrtf(x[:]*x[:] + y[:]*y[:] + z[:]*z[:]);  
    dist[i:16] = d[:];  
}
```

- Most logical data organization layout
- Extremely difficult to access memory for reads (gathers) and writes (scatters)
- Prevents efficient vectorization

SoA Code

```
struct node1 {  
    float x[1024], y[1024], z[1024];  
}  
struct node1 NODES1;  
  
float dist[1024];  
for(i=0;i<1024;i+=16){  
    float x[16],y[16],z[16],d[16];  
    x[:] = NODES1.x[i:16];  
    y[:] = NODES1.y[i:16];  
    z[:] = NODES1.z[i:16];  
    d[:] = sqrtf(x[:]*x[:] + y[:]*y[:] + z[:]*z[:]);  
    dist[i:16] = d[:];  
}
```

- Separate arrays for each structure-field keeps memory accesses contiguous when vectorization is performed over structure instances



Stencil Pattern

Parallel Computing

CIS 410/510

Department of Computer and Information Science

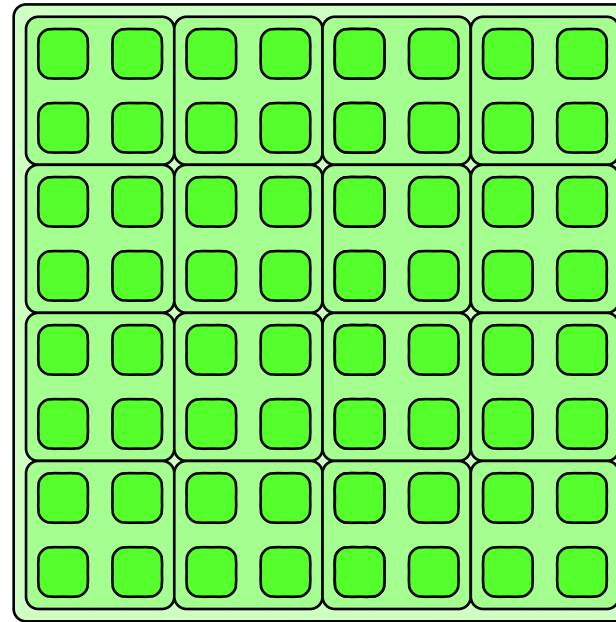
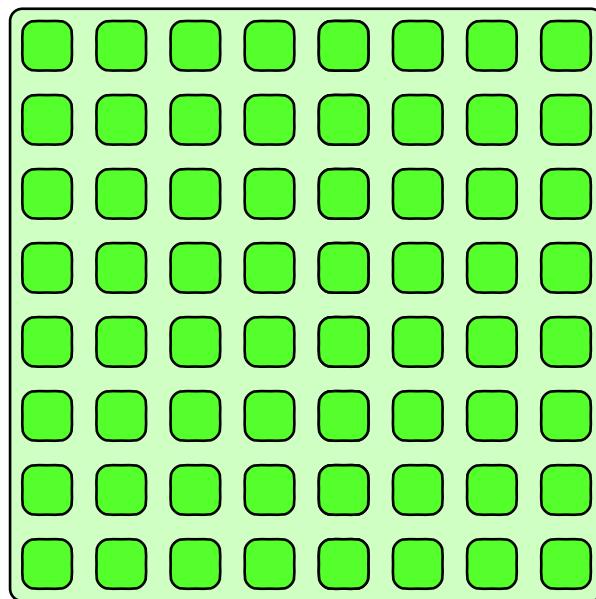
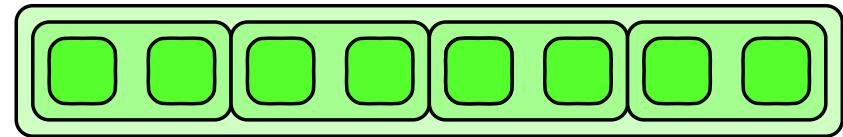
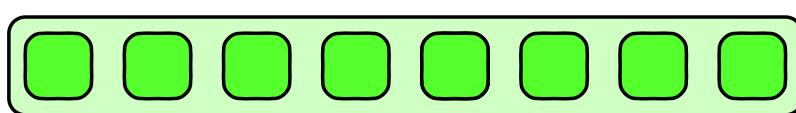


UNIVERSITY OF OREGON

Outline

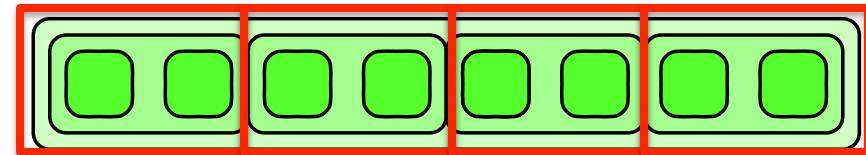
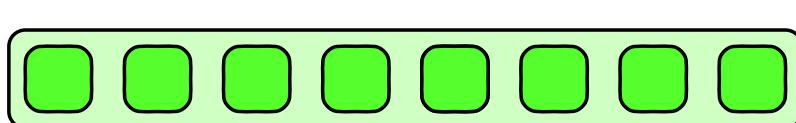
- Partitioning
- What is the stencil pattern?
 - Update alternatives
 - 2D Jacobi iteration
 - SOR and Red/Black SOR
- Implementing stencil with shift
- Stencil and cache optimizations
- Stencil and communication optimizations
- Recurrence

Partitioning

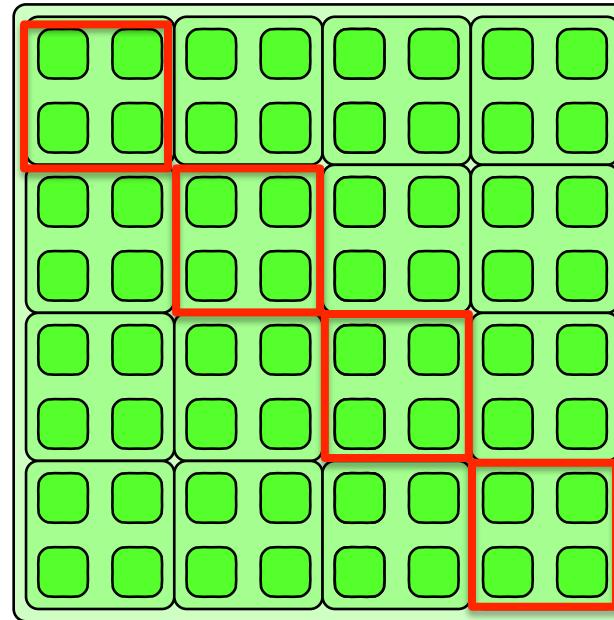
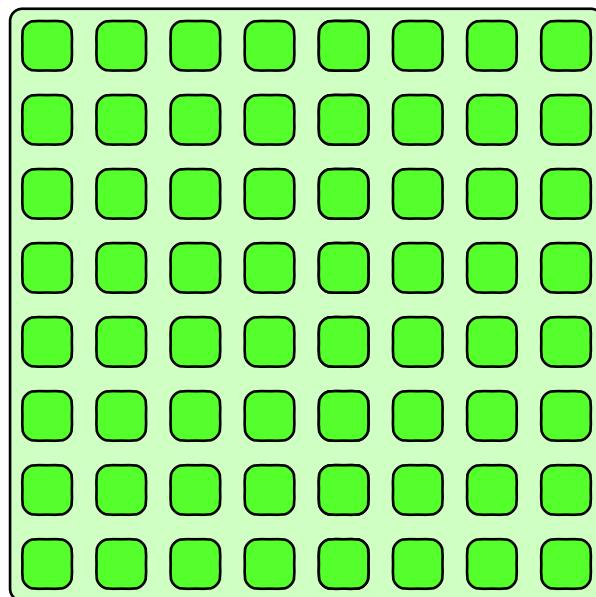


- Data is divided into
 - **non-overlapping** regions (avoid write conflicts, race conditions)
 - **equal-sized** regions (improve load balancing)

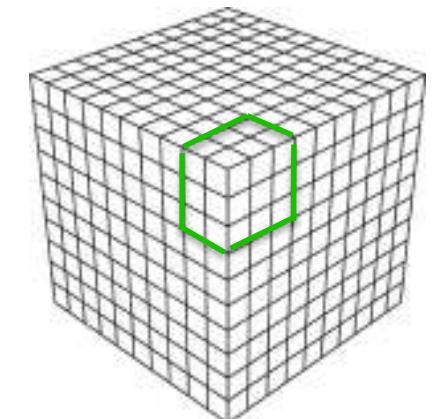
Partitioning



1D



2D



3D

- Data is divided into
 - **non-overlapping** regions (avoid write conflicts, race conditions)
 - **equal-sized** regions (improve load balancing)

Outline

- Partitioning
- What is the stencil pattern?
 - Update alternatives
 - 2D Jacobi iteration
 - SOR and Red/Black SOR
- Implementing stencil with shift
- Stencil and cache optimizations
- Stencil and communication optimizations
- Recurrence

Stencil Pattern

- A stencil pattern is a map where each output depends on a “neighborhood” of inputs
- These inputs are a set of fixed offsets relative to the output position
- A stencil output is a function of a “neighborhood” of elements in an input collection
 - Applies the stencil to select the inputs
- Data access patterns of stencils are regular
 - Stencil is the “shape” of “neighborhood”
 - Stencil remains the same

Serial Stencil Example (part 1)

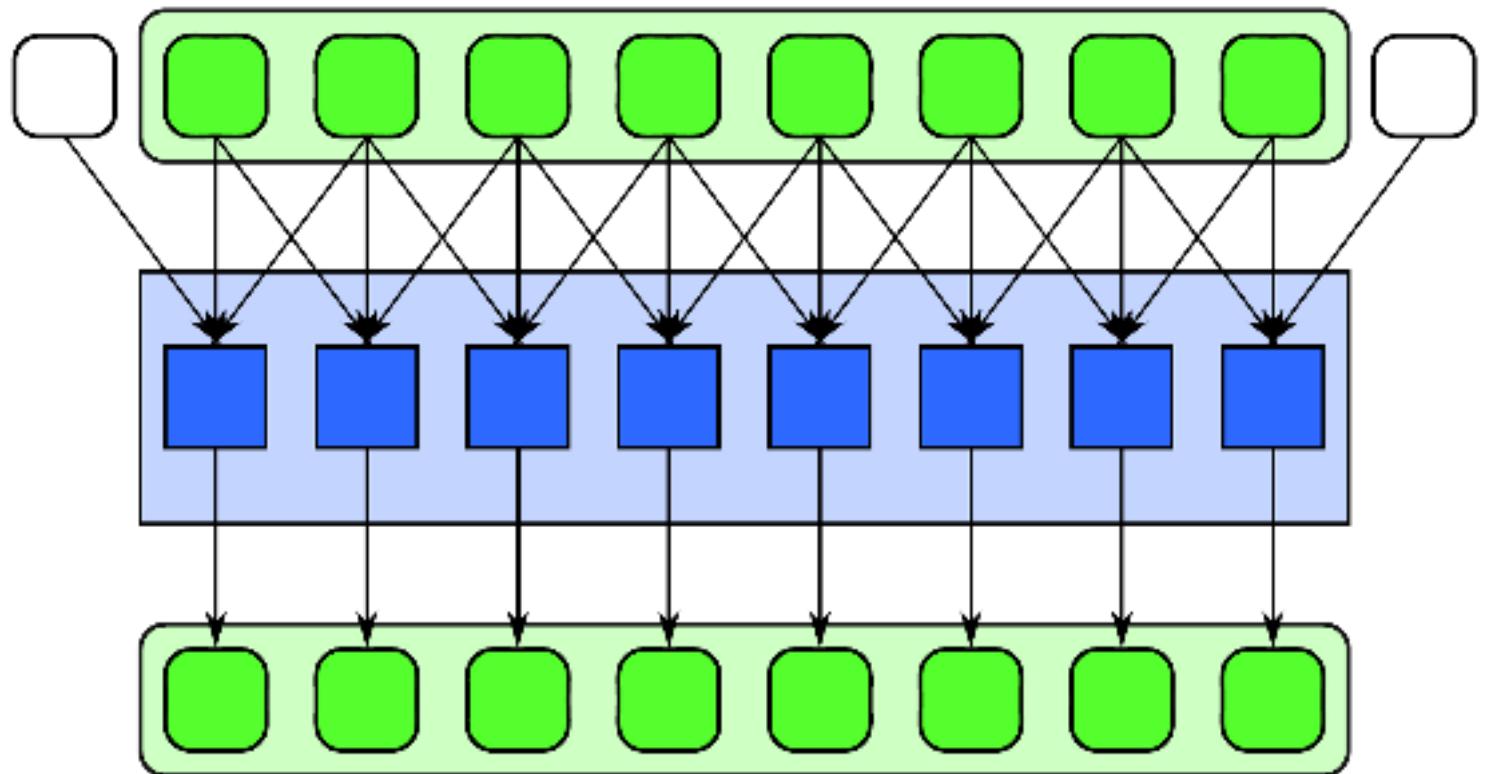
```
1 template<
2     int NumOff,      // number of offsets
3     typename In,     // type of input locations
4     typename Out,    // type of output locations
5     typename F       // type of function/funcitor
6 >
7 void stencil(
8     int n,          // number of elements in data collection
9     const In a[],   // input data collection (n elements)
10    Out r[],       // output data collection (n elements)
11    In b,          // boundary value
12    F func,        // function/funcitor from neighborhood inputs to output
13    const int offsets[] // offsets (NumOffsets elements)
14 ) {
```

Serial Stencil Example (part 2)

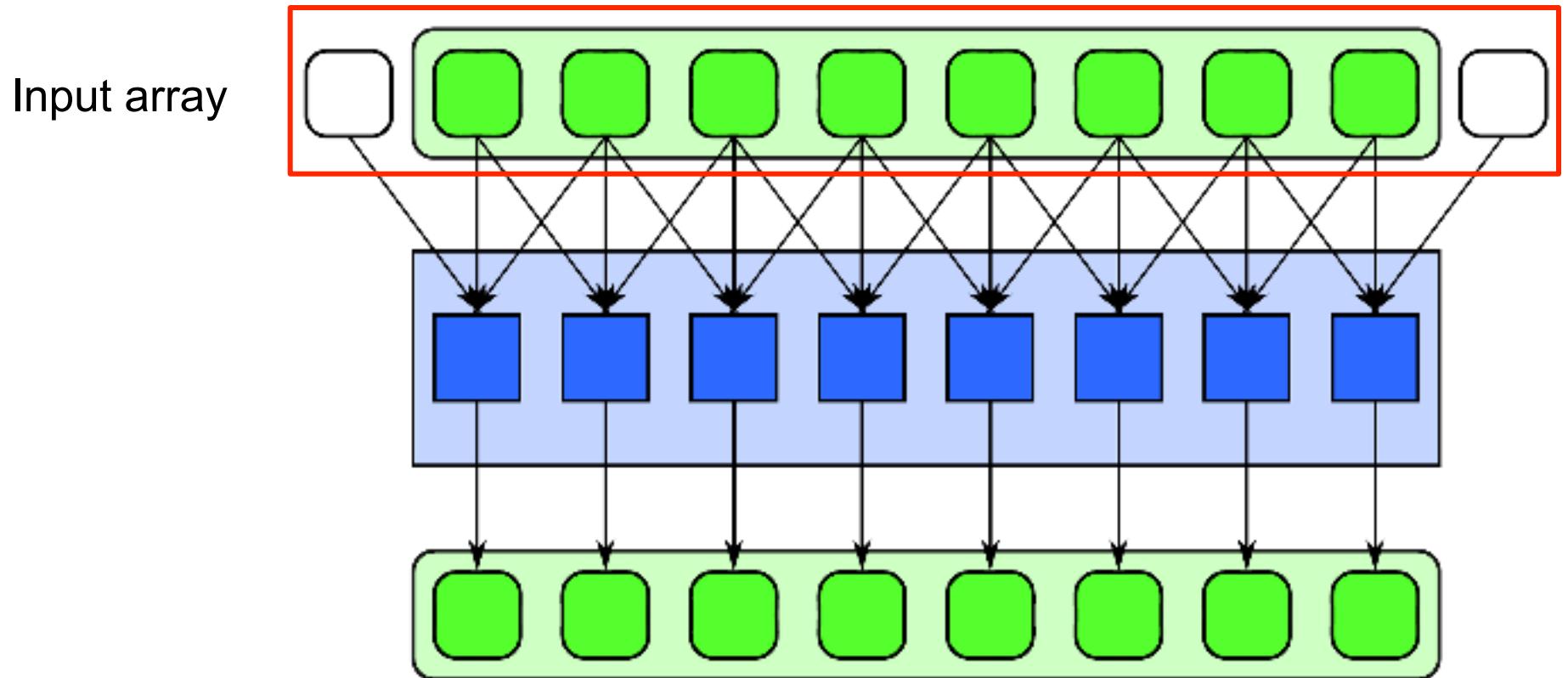
```
15 // array to hold neighbors
16 In neighborhood[NumOff];
17 // loop over all output locations
18 for (int i = 0; i < n; ++i) {
19     // loop over all offsets and gather neighborhood
20     for (int j = 0; j < NumOff; ++j) {
21         // get index of jth input location
22         int k = i+offsets[j];
23         if (0 <= k && k < n) {
24             // read input location
25             neighborhood[j] = a[k];
26         } else {
27             // handle boundary case
28             neighborhood[j] = b;
29         }
30     }
31     // compute output value from input neighborhood
32     r[i] = func(neighborhood);
33 }
34 }
```

How would we parallelize this?

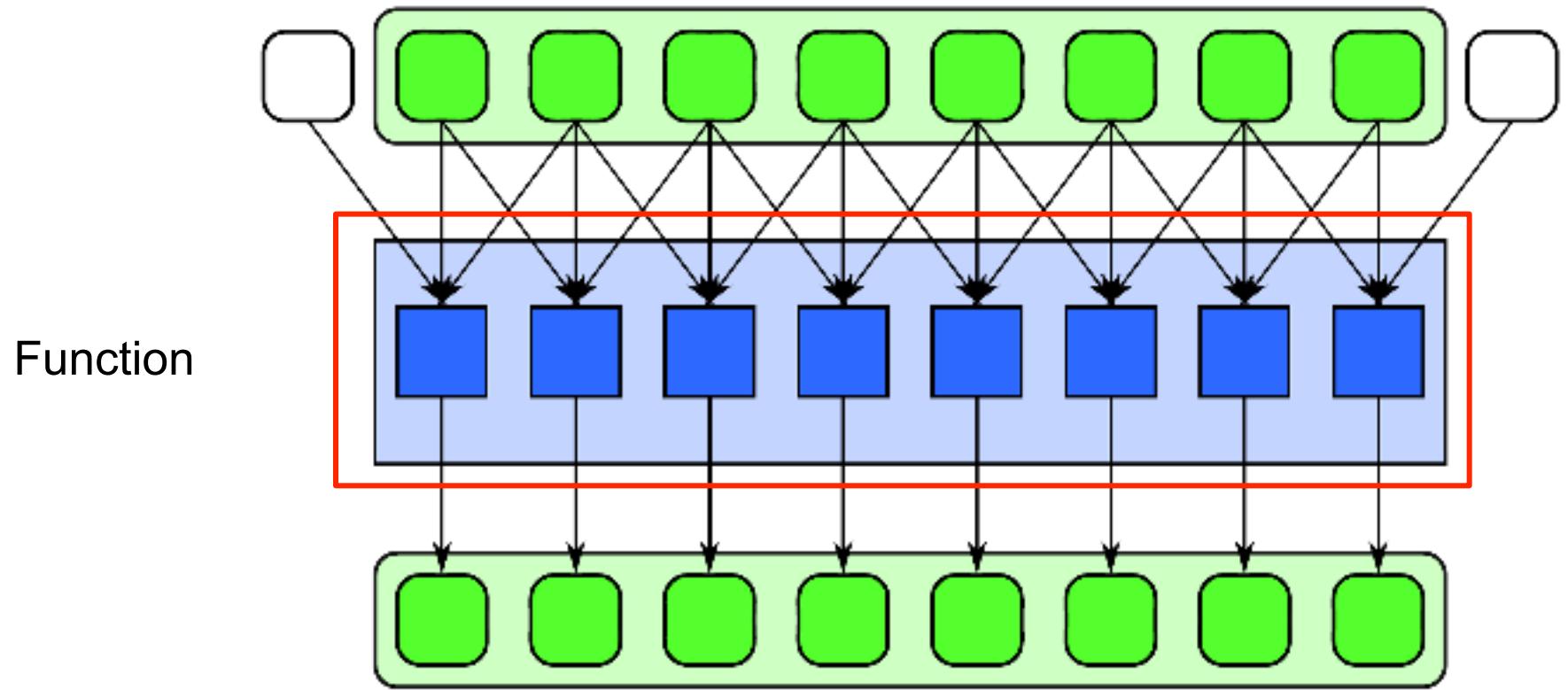
What is the stencil pattern?



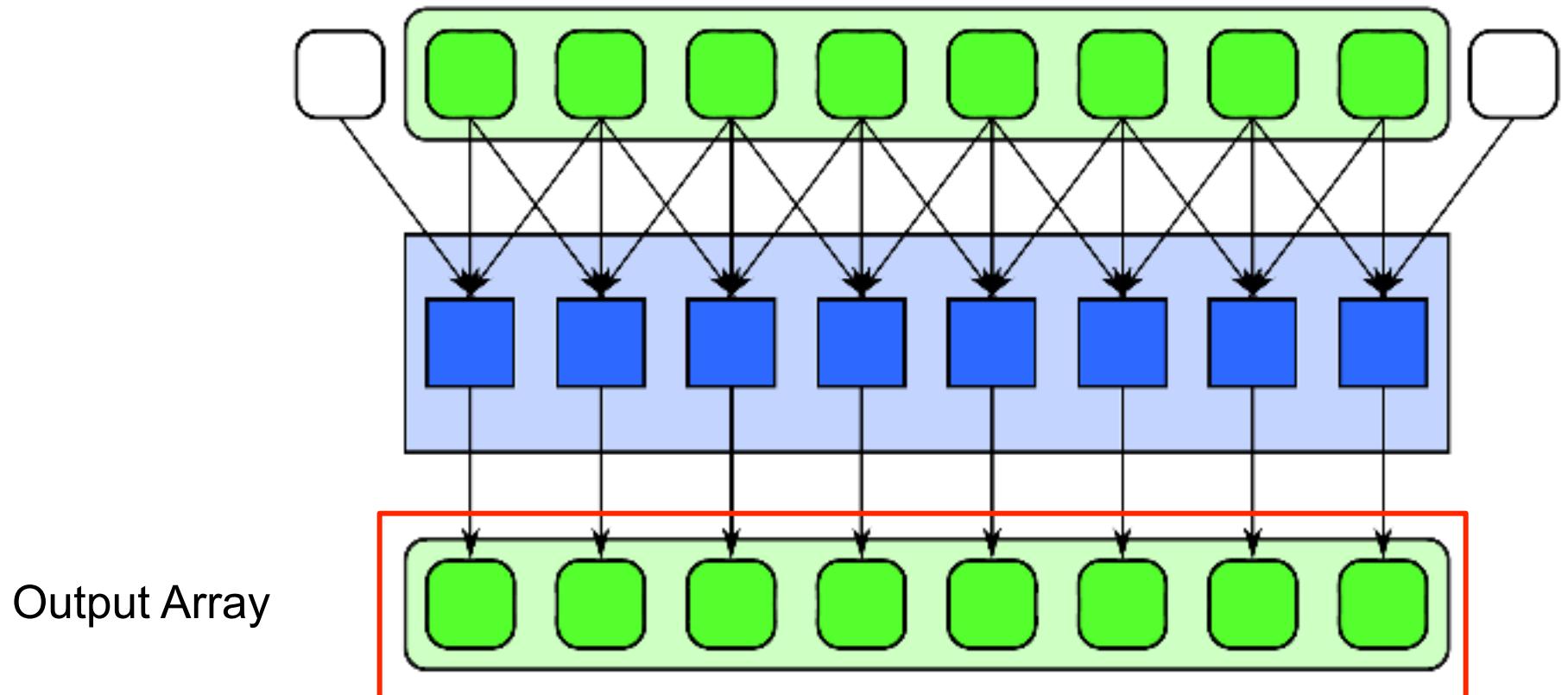
What is the stencil pattern?



What is the stencil pattern?

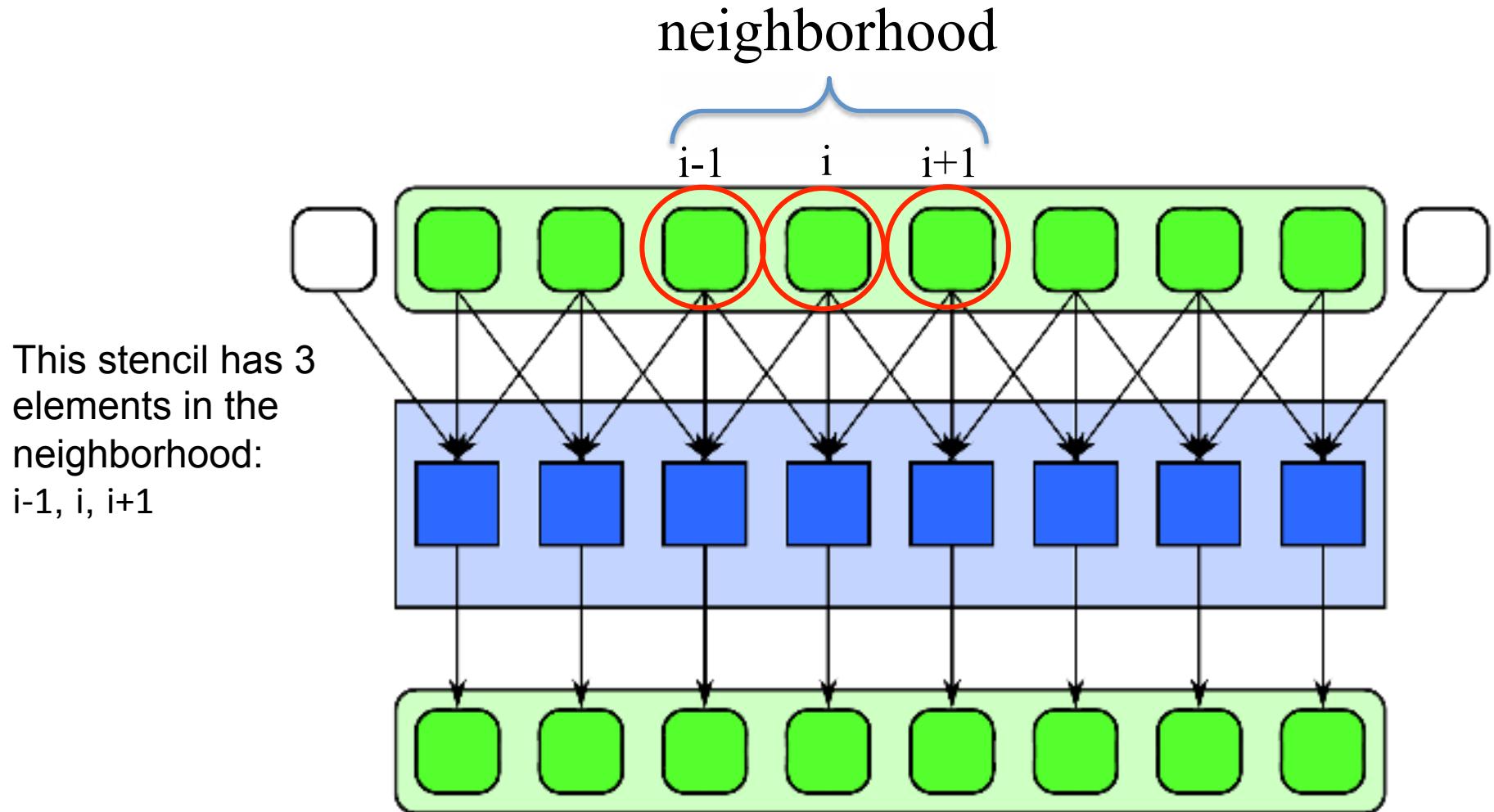


What is the stencil pattern?

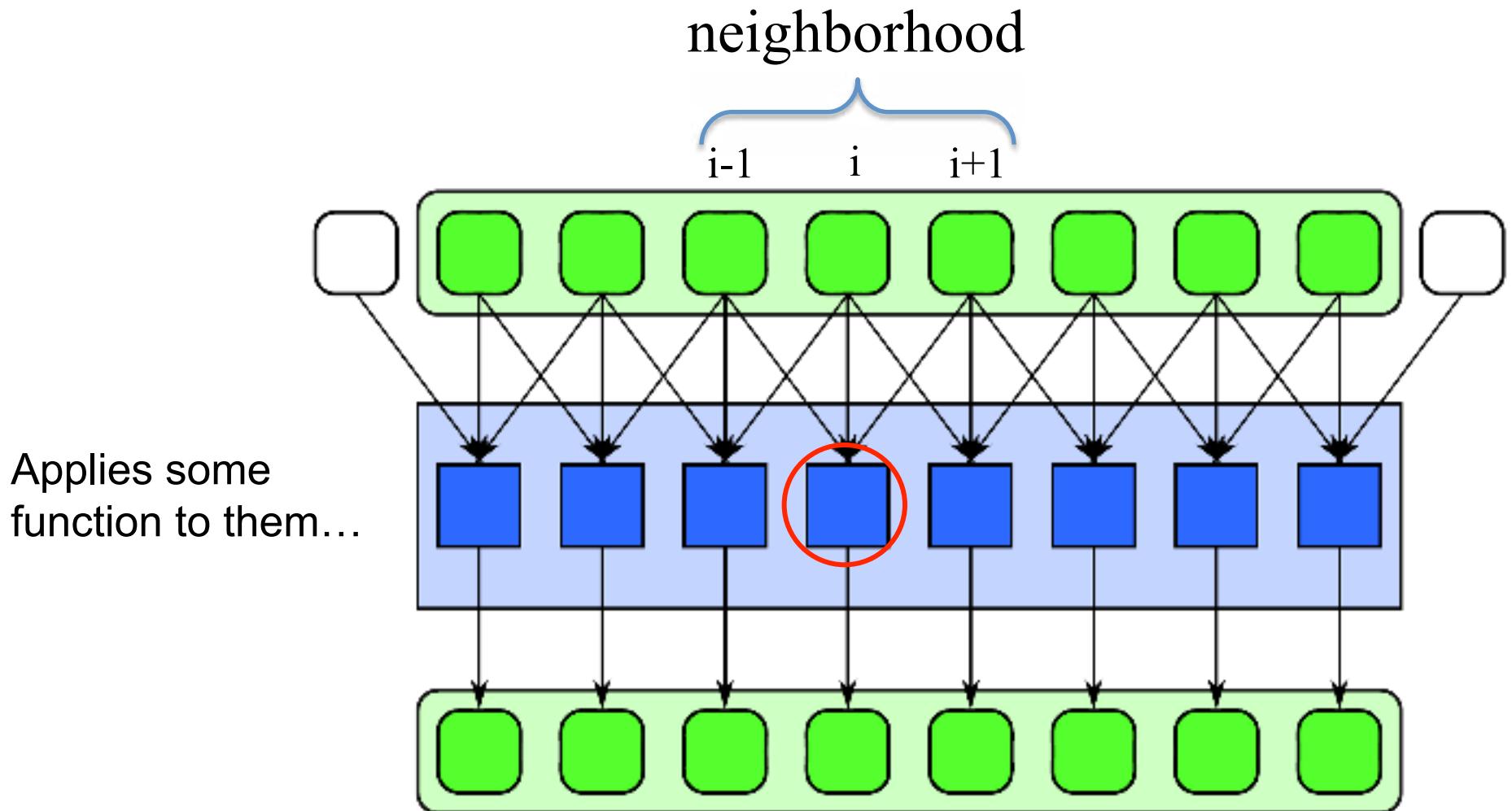


Output Array

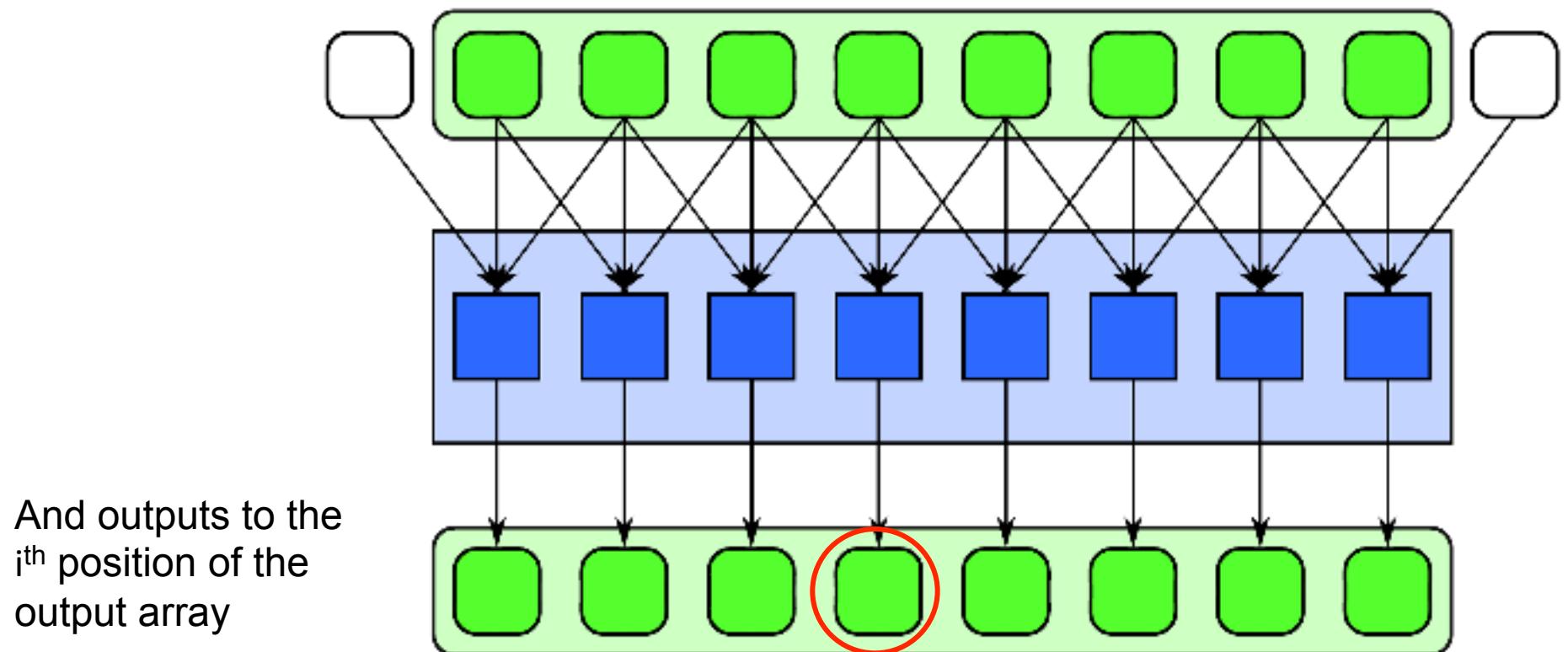
What is the stencil pattern?



What is the stencil pattern?



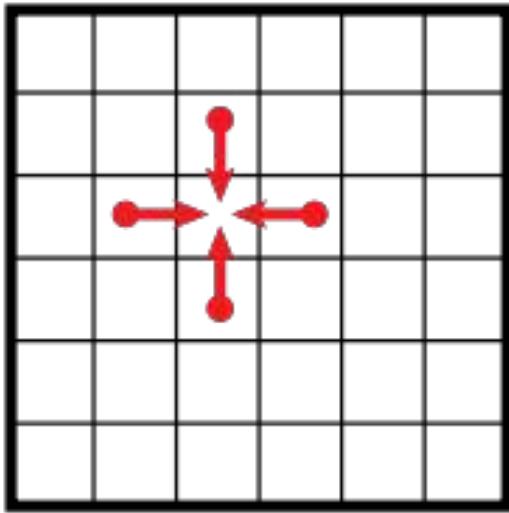
What is the stencil pattern?



Stencil Patterns

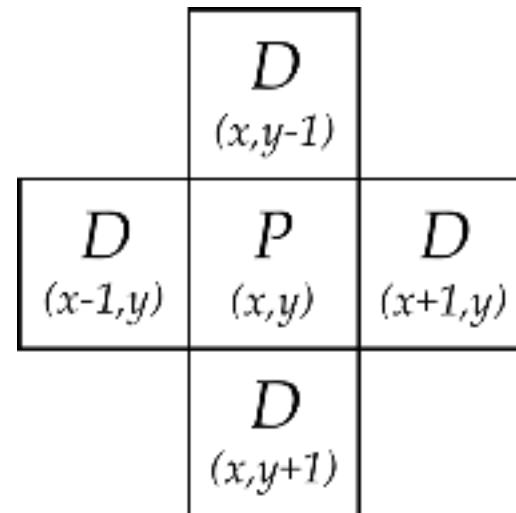
- Stencils can operate on one dimensional and multidimensional data
- Stencil neighborhoods can range from compact to sparse, square to cube, and anything else!
- It is the pattern of the stencil that determines how the stencil operates in an application

2-Dimensional Stencils



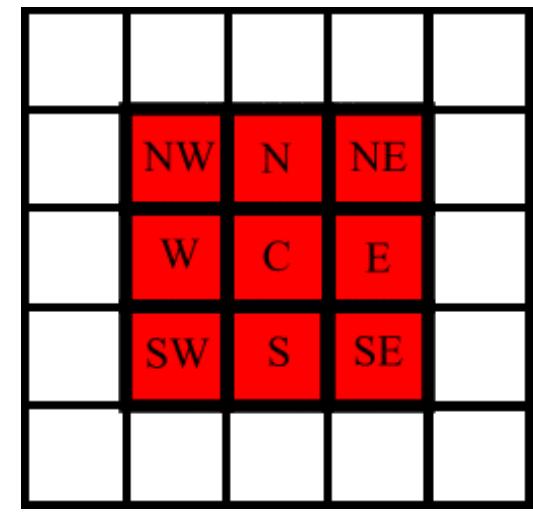
4-point stencil

Center cell (P)
is not used



5-point stencil

Center cell (P)
is used as well

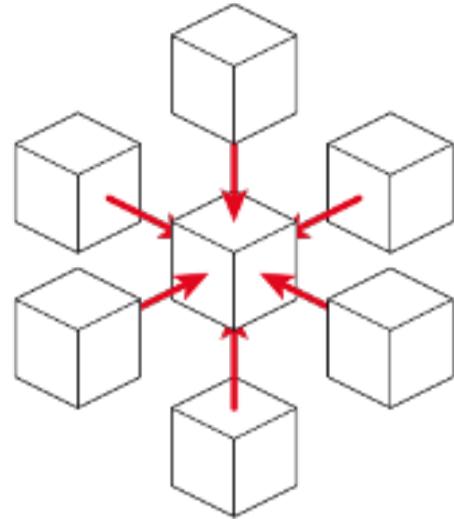


9-point stencil

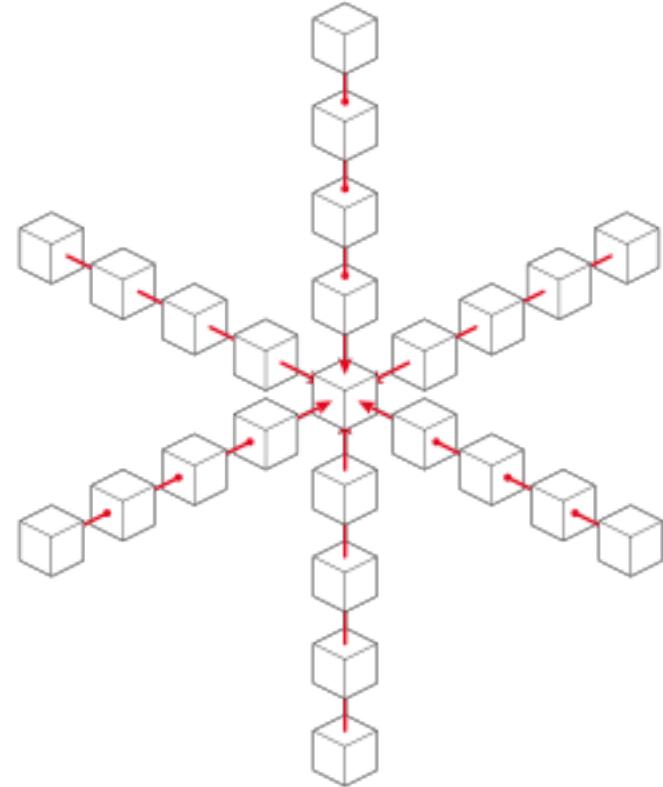
Center cell (C)
is used as well

Source: http://en.wikipedia.org/wiki/Stencil_code

3-Dimensional Stencils



6-point stencil
(7-point stencil)

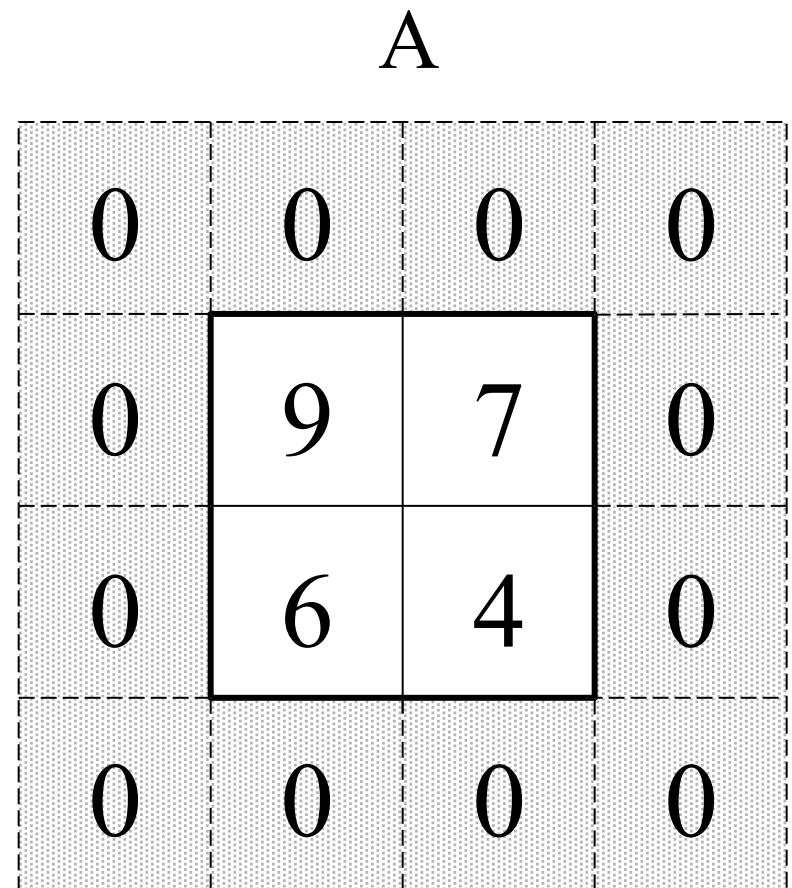


24-point stencil
(25-point stencil)

Source: http://en.wikipedia.org/wiki/Stencil_code

Stencil Example

- Here is our array, A



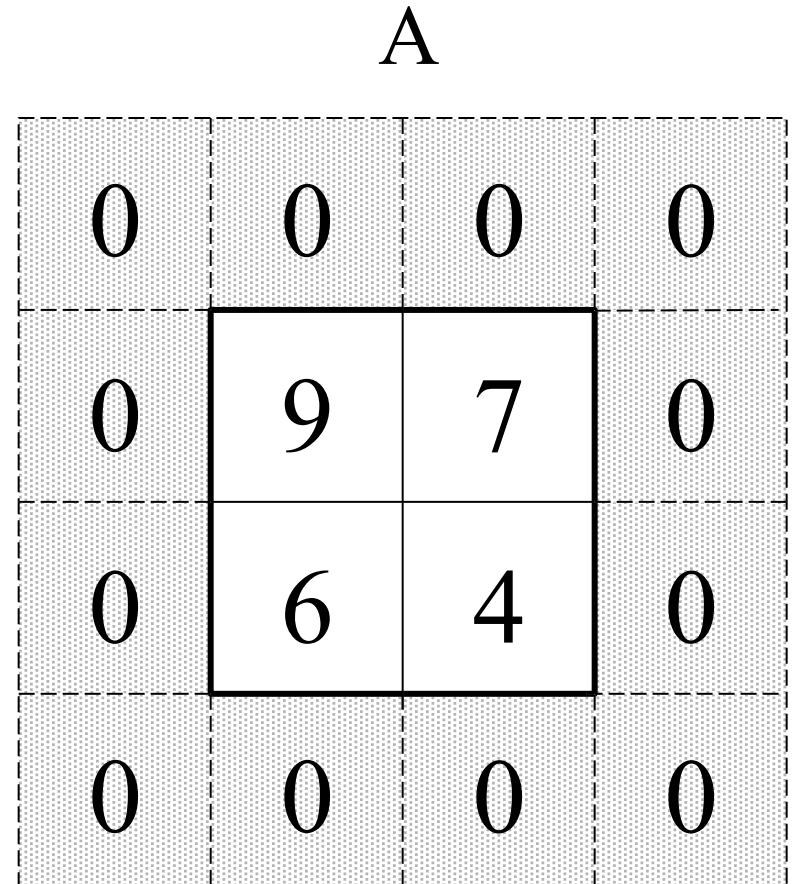
Stencil Example

- Here is our array **A**
- **B** is the output array
 - Initialize to all 0
- Apply a stencil operation to the inner square of the form:

$$B(i,j) = \text{avg}(\ A(i,j), \\ A(i-1,j), A(i+1,j), \\ A(i,j-1), A(i,j+1))$$

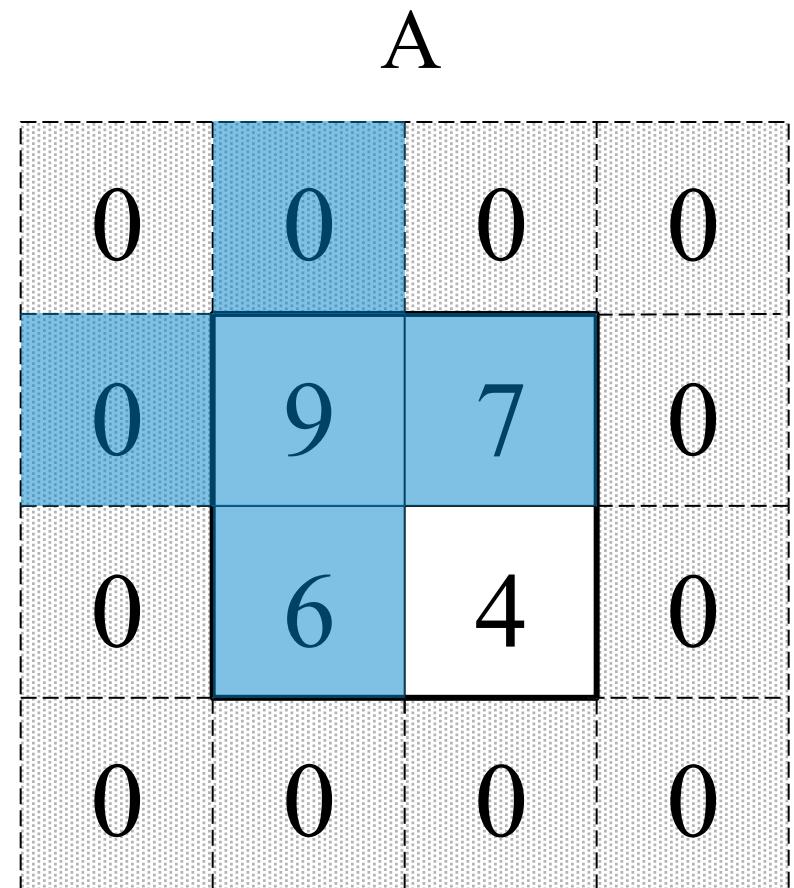
)

What is the stencil?



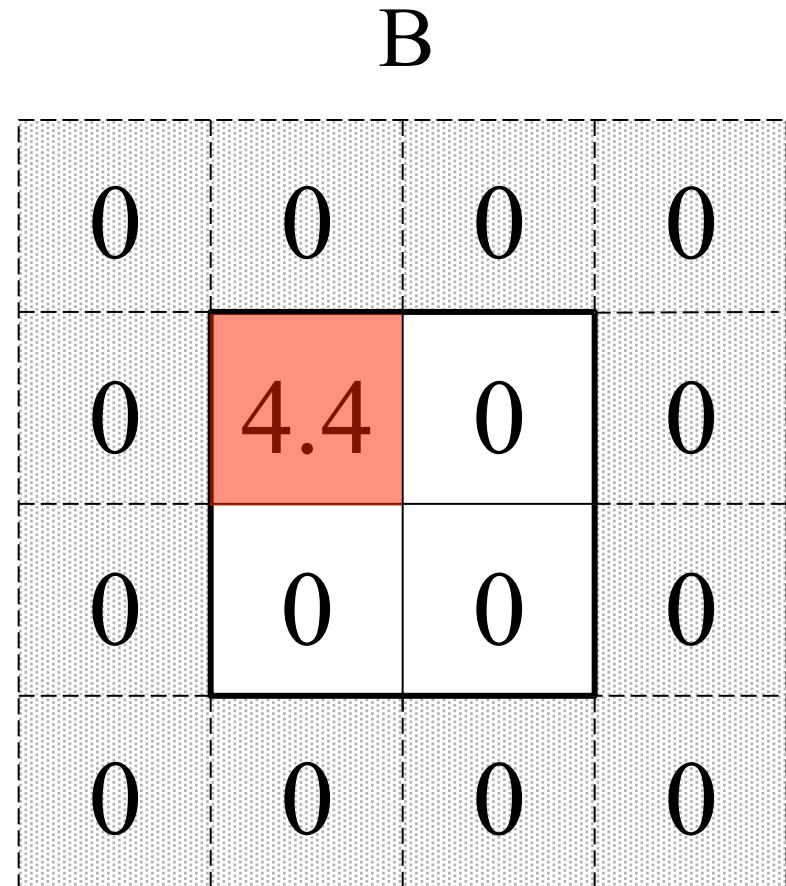
Stencil Pattern Procedure

- 1) Average all blue squares



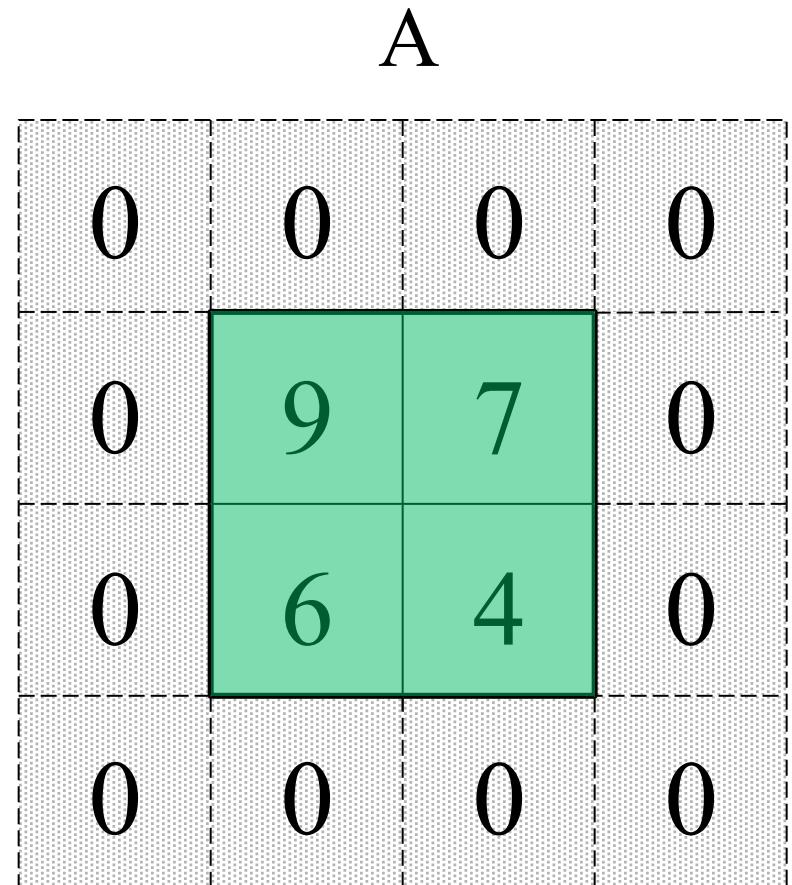
Stencil Pattern Procedure

- 1) Average all blue squares
- 2) Store result in B



Stencil Pattern Procedure

- 1) Average all blue squares
- 2) Store result in B
- 3) Repeat 1 and 2 for all green squares



Practice!

Stencil Pattern Practice

A

0	0	0	0
0	9	7	0
0	6	4	0
0	0	0	0

B

0	0	0	0
0	4.4	0	0
0	0	0	0
0	0	0	0

Stencil Pattern Practice

A

0	0	0	0
0	9	7	0
0	6	4	0
0	0	0	0

B

0	0	0	0
0	4.4	4.0	0
0	0	0	0
0	0	0	0

Stencil Pattern Practice

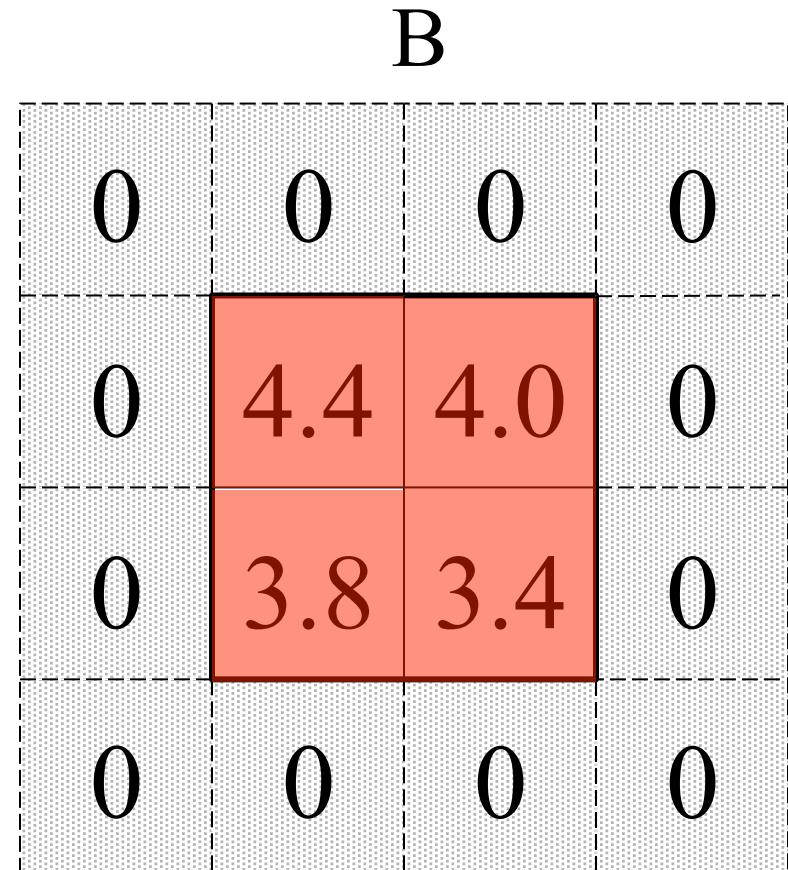
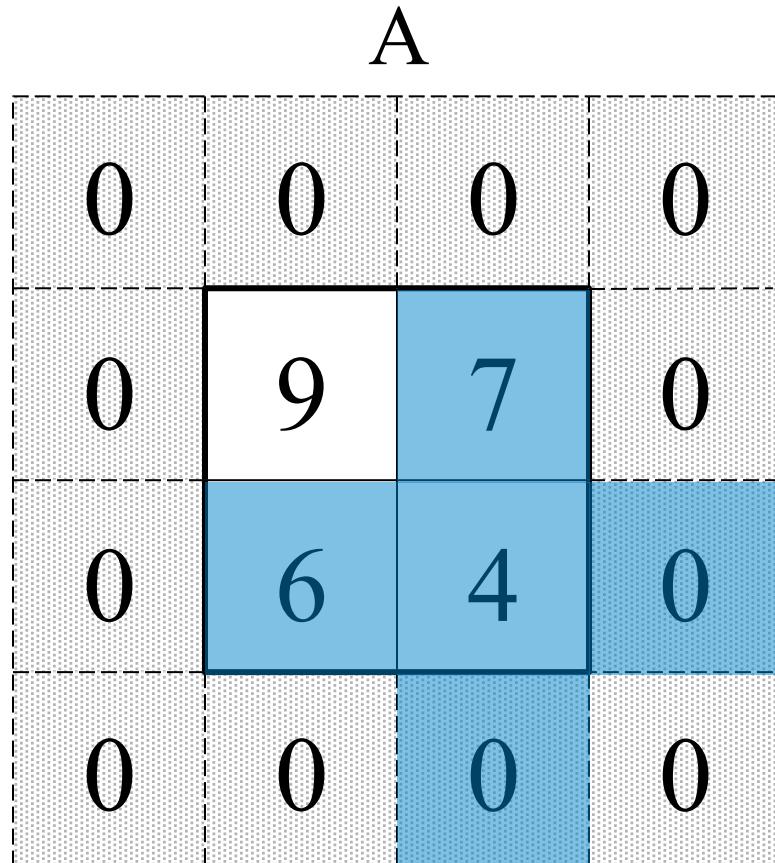
A

0	0	0	0
0	9	7	0
0	6	4	0
0	0	0	0

B

0	0	0	0
0	4.4	4.0	0
0	3.8	0	0
0	0	0	0

Stencil Pattern Practice



Outline

- Partitioning
- What is the stencil pattern?
 - Update alternatives
 - 2D Jacobi iteration
 - SOR and Red/Black SOR
- Implementing stencil with shift
- Stencil and cache optimizations
- Stencil and communication optimizations
- Recurrence

Serial Stencil Example (part 1)

```
1 template<
2     int NumOff,      // number of offsets
3     typename In,     // type of input locations
4     typename Out,    // type of output locations
5     typename F       // type of function/funcitor
6 >
7 void stencil(
8     int n,          // number of elements in data collection
9     const In a[],   // input data collection (n elements)
10    Out r[],      // output data collection (n elements)
11    In b,          // boundary value
12    F func,        // function/funcitor from neighborhood inputs to output
13    const int offsets[] // offsets (NumOffsets elements)
14 ) {
```

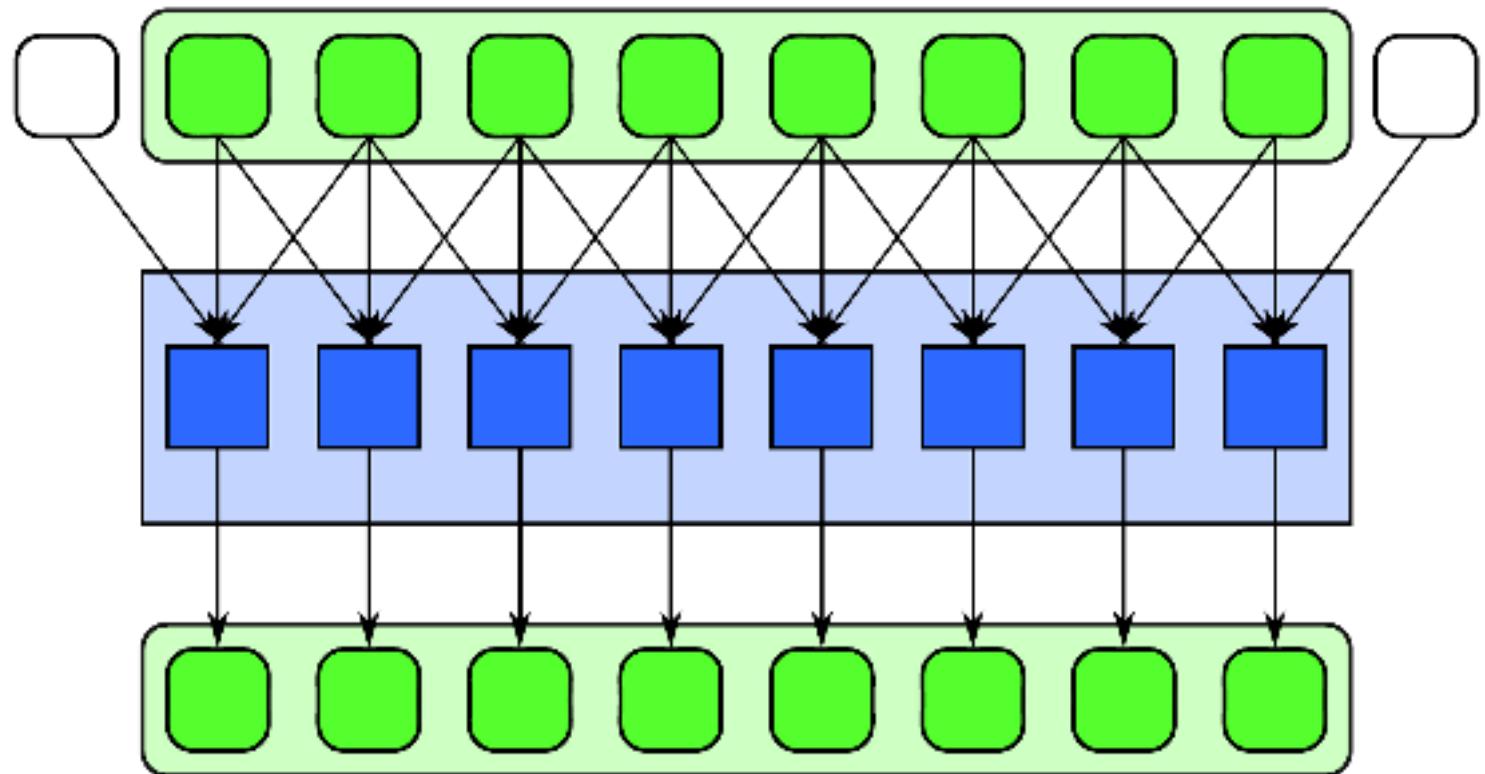
Serial Stencil Example (part 2)

```
15 // array to hold neighbors
16 In neighborhood[NumOff];
17 // loop over all output locations
18 for (int i = 0; i < n; ++i) {
19     // loop over all offsets and gather neighborhood
20     for (int j = 0; j < NumOff; ++j) {
21         // get index of jth input location
22         int k = i+offsets[j];
23         if (0 <= k && k < n) {
24             // read input location
25             neighborhood[j] = a[k];
26         } else {
27             // handle boundary case
28             neighborhood[j] = b;
29         }
30     }
31     // compute output value from input neighborhood
32     a[i] - r[i] = func(neighborhood);
33 }
34 }
```

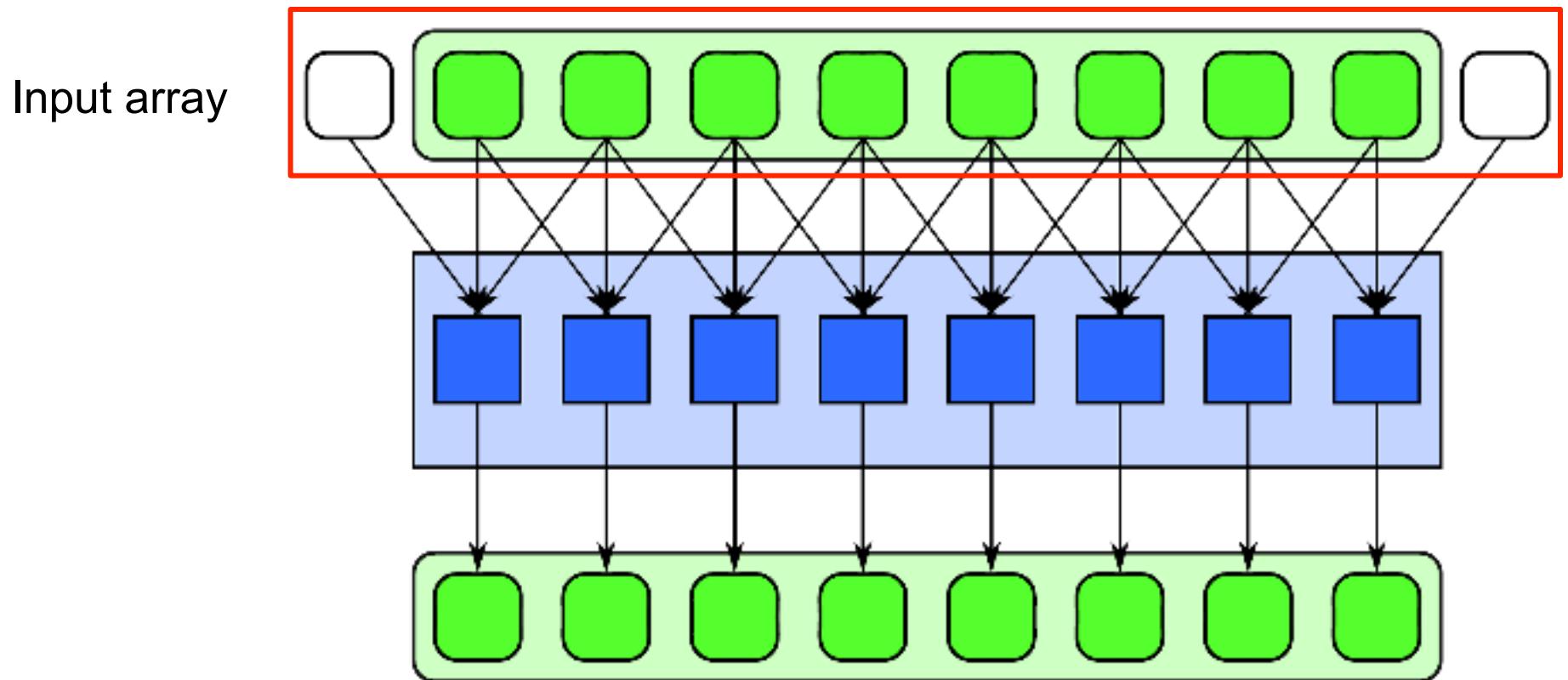
How would we parallelize this?

Updates occur in place!!!

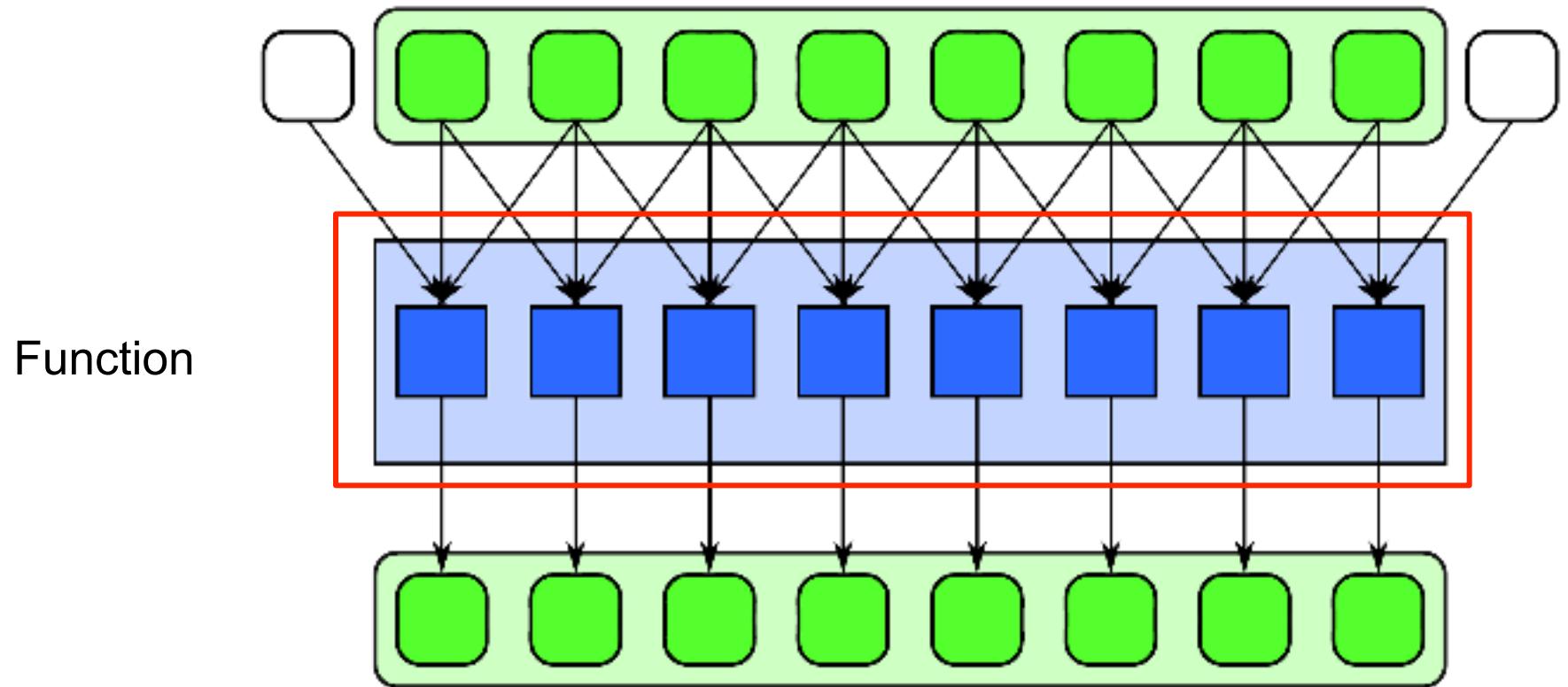
Stencil Pattern with In Place Update



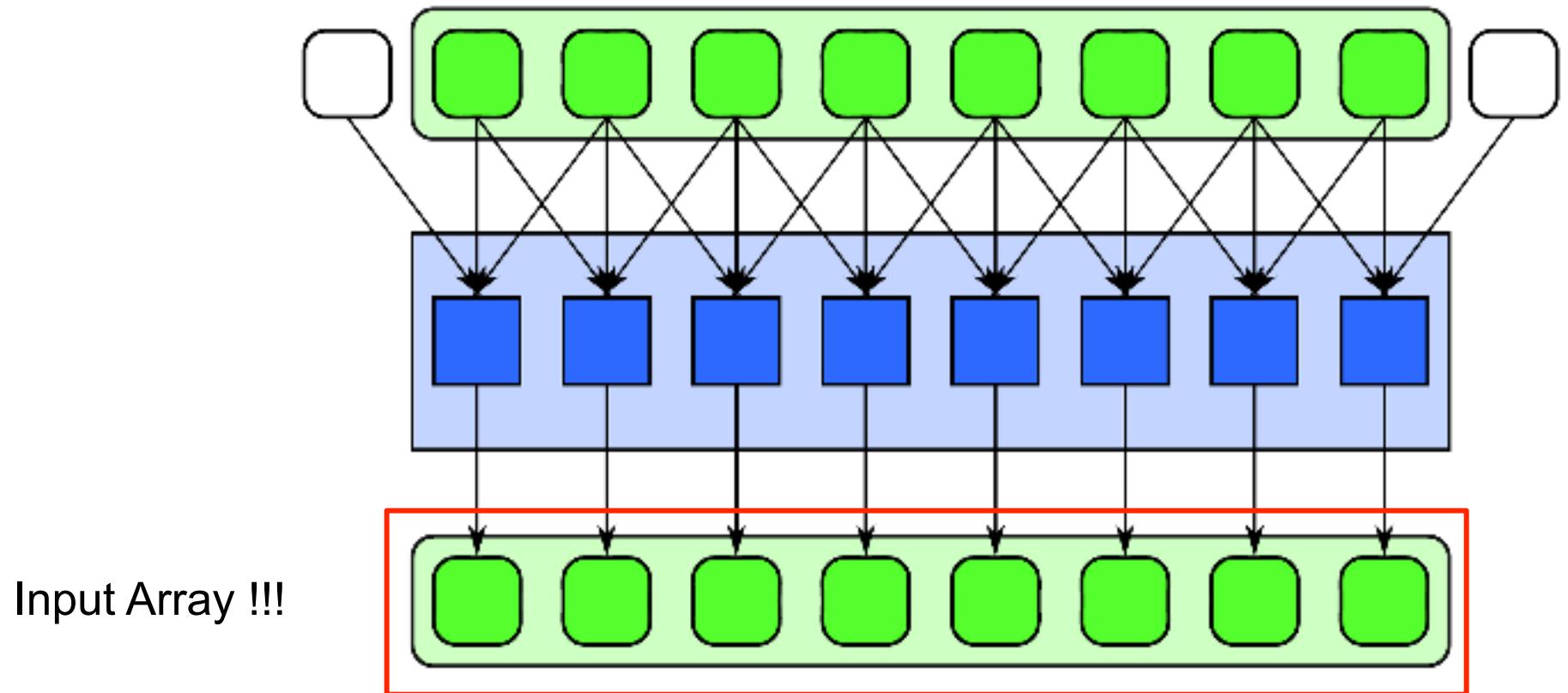
Stencil Pattern with In Place Update



Stencil Pattern with In Place Update

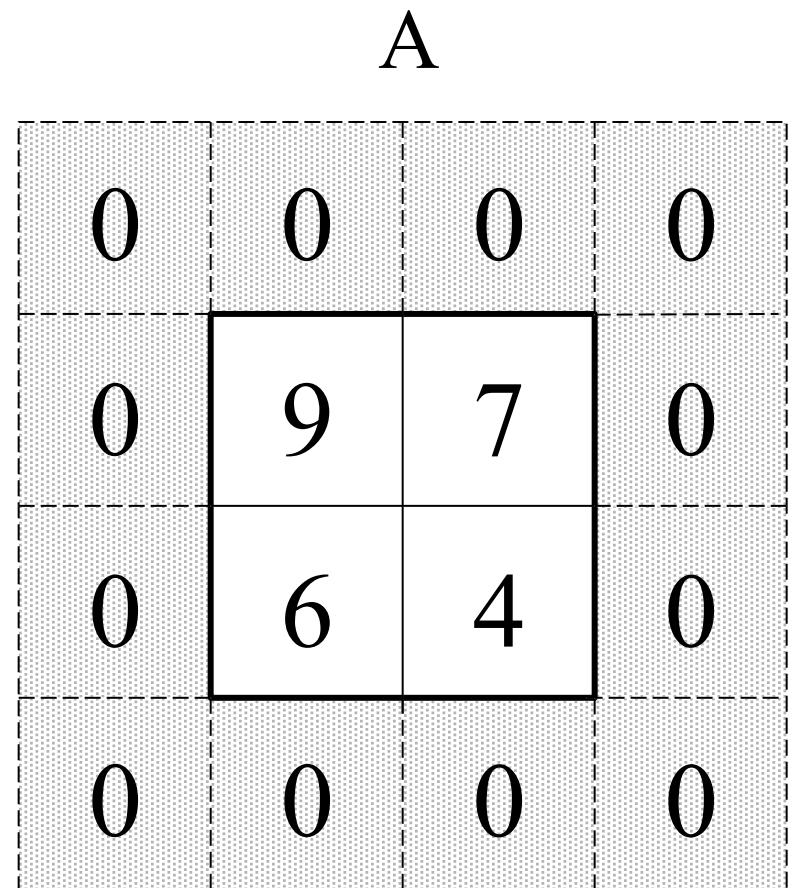


Stencil Pattern with In Place Update



Stencil Example

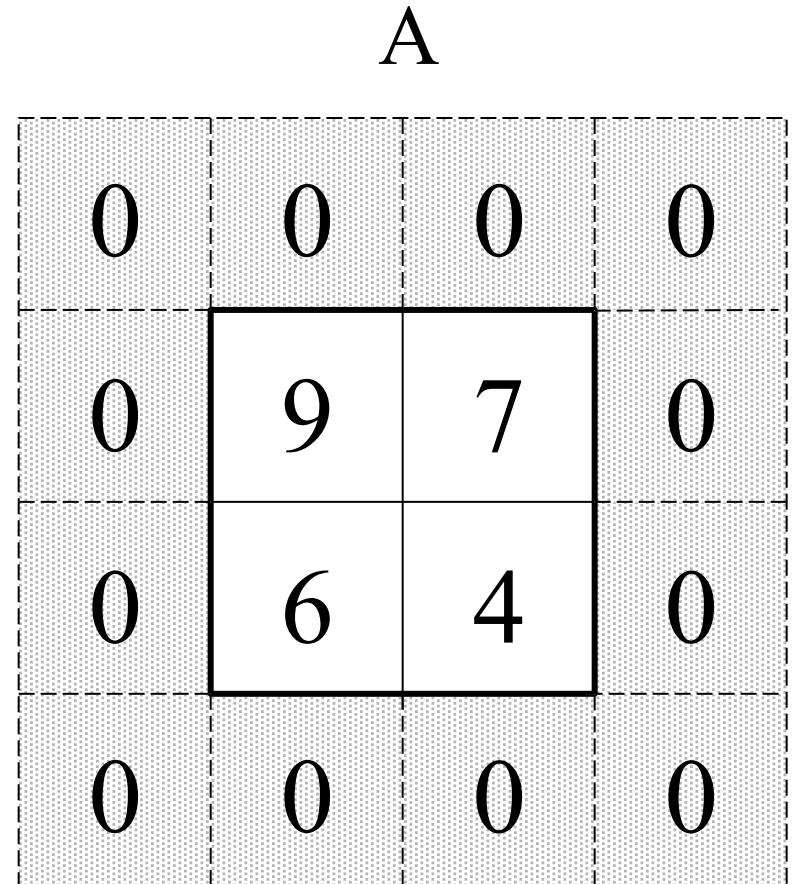
- Here is our array, A



Stencil Example

- Here is our array A
- Update A in place
- Apply a stencil operation to the inner square of the form:

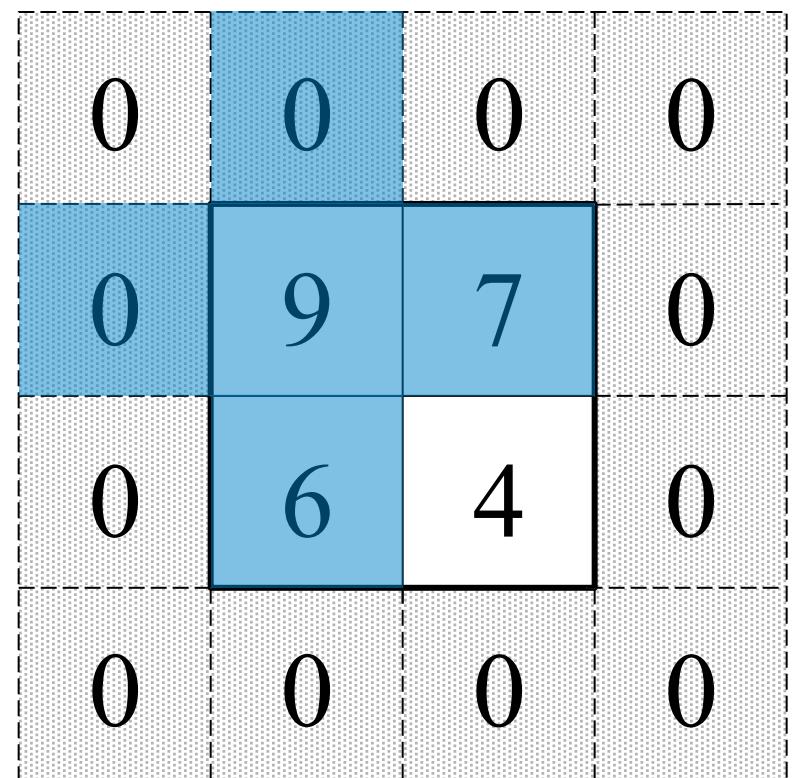
$$A(i,j) = \text{avg}(\ A(i,j), \\ A(i-1,j), A(i+1,j), \\ A(i,j-1), A(i,j+1) \\)$$



What is the stencil?

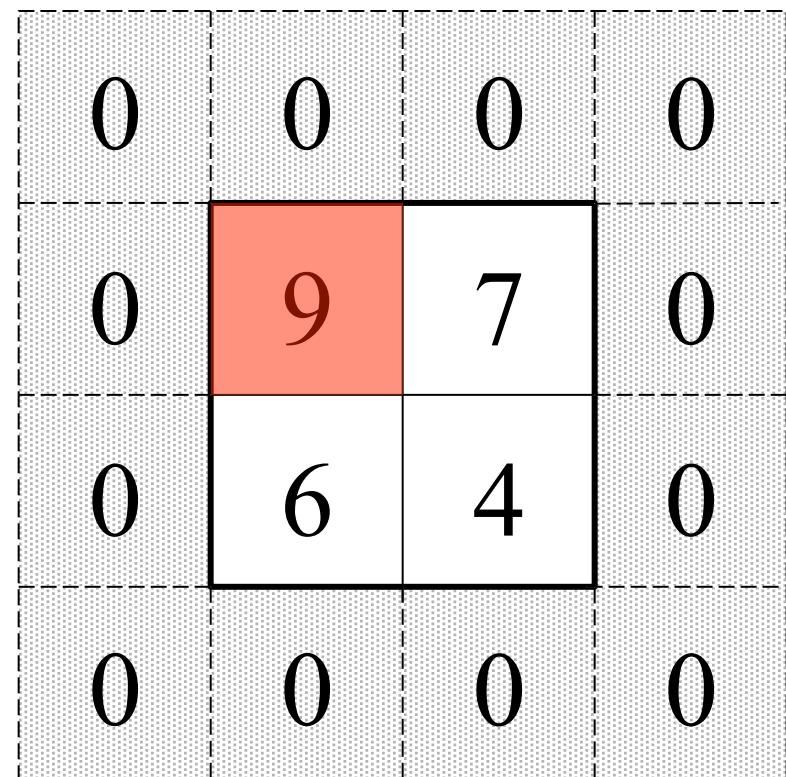
Stencil Pattern Procedure

- 1) Average all blue squares



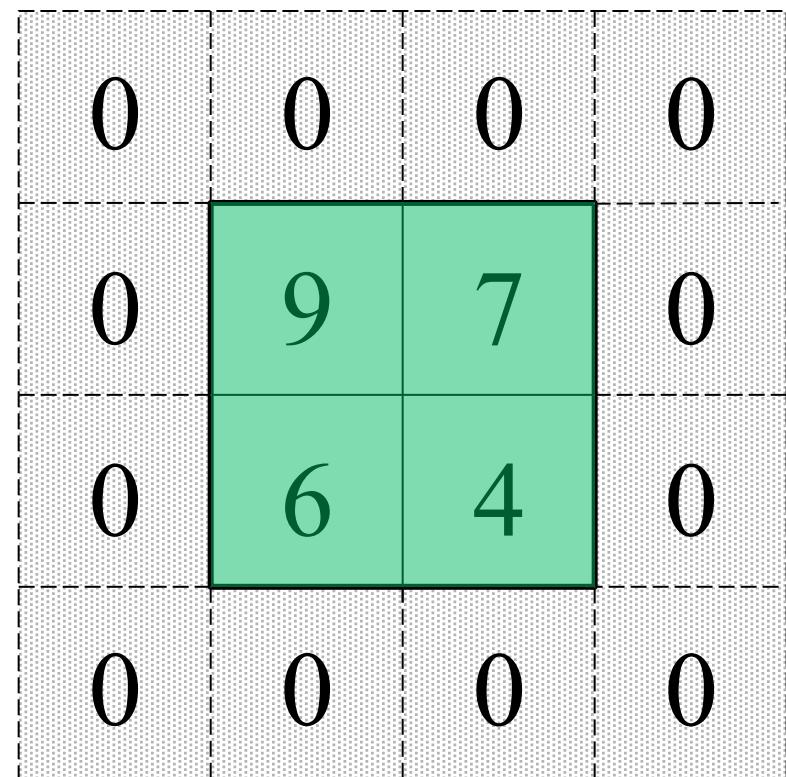
Stencil Pattern Procedure

- 1) Average all blue squares
- 2) Store result in red square



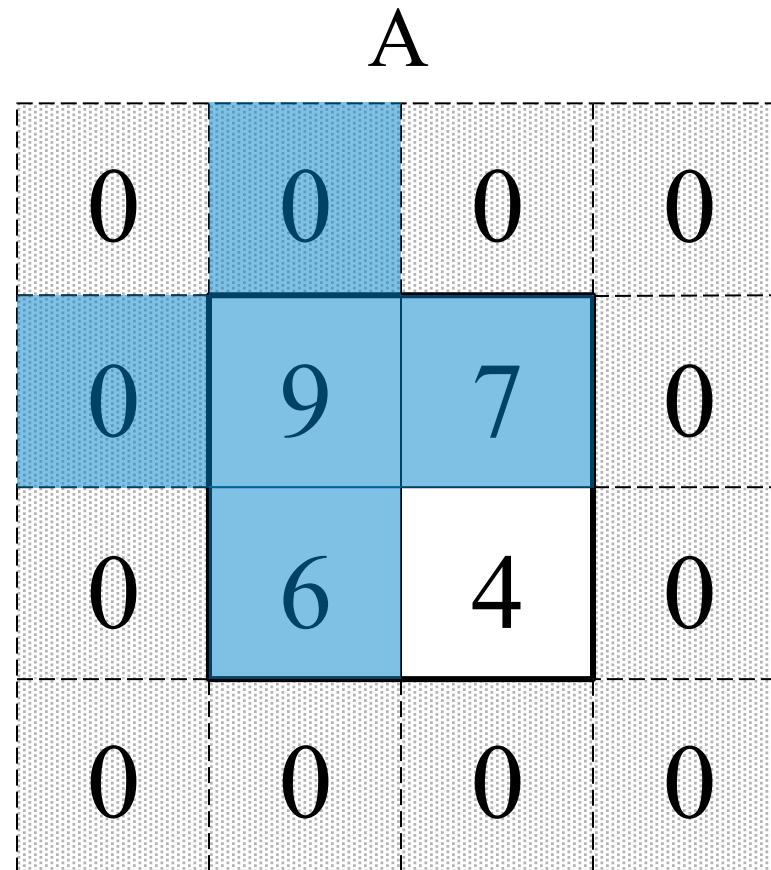
Stencil Pattern Procedure

- 1) Average all blue squares
- 2) Store result in red square
- 3) Repeat 1 and 2 for all green squares

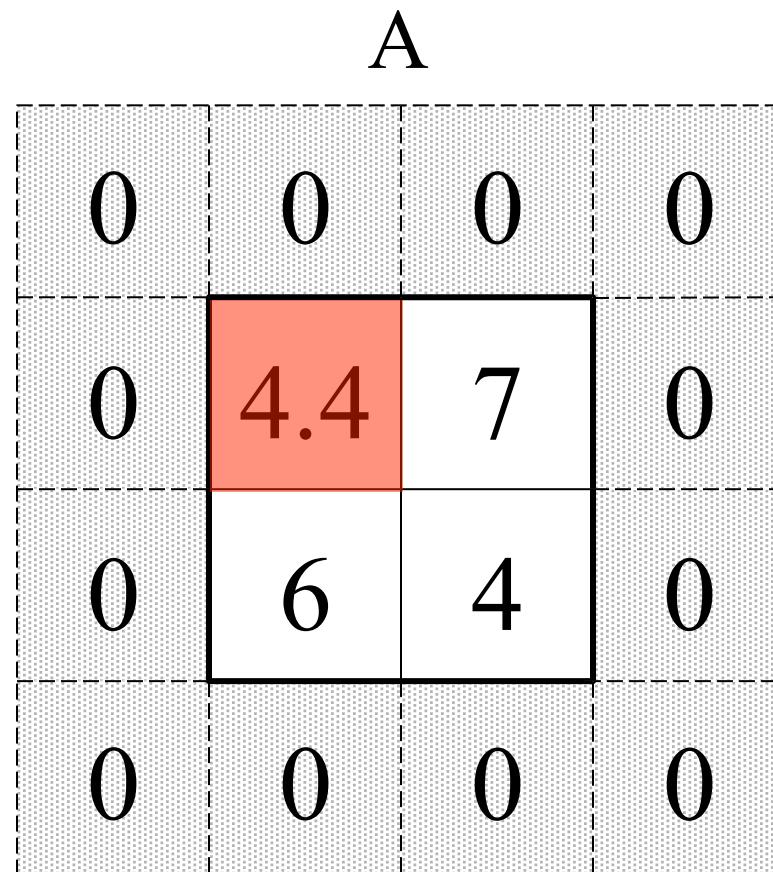


Practice!

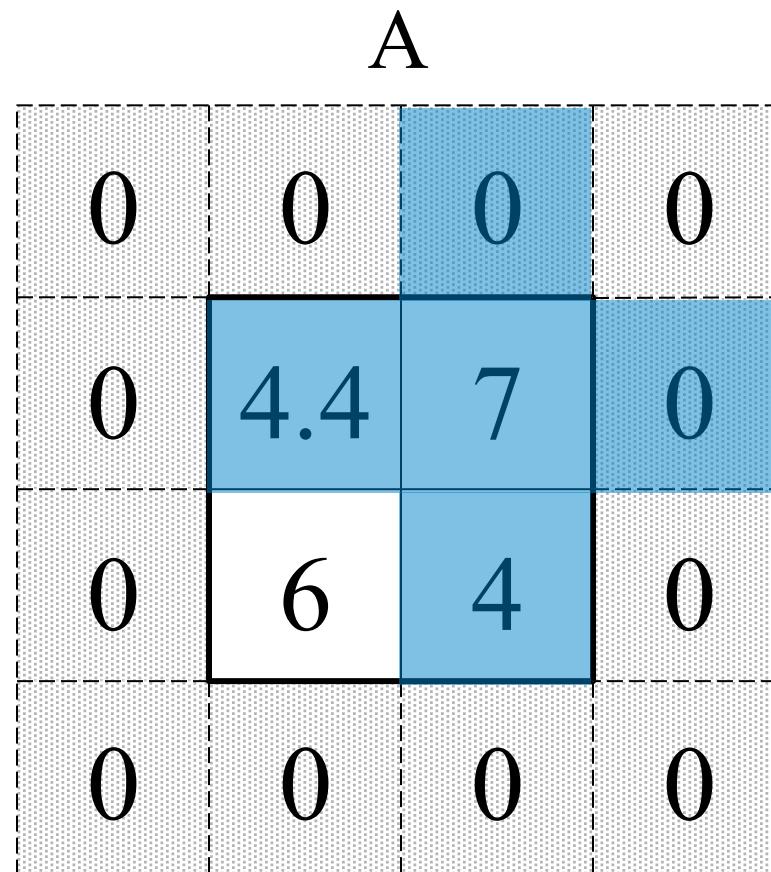
Stencil Pattern Practice



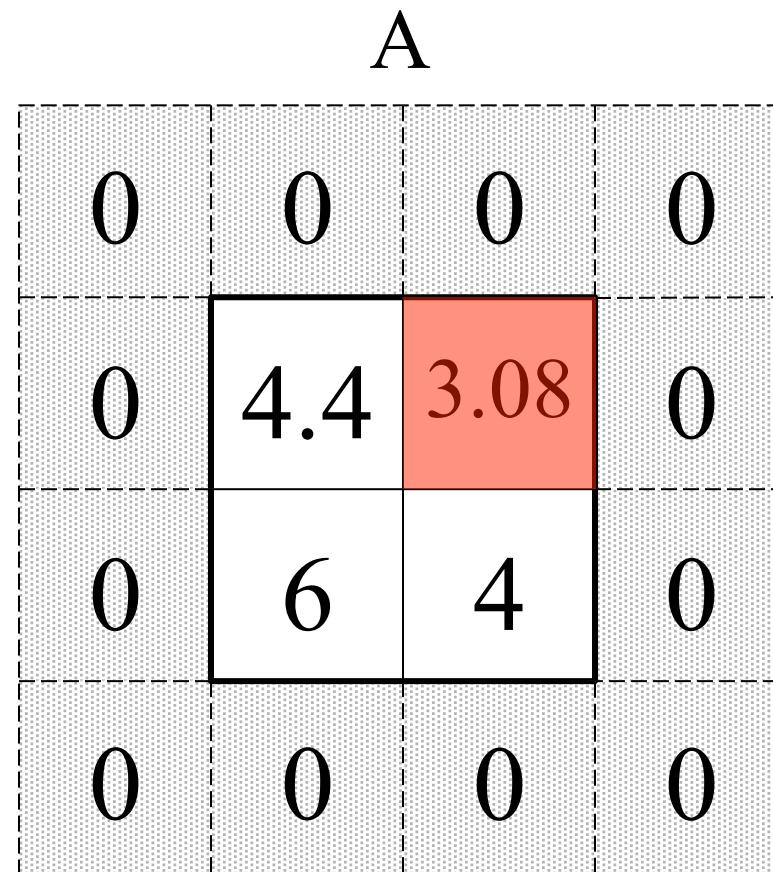
Stencil Pattern Practice



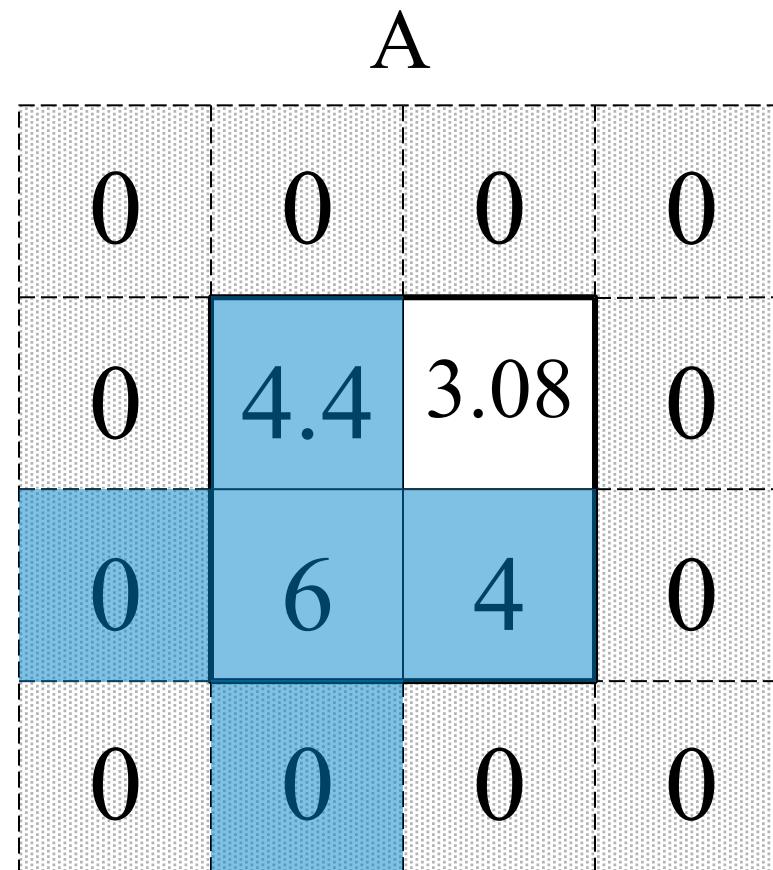
What is the stencil pattern?



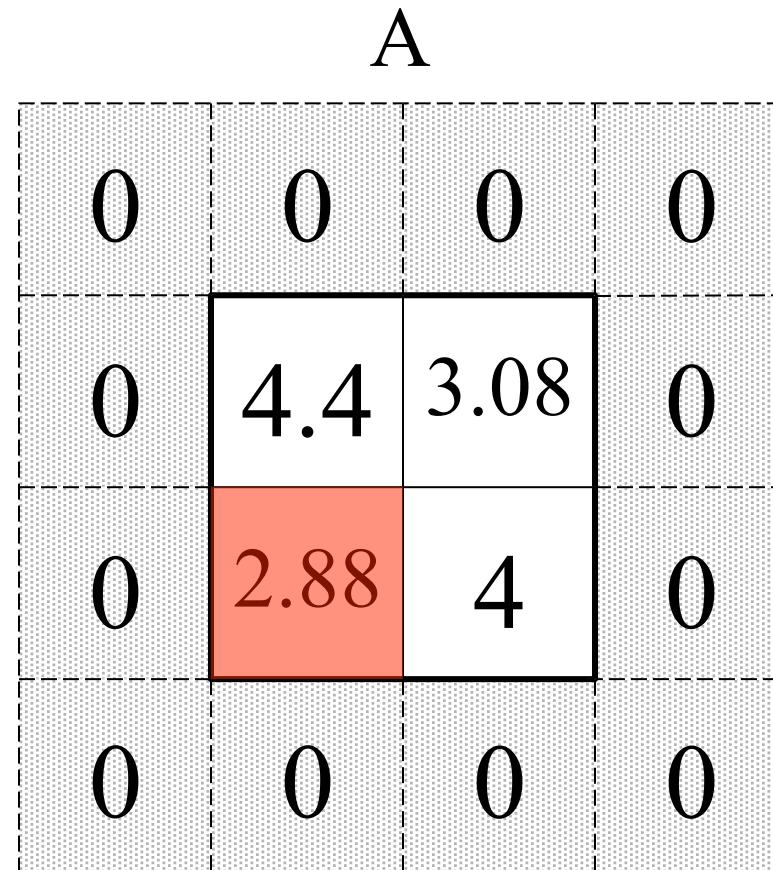
What is the stencil pattern?



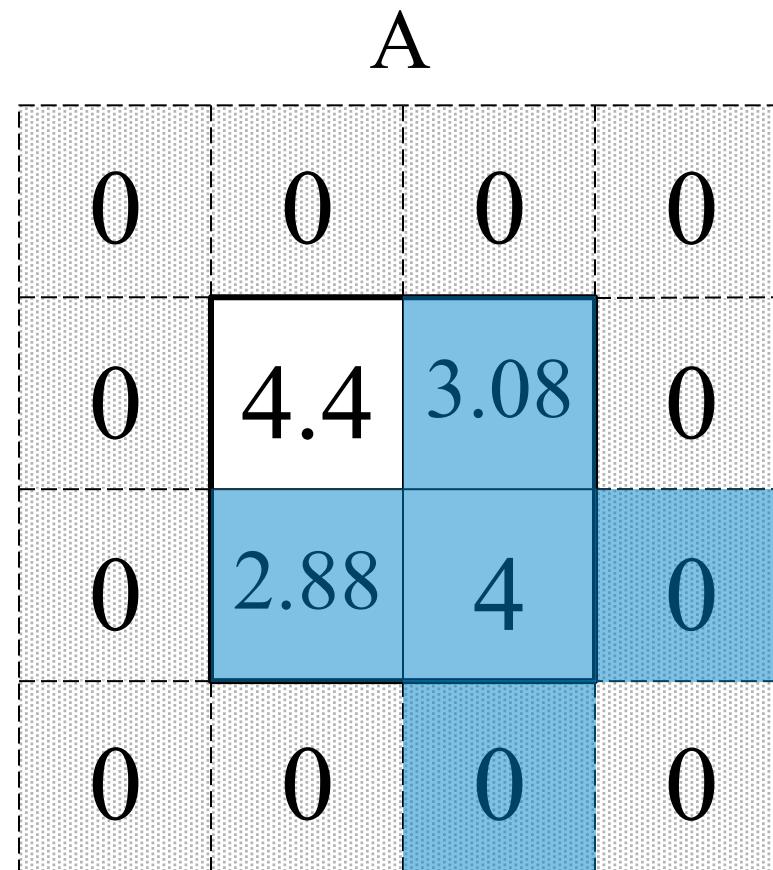
What is the stencil pattern?



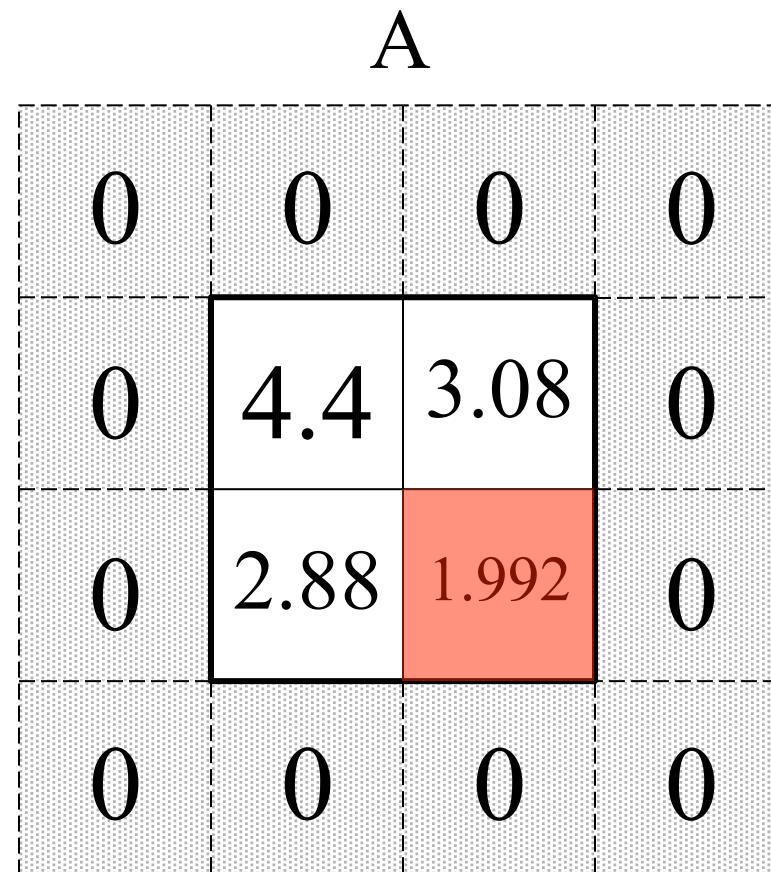
What is the stencil pattern?



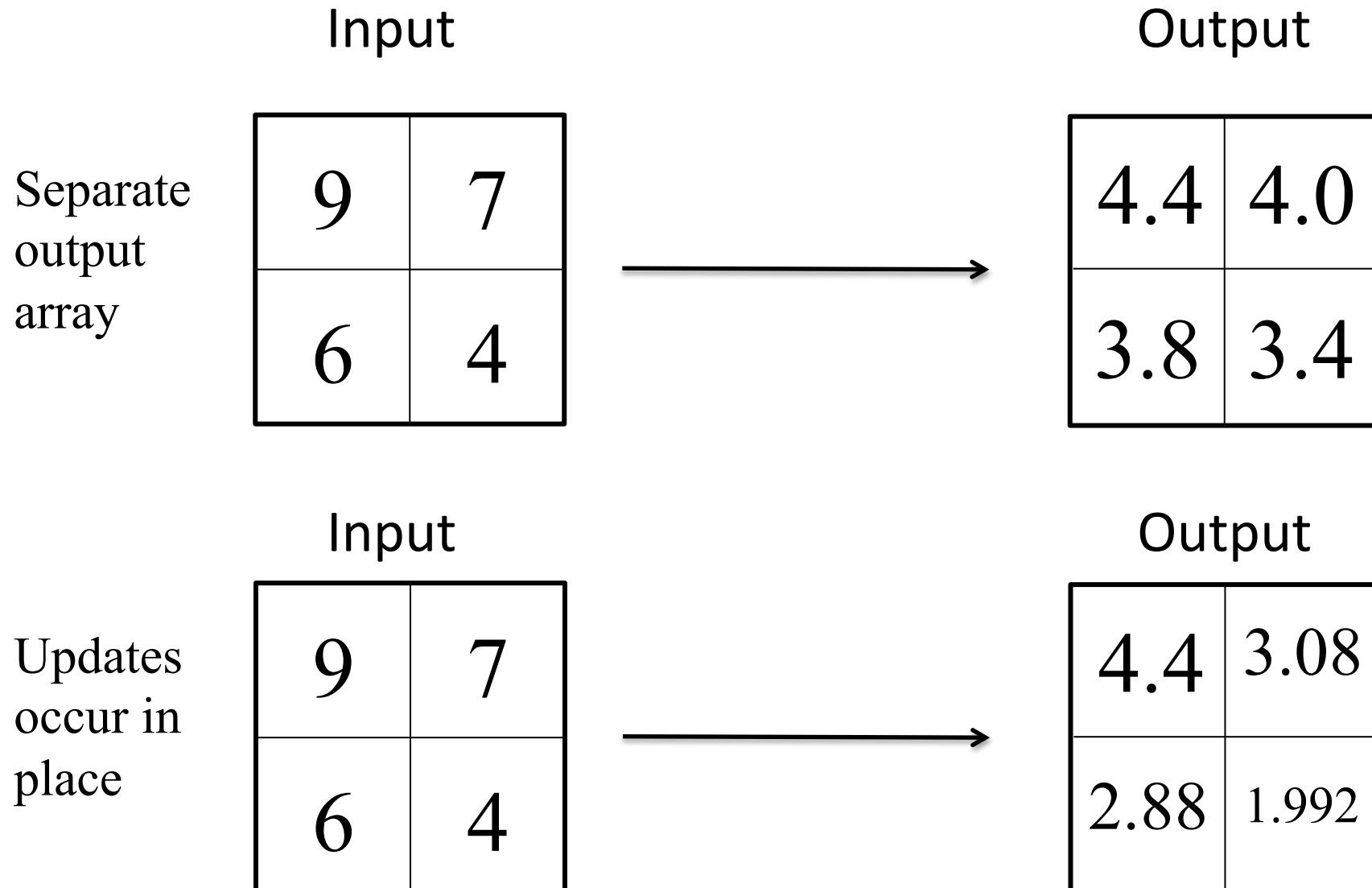
What is the stencil pattern?



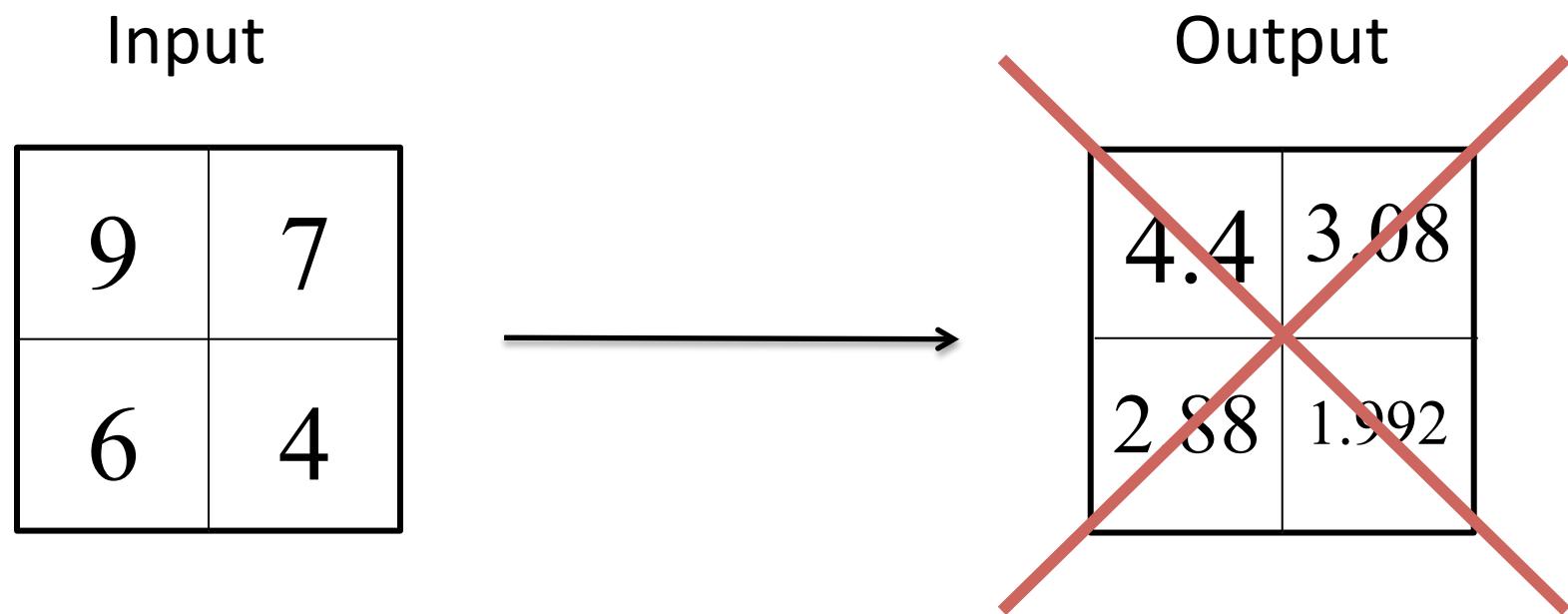
What is the stencil pattern?



Different Cases



Which is correct?



Is this output incorrect?

Outline

- Partitioning
- What is the stencil pattern?
 - Update alternatives
 - 2D Jacobi iteration
 - SOR and Red/Black SOR
- Implementing stencil with shift
- Stencil and cache optimizations
- Stencil and communication optimizations
- Recurrence

Iterative Codes

- Iterative codes are ones that update their data in steps
 - At each step, a new value of an element is computed using a formula based on other elements
 - Once all elements are updated, the computation proceeds to the next step or completes
- Iterative codes are most commonly found in computer simulations of physical systems for scientific and engineering applications
 - Computational fluid dynamics
 - Electromagnetics modeling
- They are often applied to solve partial differential equations
 - Jacobi iteration
 - Gauss-Seidel iteration
 - Successive over relaxation

Iterative Codes and Stencils

- Stencils essentially define which elements are used in the update formula
- Because the data is organized in a regular manner, stencils can be applied across the data uniformly

Simple 2D Example

- Consider the following code

```
for k=1, 1000
```

```
    for i=1, N-2
```

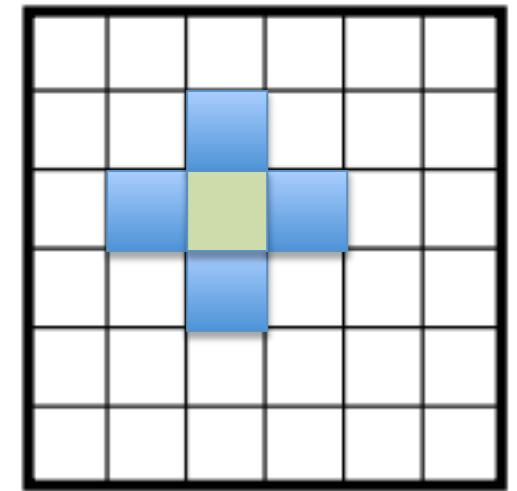
```
        for j = 1, N-2
```

```
            a[i][j] = 0.25 * (a[i][j] + a[i-1][j]  
                                + a[i+1][j]  
                                + a[i][j-1]  
                                + a[i][j+1])
```

```
}
```

```
}
```

```
}
```



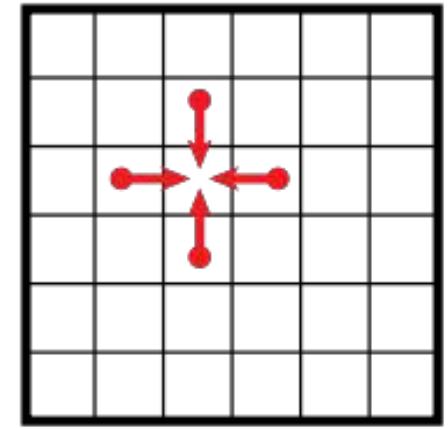
5-point stencil

Do you see anything interesting?

How would you parallelize?

2-Dimension Jacobi Iteration

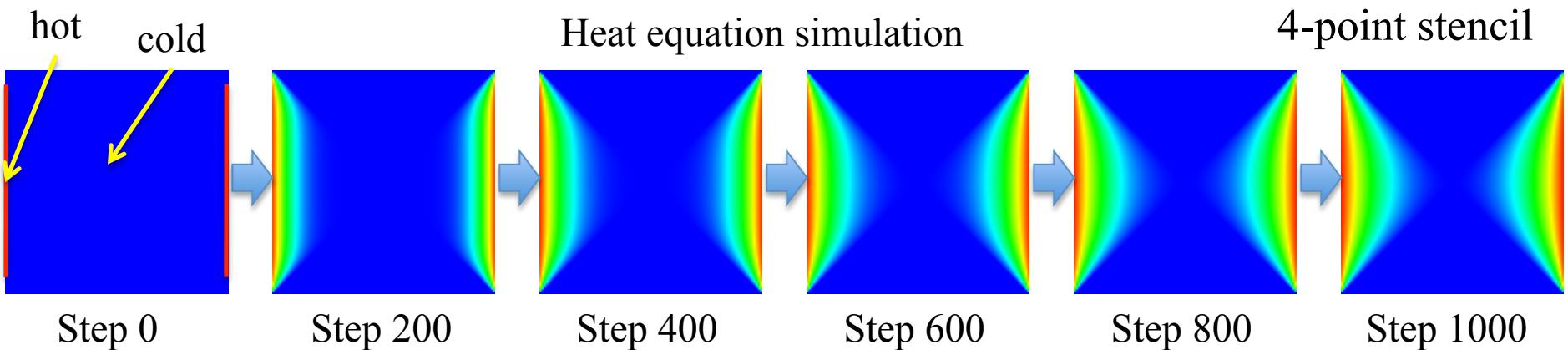
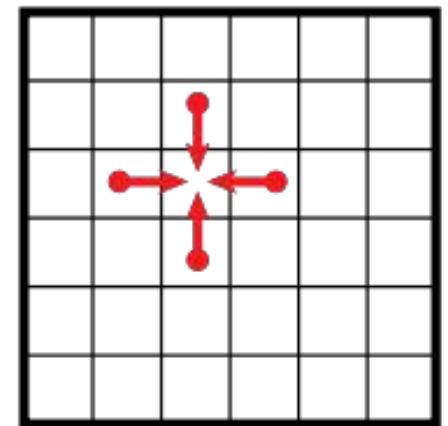
- Consider a 2D array of elements
- Initialize each array element to some value
- At each step, update each array element to the arithmetic mean of its N, S, E, W neighbors
- Iterate until array values converge
- Here we are using a 4-point stencil
- It is different from before because we want to update all array elements simultaneously ... How?



4-point stencil

2-Dimension Jacobi Iteration

- Consider a 2D array of elements
- Initialize each array element to some value
- At each step, update each array element to the arithmetic mean of its N, S, E, W neighbors
- Iterate until array values converge

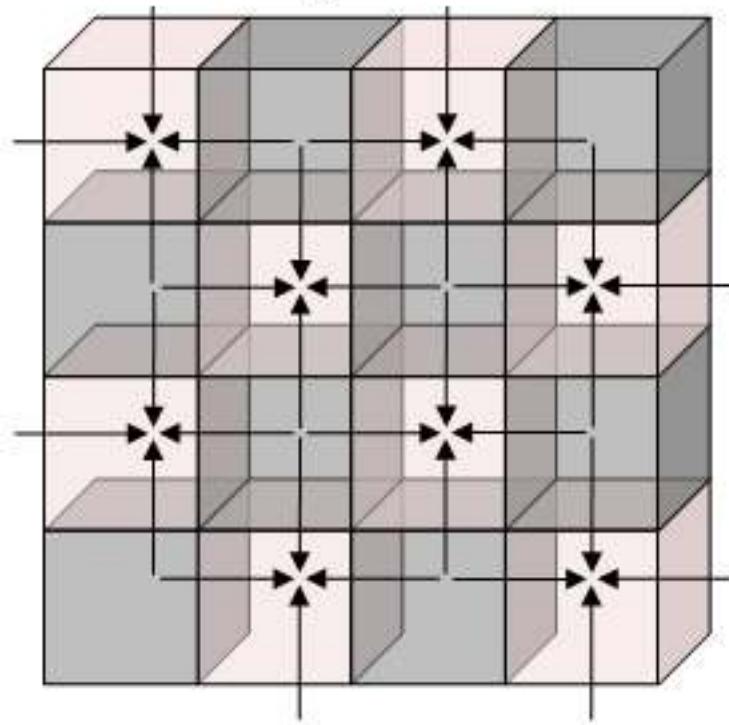


Successive Over Relaxation (SOR)

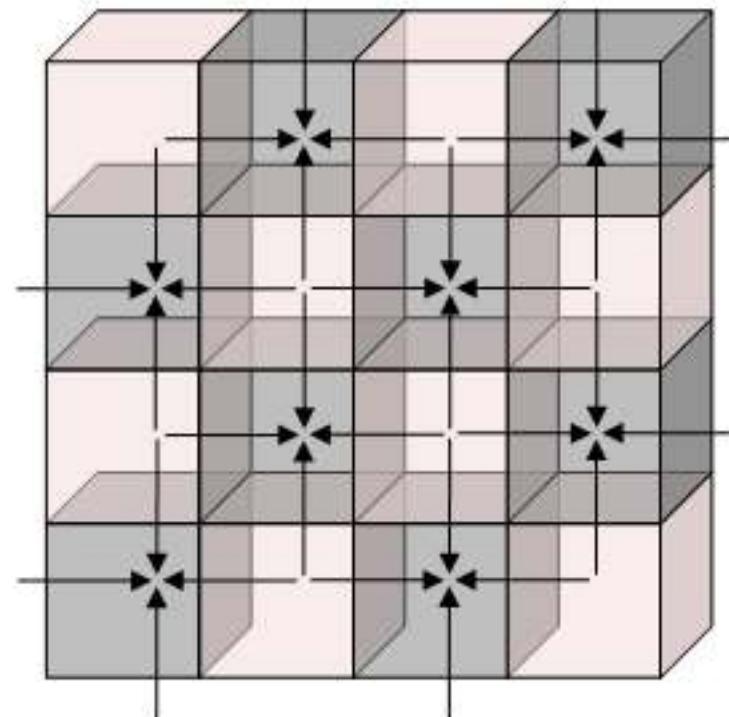
- SOR is an alternate method of solving partial differential equations
- While the Jacobi iteration scheme is very simple and parallelizable, its slow convergent rate renders it impractical for any "real world" applications
- One way to speed up the convergent rate would be to "over predict" the new solution by linear extrapolation
- It also allows a method known as Red-Black SOR to be used to enable parallel updates in place

Red / Black SOR

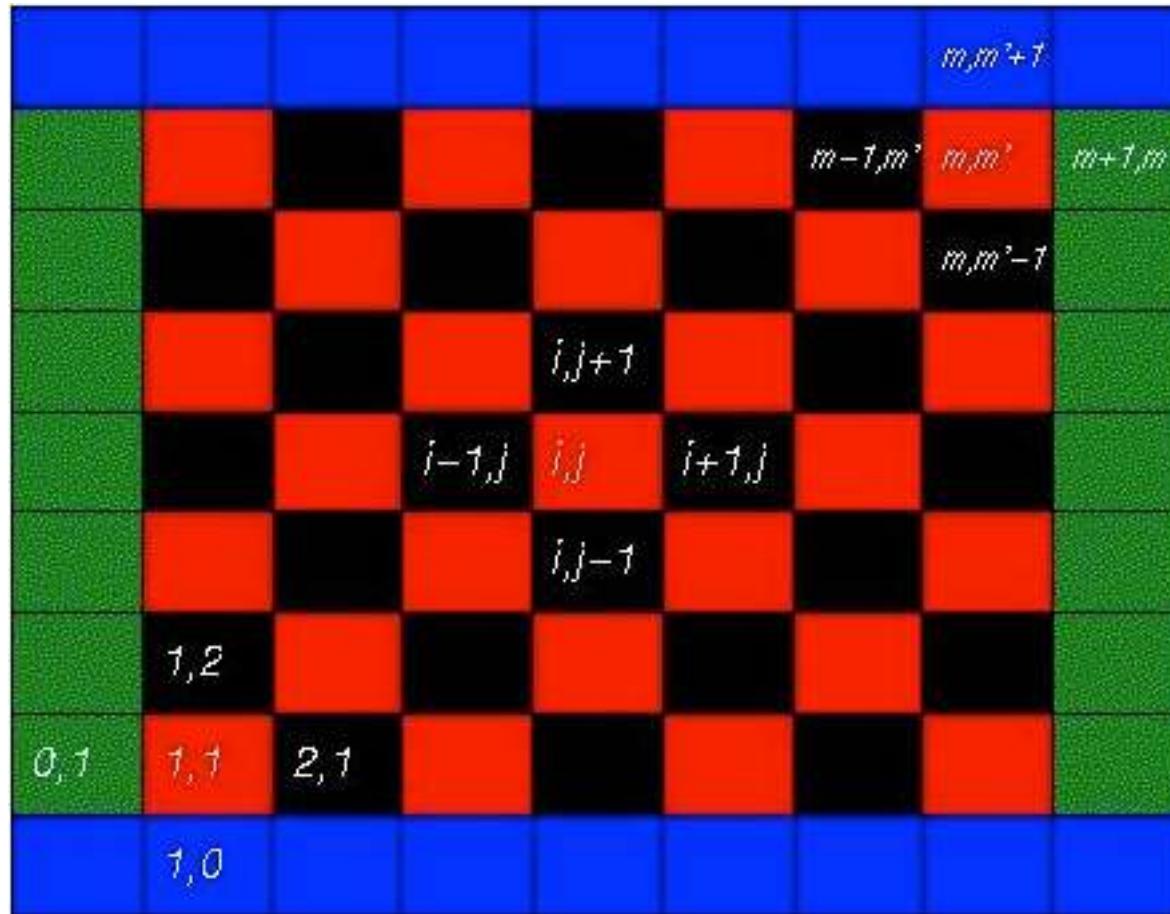
Pass 1: Writing to red cells,
reading from black



Pass 2: Writing to black cells,
reading from red



Red / Black SOR



Outline

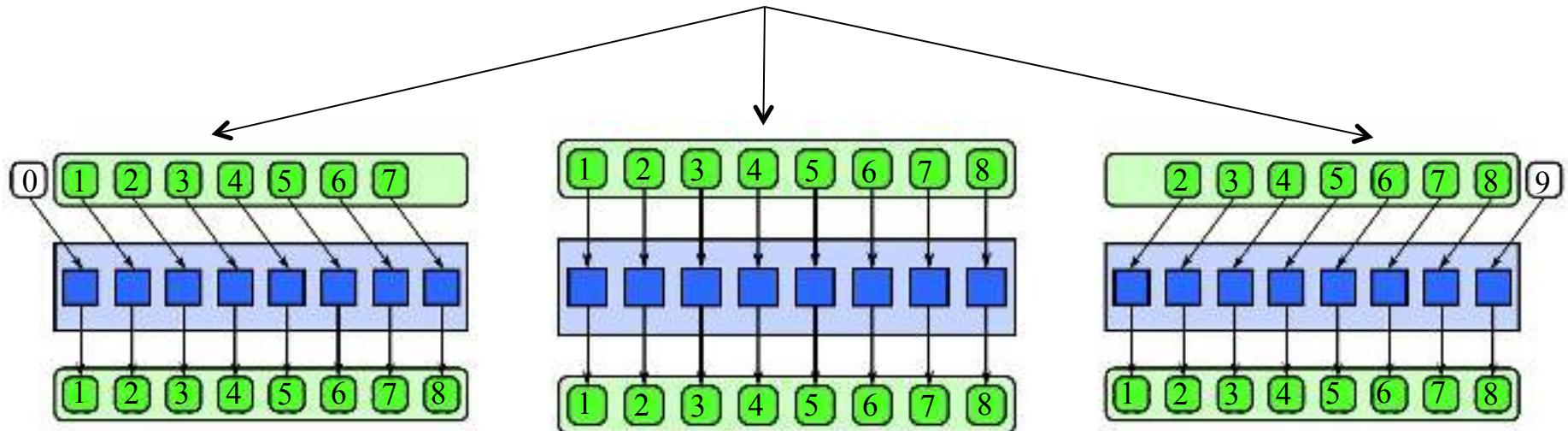
- Partitioning
- What is the stencil pattern?
 - Update alternatives
 - 2D Jacobi iteration
 - SOR and Red/Black SOR
- Implementing stencil with shift
- Stencil and cache optimizations
- Stencil and communication optimizations
- Recurrence

Implementing Stencil with Shift

- One possible implementation of the stencil pattern includes shifting the input data
- For each offset in the stencil, we gather a new input vector by **shifting** the original input by the offset amount

Implementing Stencil with Shift

All input arrays are derived from the same original input array



Implementing Stencil with Shift

- This implementation is only beneficial for one dimensional stencils or the memory-contiguous dimension of a multidimensional stencil
- Memory traffic to external memory is not reduced with shifts
- But, shifts allow vectorization of the data reads, which may reduce the total number of instructions

Contents

- Partitioning
- What is the stencil pattern?
 - Update alternatives
 - 2D Jacobi iteration
 - SOR and Red/Black SOR
- Implementing stencil with shift
- Stencil and cache optimizations
- Stencil and communication optimizations
- Recurrence

Stencil and Cache Optimizations

- Assuming 2D array where rows are contiguous in memory...
 - Horizontally related data will tend to belong to the same cache line
 - Vertical offset accesses will most likely result in cache misses

Stencil and Cache Optimizations

- Assigning rows to cores:
 - Maximizes horizontal data locality
 - Assuming vertical offsets in stencil, this will create redundant reads of adjacent rows from each core
- Assigning columns to cores:
 - Redundantly read data from same cache line
 - Create false sharing as cores write to same cache line

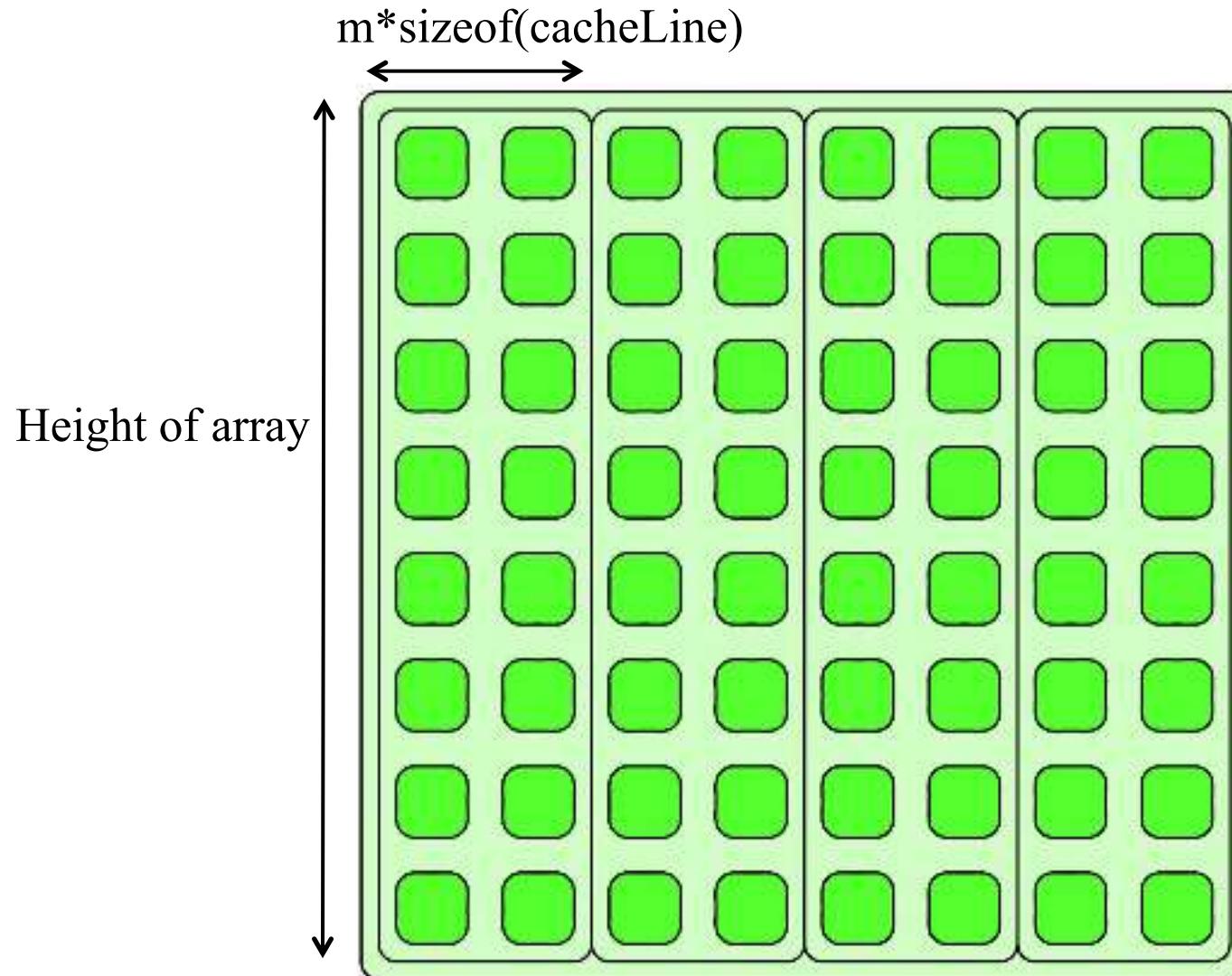
Stencil and Cache Optimizations

- Assigning “strips” to each core can be a better solution
- **Strip-mining:** an optimization in a stencil computation that groups elements in a way that avoids redundant memory accesses and aligns memory accesses with cache lines

Stencil and Cache Optimizations

- A strip's size is a multiple of a cache line in width, and the height of the 2D array
- Strip widths are in increments of the cache line size so as to avoid false sharing and redundant reads
- Each strip is processed serially from top to bottom within each core

Stencil and Cache Optimizations

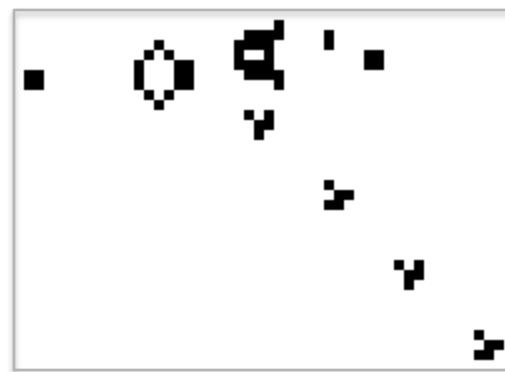


Outline

- Partitioning
- What is the stencil pattern?
 - Update alternatives
 - 2D Jacobi iteration
 - SOR and Red/Black SOR
- Implementing stencil with shift
- Stencil and cache optimizations
- **Stencil and communication optimizations**
- Recurrence

But first... Conway's Game of Life

- The **Game of Life** is a cellular automaton created by John Conway in 1970
- The evolution of the game is entirely based on the input state – zero player game
- To play: create initial state, observe how the system evolves over successive time steps



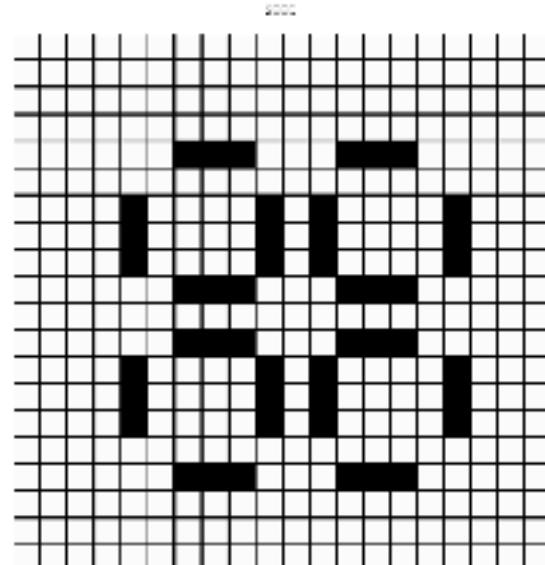
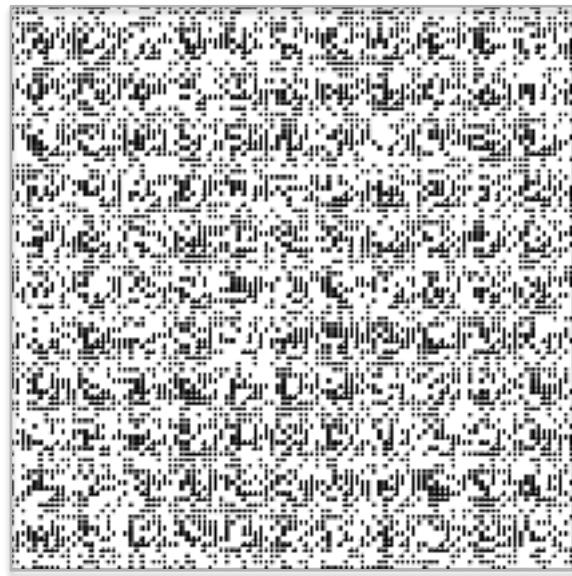
2D landscape

Conway's Game of Life

- Typical rules for the Game of Life
 - Infinite 2D grid of square cells, each cell is either “alive” or “dead”
 - Each cell will interact with all 8 of its neighbors
 - ◆ Any cell with < 2 live neighbors dies (under-population)
 - ◆ Any cell with 2 or 3 live neighbors lives to next gen.
 - ◆ Any cell with > 3 live neighbors dies (overcrowding)
 - ◆ Any dead cell with 3 live neighbors becomes a live cell

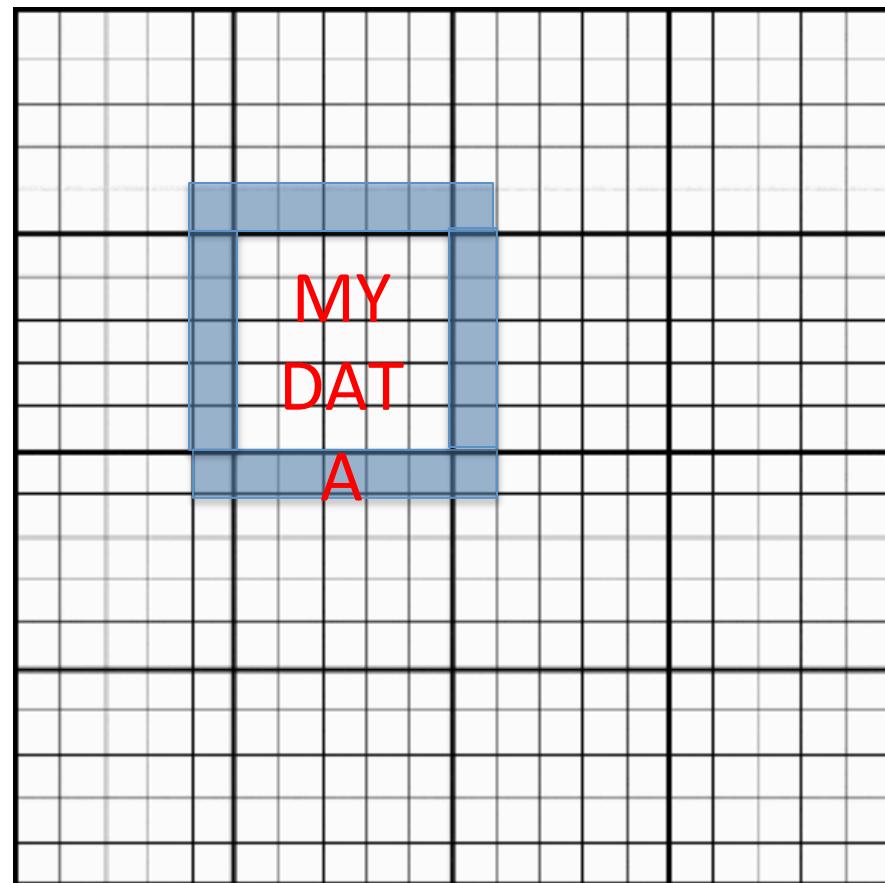


Conway's Game of Life: Examples



Conway's Game of Life

- The Game of Life computation can easily fit into the stencil pattern!
- Each larger, black box is owned by a thread
- What will happen at the boundaries?

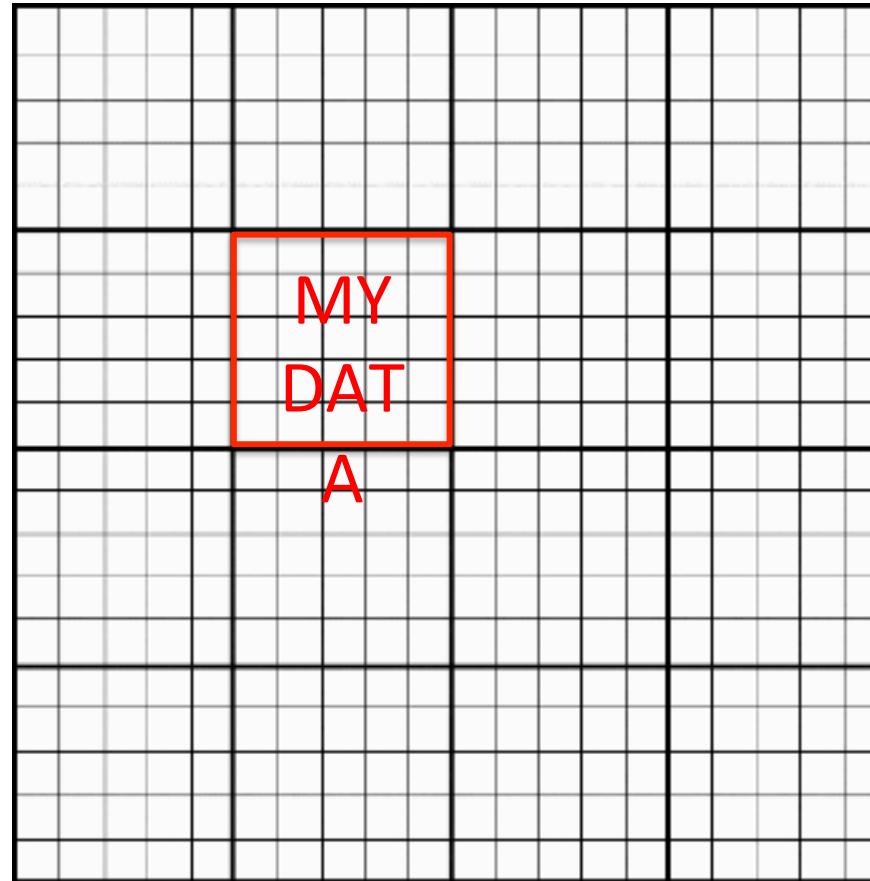


Conway's Game of Life

- We need some way to preserve information from the previous iteration without overwriting it
- **Ghost Cells** are one solution to the boundary and update issues of a stencil computation
- Each thread keeps a copy of neighbors' data to use in its local computations
- These ghost cells must be updated after each iteration of the stencil

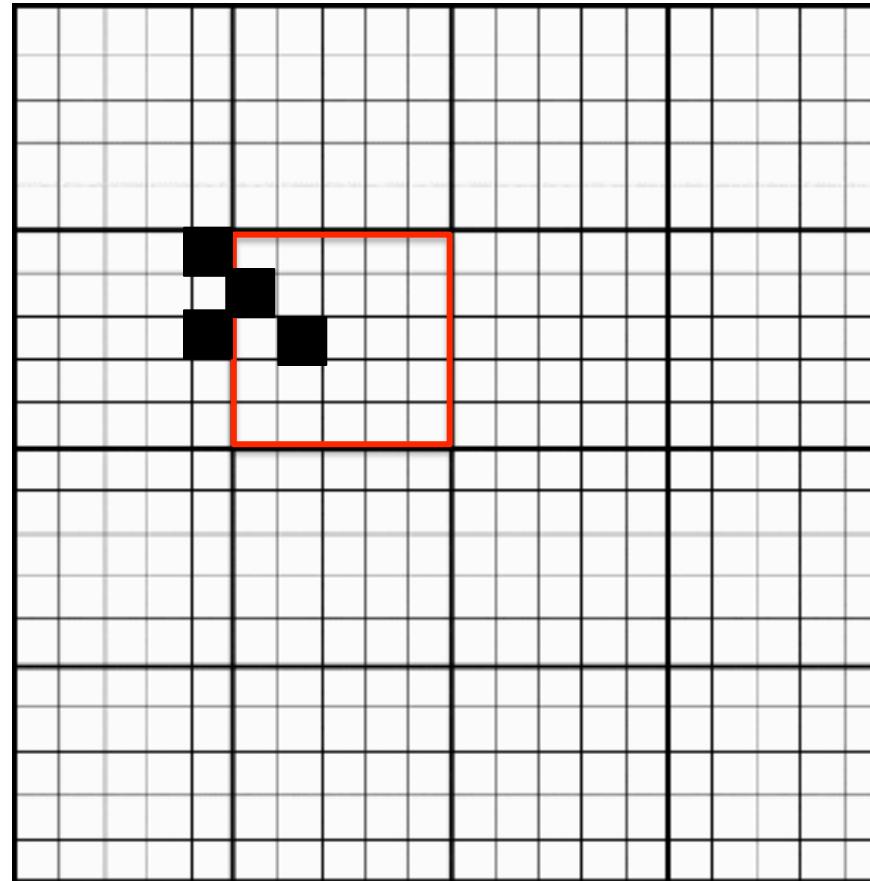
Conway's Game of Life

- Working with ghost cells



Conway's Game of Life

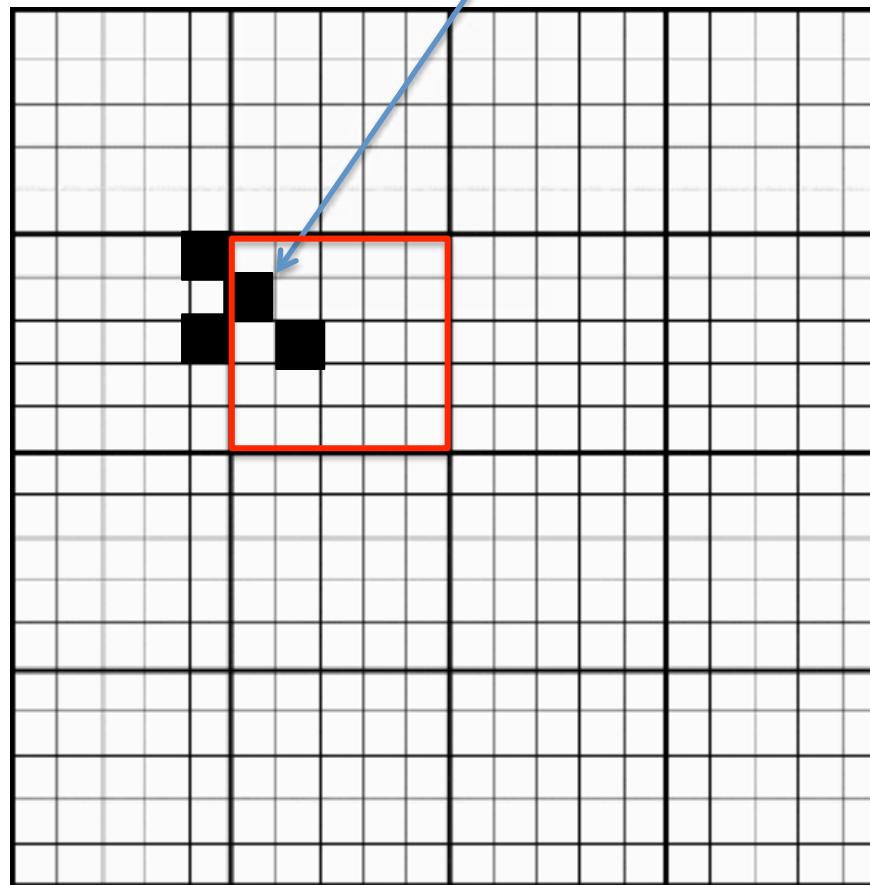
- Working with ghost cells



Conway's Game of Life

- Working with ghost cells

Compute the new value for this cell

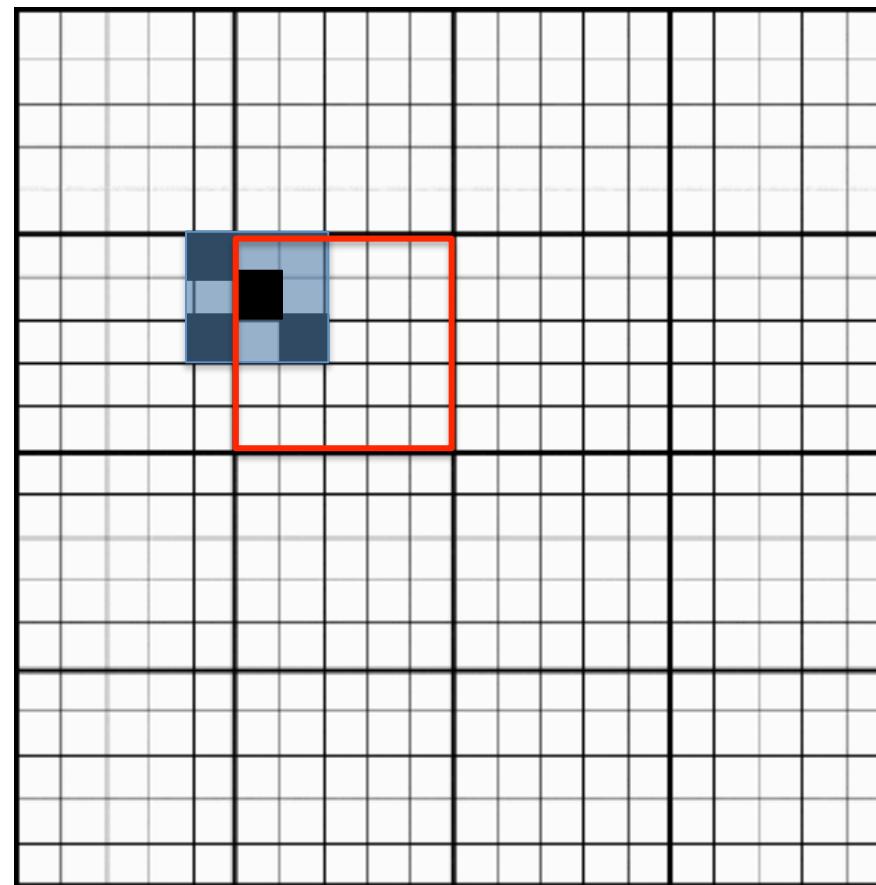


Conway's Game of Life

- Working with ghost cells

Five of its eight
neighbors already
belong to this thread

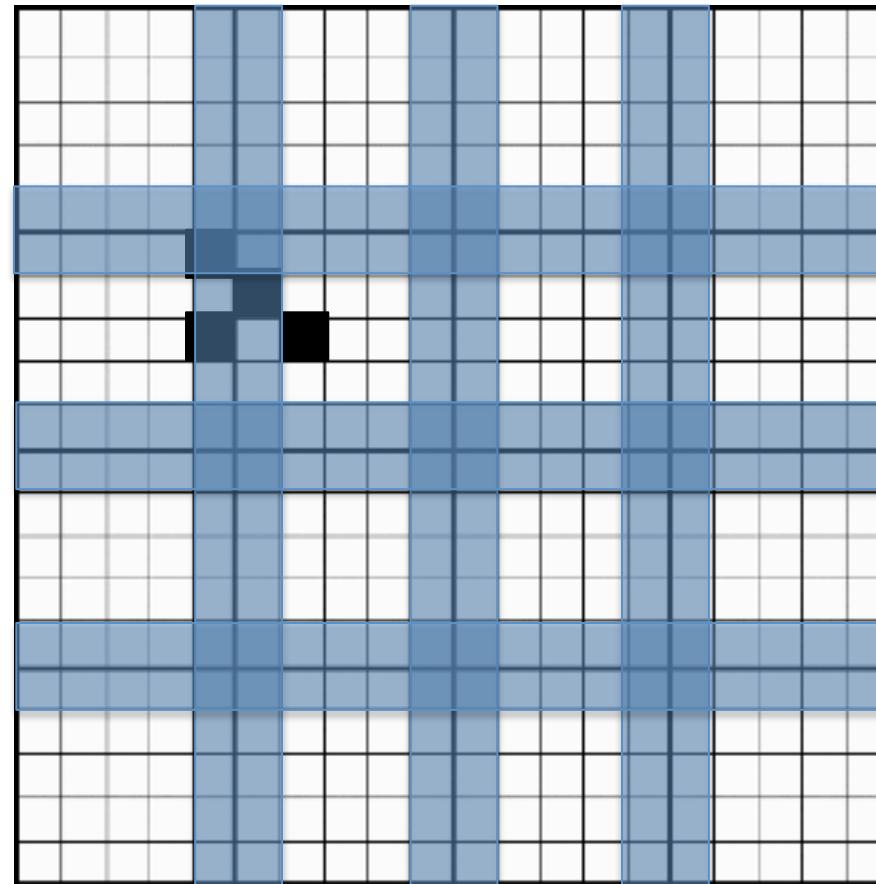
But three of its
neighbors belong to
a different thread



Conway's Game of Life

- Working with ghost cells

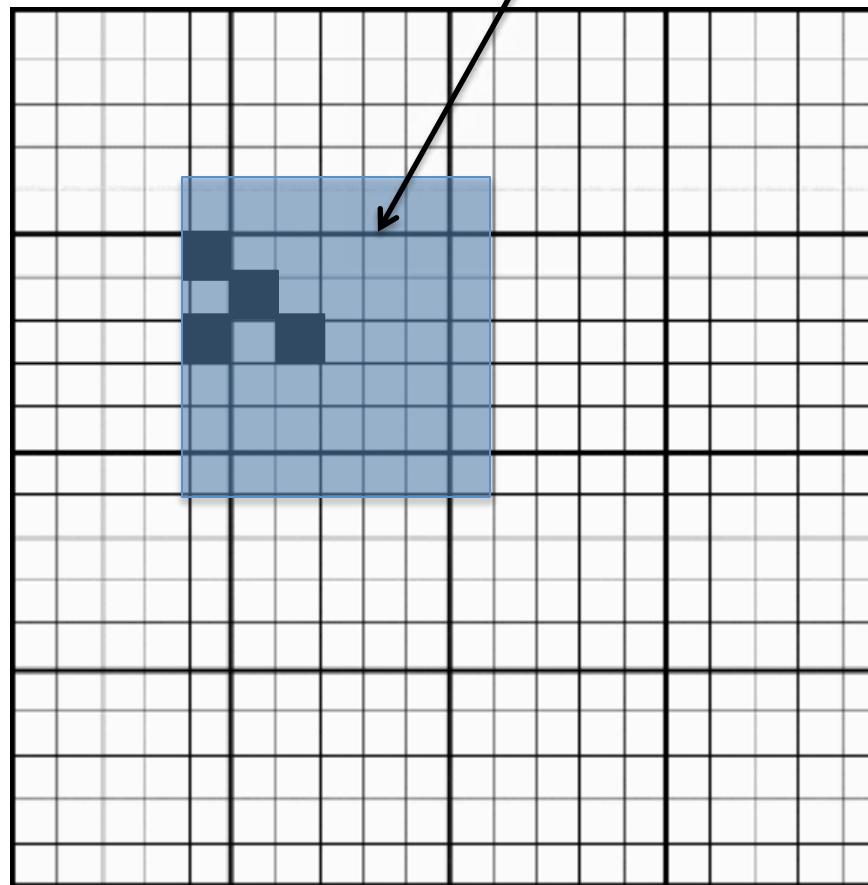
Before any updates are done in a new iteration, all threads must update their ghost cells



Conway's Game of Life

□ Working with ghost cells

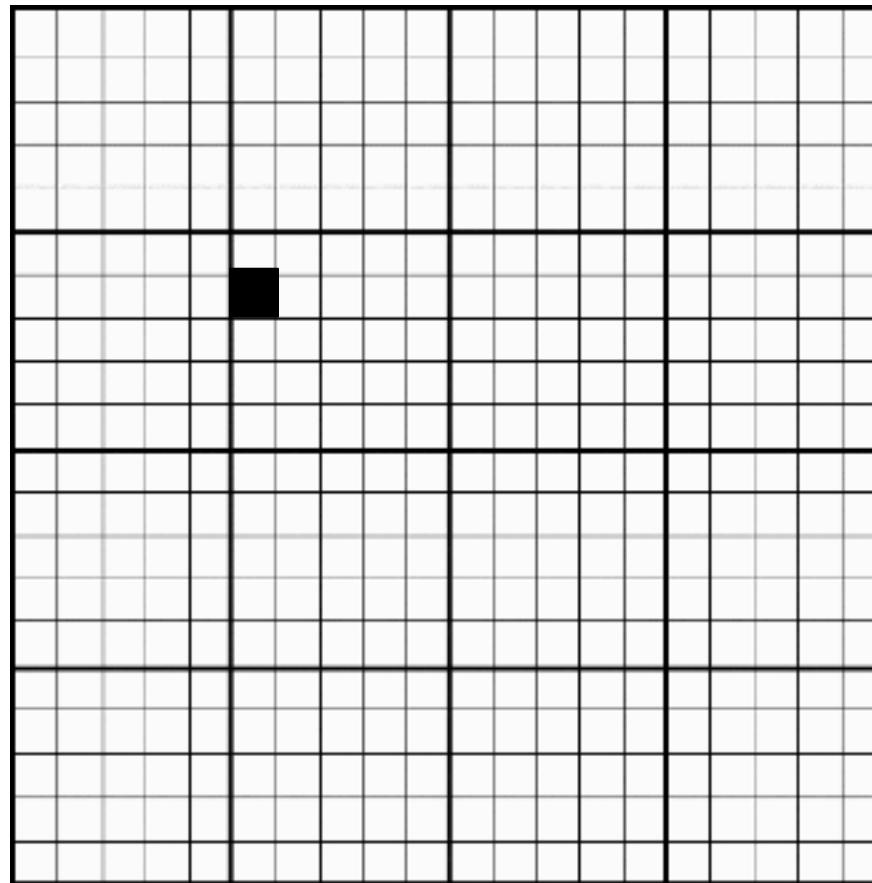
Data this thread can use (including ghost cells from neighbors)



Conway's Game of Life

- Working with ghost cells

Updated cells

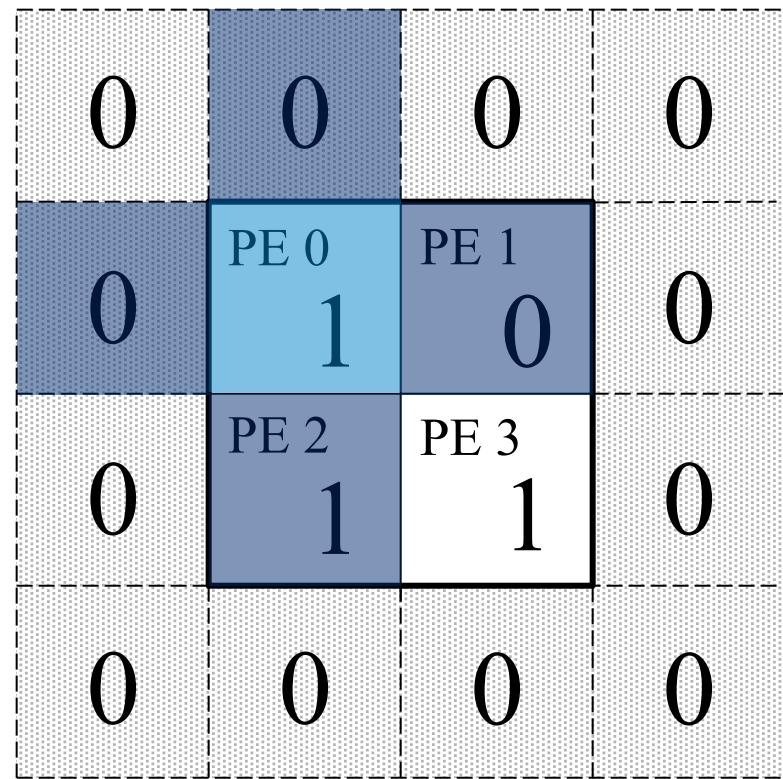


Conway's Game of Life

- Things to consider...
 - What might happen to our ghost cells as we increase the number of threads?
 - ◆ the ghost cells to total cells ratio will rapidly increase causing a greater demand on memory
 - What would be the benefits of using a larger number of ghost cells per thread? Negatives?
 - ◆ in the Game of Life example, we could double or triple our ghost cell boundary, allowing us to perform several iterations without stopping for a ghost cell update

Stencil and Communication Optimizations

- When data is distributed, ghost cells must be **explicitly** communicated among nodes between loop iterations
- Darker cells are PE 0's ghost cells
- After first iteration of stencil computation
 - PE 0 must request PE 1 & PE 2's stencil results
 - PE 0 can perform another iteration of stencil



Stencil and Communication Optimizations

- Generally better to replicate ghost cells in each local memory and swap after each iteration than to share memory
 - Fine-grained sharing can lead to increased communication cost

Stencil and Communication Optimizations

- **Halo:** set of all ghost cells
- Halo must contain all neighbors needed for one iteration
- Larger halo (**deep halo**)
 - Trade off
 - ◆ less communications and more independence, but...
 - ◆ more redundant computation and more memory used
- **Latency Hiding:** Compute interior of stencil while waiting for ghost cell updates

Recurrence

- What if we have several nested loops with data dependencies between them when doing a stencil computation?

Recurrence

```
1 void my_recurrence(
2     size_t v,           // number of elements vertically
3     size_t h,           // number of elements horizontally
4     const float a[v][h], // input 2D array
5     float b[v][h]        // output 2D array (boundaries already initialized )
6 ) {
7     for (int i=1; i<v; ++i)
8         for (int j=1; j<h; ++j)
9             b[i][j] = f(b[i-1][j], b[i][j-1], a[i][j]);
10 }
```

Recurrence

```
1 void my_recurrence(  
2     size_t v,           // number of elements vertically  
3     size_t h,           // number of elements horizontally  
4     const float a[v][h], // input 2D array  
5     float b[v][h]        // output 2D array (boundaries already initialized )  
6 ) {  
7     for (int i=1; i<v; ++i)  
8         for (int j=1; j<h; ++j)  
9             b[i][j] = f(b[i-1][j], b[i][j-1], a[i][j]);  
10 }
```

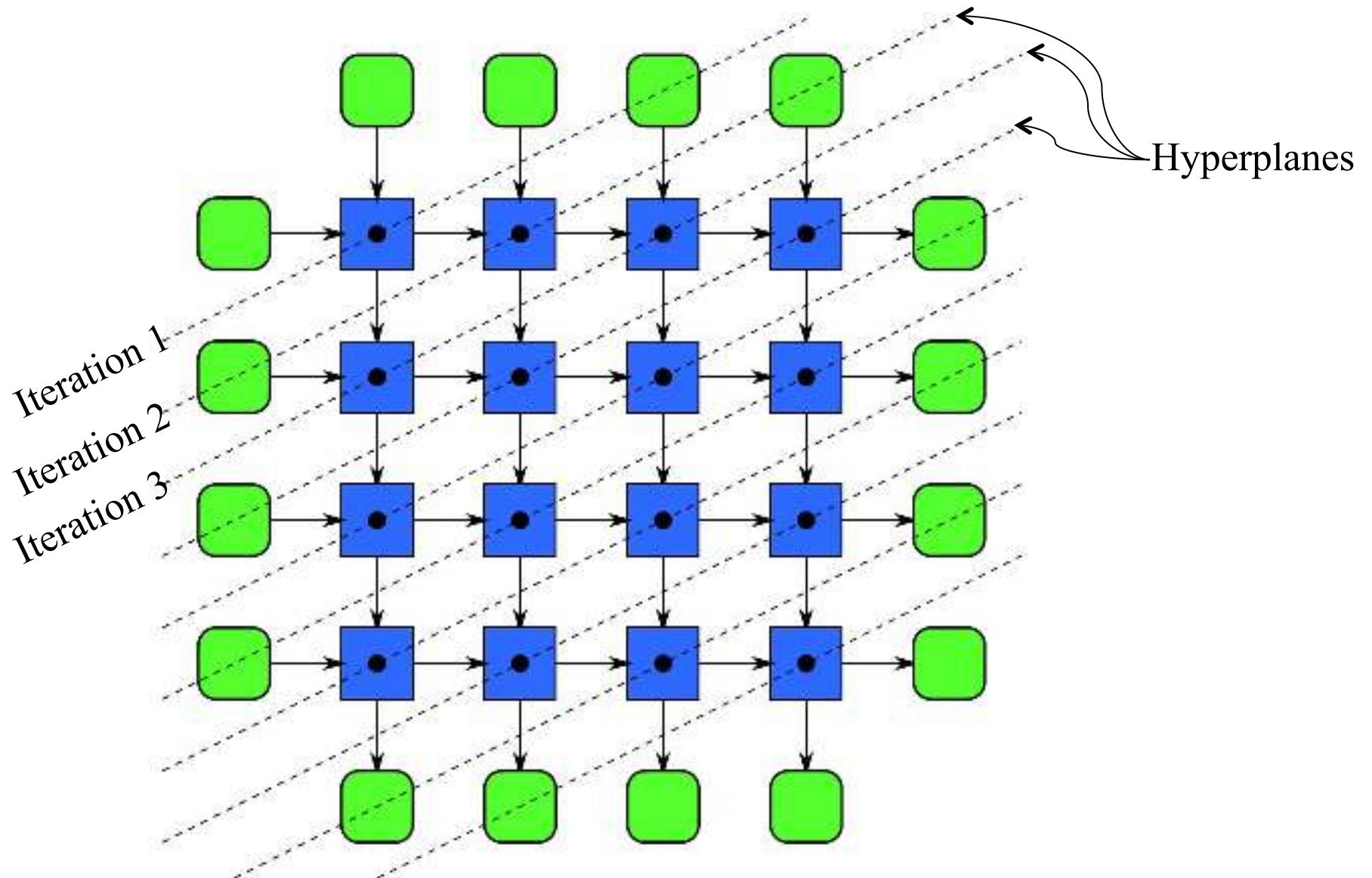


Data dependencies between loops

Recurrence Parallelization

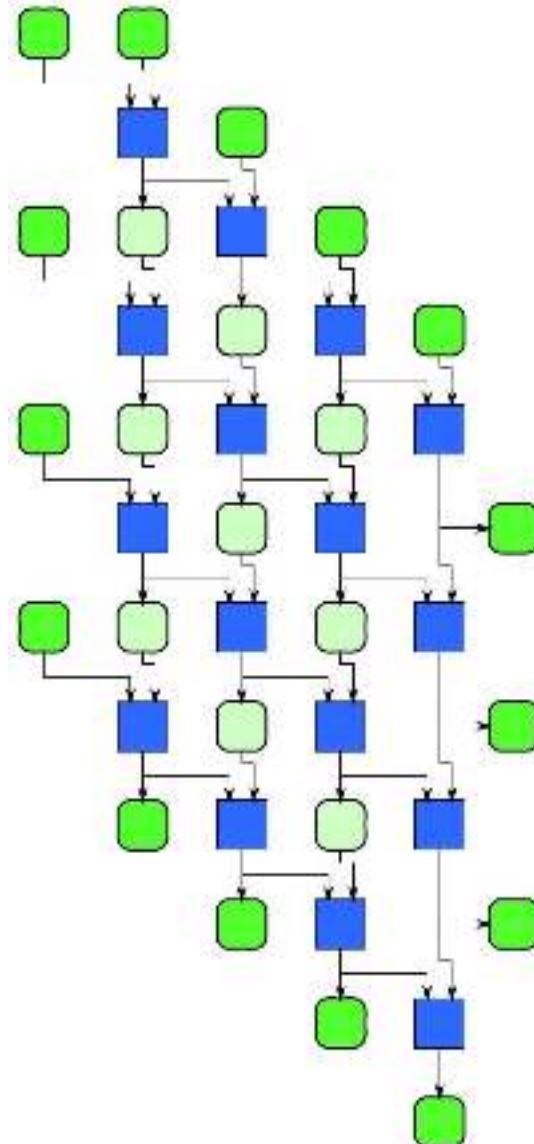
- This can still be parallelized!
- Trick: find a plane that cuts through grid of intermediate results
 - Previously computed values on one side of plane
 - Values to still be computed on other side of plane
 - Computation proceeds perpendicular to plane through time (this is known as a sweep)
- This plane is called a *separating hyperplane*

Recurrence



Recurrence

- Same grid of intermediate results
- Each level corresponds to a loop iteration
- Computation proceeds downward



Conclusion

- Examined the stencil and recurrence pattern
 - Both have a regular pattern of communication and data access
- In both patterns we can convert a set of offset memory accesses to shifts
- Stencils can use strip-mining to optimize cache use
- Ghost cells should be considered when stencil data is distributed across different memory spaces



Fork-Join Pattern

Parallel Computing

CIS 410/510

Department of Computer and Information Science



UNIVERSITY OF OREGON

Outline

- What is the fork-join concept?
- What is the fork-join pattern?
- Programming Model Support for Fork-Join
- Recursive Implementation of Map
- Choosing Base Cases
- Load Balancing
- Cache Locality and Cache-Oblivious Algorithms
- Implementing Scan with Fork-Join
- Applying Fork-Join to Recurrences

Fork-Join Philosophy

When you come to a fork in the road, take it.

(Yogi Bera, 1925 –)

Fork-Join Concept

- Fork-Join is a fundamental way (primitive) of expressing concurrency within a computation
- ***Fork*** is called by a (logical) thread (*parent*) to create a new (logical) thread (*child*) of concurrency
 - Parent continues after the *Fork* operation
 - Child begins operation separate from the parent
 - *Fork* creates concurrency
- ***Join*** is called by both the parent and child
 - Child calls *Join* after it finishes (implicitly on exit)
 - Parent waits until child joins (continues afterwards)
 - *Join* removes concurrency because child exits

Fork-Join Concurrency Semantics

- Fork-Join is a concurrency control mechanism
 - Fork increases concurrency
 - Join decreases concurrency
- Fork-Join dependency rules
 - A parent must join with its forked children
 - Forked children with the same parent can join with the parent in any order
 - A child can not join with its parent until it has joined with all of its children
- Fork-Join creates a special type of DAG
 - What do they look like?

Fork Operation

- Fork creates a child thread
- What does the child do?
- Typically, fork operates by assigning the child thread with some piece of “work”
 - Child thread performs the piece of work and then exits by calling join with the parent
- Child work is usually specified by providing the child with a function to call on startup
- Nature of the child work relative to the parent is not specified

Join Operation

- Join informs the parent that the child has finished
- Child thread notifies the parent and then exits
 - Might provide some status back to the parent
- Parent thread waits for the child thread to join
 - Continues after the child thread joins
- Two scenarios
 1. Child joins first, then parent joins with no waiting
 2. Parent joins first and waits, child joins and parent then continues

Fork-Join Heritage in Unix

- Fork-Join comes from basic forms of creating processes and threads in operating system
- Forking a child process from a parent process
 - Creates a new child process with *fork()*
 - Process state of parent is copied to child process
 - ◆ process ID of parent stored in child process state
 - ◆ process ID of child stored in parent process state
 - Parent process continues to next PC on *fork()* return
 - Child process starts execution at next PC
 - ◆ process ID is automatically set to child process
 - ◆ child can call *exec()* to overlay another program

Fork-Join Heritage in Unix (2)

- Joining a child process with a parent process
 - Child process exits and parent process is notified
 - ◆ if parent is blocked waiting, it unblocks
 - ◆ if parent is not waiting, some indication is made
 - ◆ child process effectively joins
 - Parent process calls waitpid() (effectively join) for a particular child process
 - ◆ if the child process has called join(), parent continues
 - ◆ if the child process has not called join(), parent blocks
- Fork-Join also implemented for threads

Fork-Join “Hello World” in Unix

```
#include <sys/types.h> /* pid_t */
#include <sys/wait.h> /* waitpid */
#include <stdio.h>      /* printf, perror */
#include <stdlib.h>      /* exit */
#include <unistd.h>      /* _exit, fork */

int main(void)
{
    pid_t pid;

    pid = fork();

    if (pid == -1) {
        /*
         * When fork() returns -1, an error happened.
         */
        perror("fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        /*
         * When fork() returns 0, we are in the child process.
         */
        printf("Hello from the child process!\n");
        _exit(EXIT_SUCCESS); /* exit() is unreliable here, so _exit must be used */
    }
    else {
        /*
         * When fork() returns a positive number, we are in the parent process
         * and the return value is the PID of the newly created child process.
         */
        int status;
        (void)waitpid(pid, &status, 0);
    }
    return EXIT_SUCCESS;
}
```

Fork-Join in POSIX Thread Programming

- POSIX standard multi-threading interface
 - For general multi-threaded concurrent programming
 - Largely independent across implementations
 - Broadly supported on different platforms
 - Common target for library and language implementation
- Provides primitives for
 - Thread creation and management
 - Synchronization

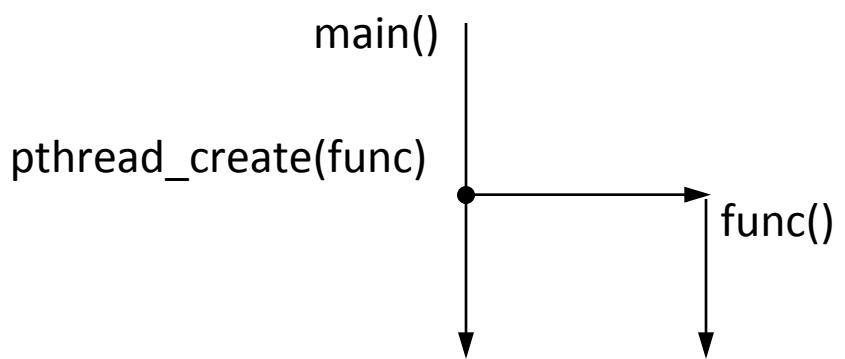
Thread Creation

```
#include <pthread.h>
int pthread_create(
    pthread_t *thread_id,
    const pthread_attr_t *attribute,
    void *(*thread_function) (void *), void *arg);
```

- thread_id
 - thread's unique identifier
- attribute
 - contain details on scheduling policy, priority, stack, ...
- thread_function
 - function to be run in parallel (entry point)
- arg
 - arguments for function func

Example of Thread Creation

```
void *func(void *arg) {  
    int *I=arg;  
    ...  
}  
  
void main()  
{  
    int x;  
    pthread_t id;  
    ...  
    pthread_create(&id, NULL, func, &x);  
    ...  
}
```



```
graph TD; main() --> pthread_create(func); pthread_create --> func();
```

The diagram illustrates the execution flow. It starts with a vertical line labeled "main()". An arrow labeled "pthread_create(func)" points from "main()" to a horizontal line labeled "func()". From the "func()" line, two arrows point downwards, indicating the execution continues in both the main thread and the newly created thread.

Pthread Termination

```
void pthread_exit(void *status)
```

- Terminates the currently running thread
- Implicitly called when function called in
pthread_create returns

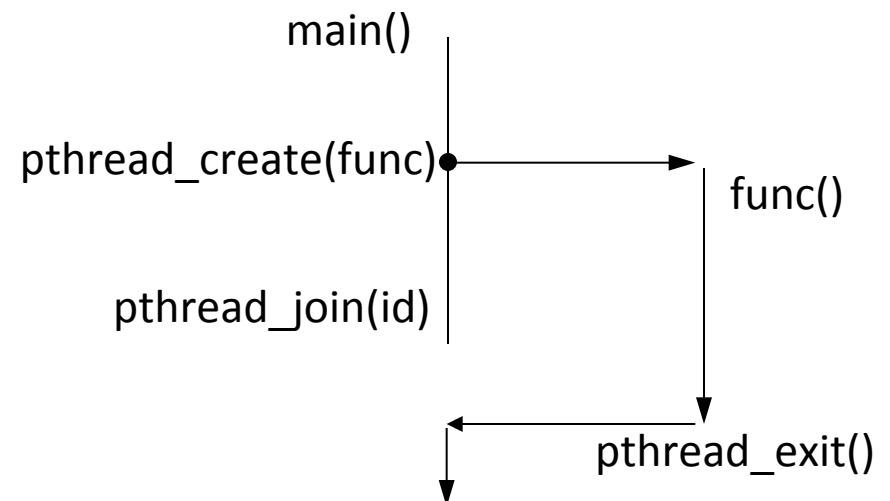
Thread Joining

```
int pthread_join(  
    pthread_t thread_id,  
    void **status);
```

- Waits for thread `thread_id` to terminate
 - Either by returning
 - Or by calling `pthread_exit()`
- Status receives the return value or the value given as argument to `pthread_exit()`

Thread Joining Example

```
void *func(void *) {  
    ...  
}  
pthread_t id;  
int x;  
...  
pthread_create(&id, NULL, func, &x);  
...  
pthread_join(id, NULL);  
...
```



General Program Structure

- Encapsulate parallel parts in functions
- Use function arguments to parameterize thread behavior
- Call `pthread_create()` with the function
- Call `pthread_join()` for each thread created
- Need to take care to make program “thread safe”

Pthread Process Management

- `pthread_create()`
 - Creates a parallel thread executing a given function
 - Passes function arguments
 - Returns thread identifier
- `pthread_exit()`
 - terminates thread.
- `pthread_join()`
 - waits for particular thread to terminate

Pthreads Synchronization

- Create/exit/join
 - Provide some coarse form of synchronization
 - “Fork-join” parallelism
 - Requires thread creation/destruction
- Need for finer-grain synchronization
 - Mutex locks
 - Condition variables

Pthreads “Hello World”

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define NUM_THREADS      5

void *TaskCode(void *argument)
{
    int tid;

    tid = *((int *) argument);
    printf("Hello World! It's me, thread %d!\n", tid);

    /* optionally: insert more useful stuff here */

    return NULL;
}

int main(void)
{
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    int rc, i;

    /* create all threads */
    for (i=0; i<NUM_THREADS; ++i) {
        thread_args[i] = i;
        printf("In main: creating thread %d\n", i);
        rc = pthread_create(&threads[i], NULL, TaskCode, (void *) &thread_args[i]);
        assert(0 == rc);
    }

    /* wait for all threads to complete */
    for (i=0; i<NUM_THREADS; ++i) {
        rc = pthread_join(threads[i], NULL);
        assert(0 == rc);
    }

    exit(EXIT_SUCCESS);
}
```

Fork-Join Pattern

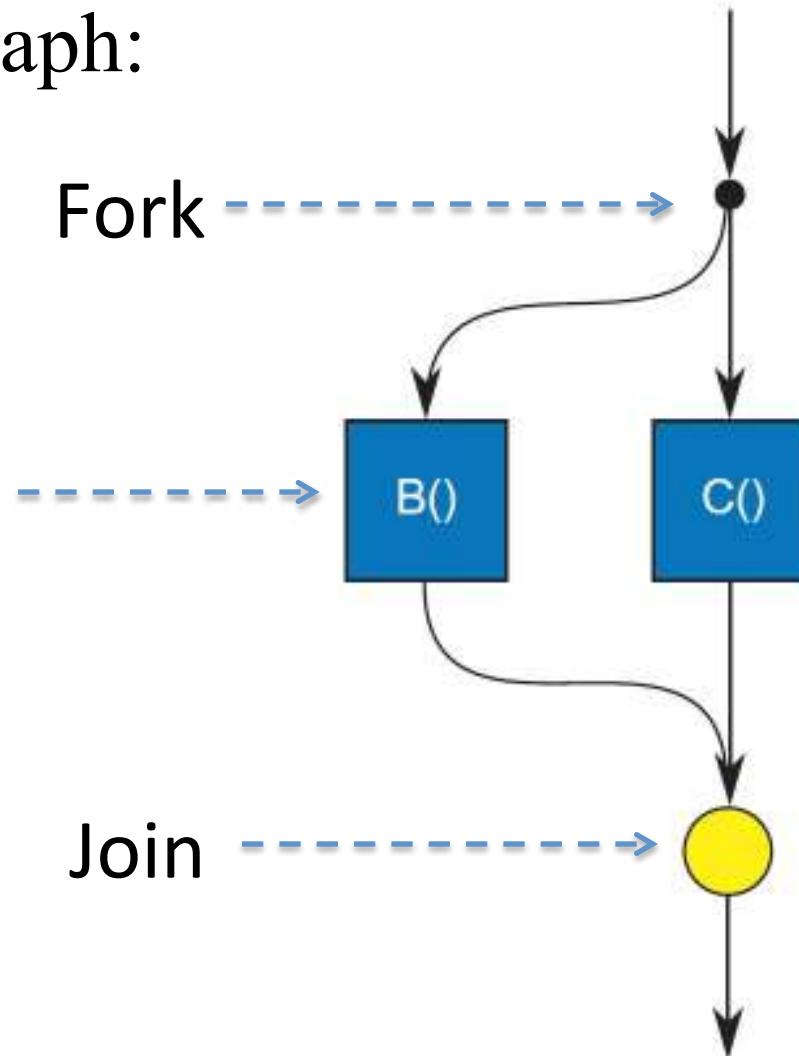
- Control flow **divides** (forks) into multiple flows, then **combines** (joins) later
- During a fork, one flow of control becomes two
- Separate flows are “independent”
 - Does “independent” mean “not dependent” ?
 - No, it just means that the 2 flows of control “are not constrained to do similar computation”
- During a join, two flows become one, and only this one flow continues

Fork-Join Pattern

- Fork-Join directed graph:

Independent work

Is it possible for B() and C() to have dependencies between them?

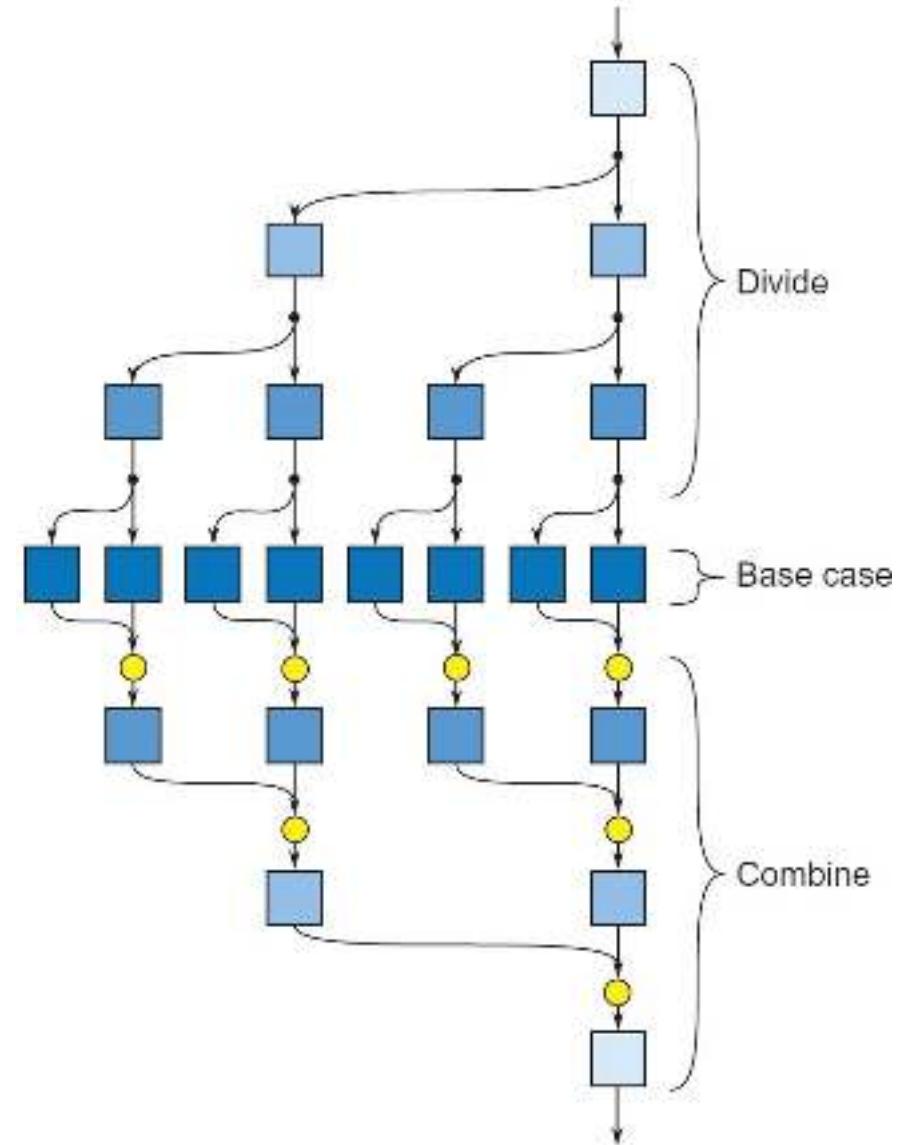


Fork-Join Pattern

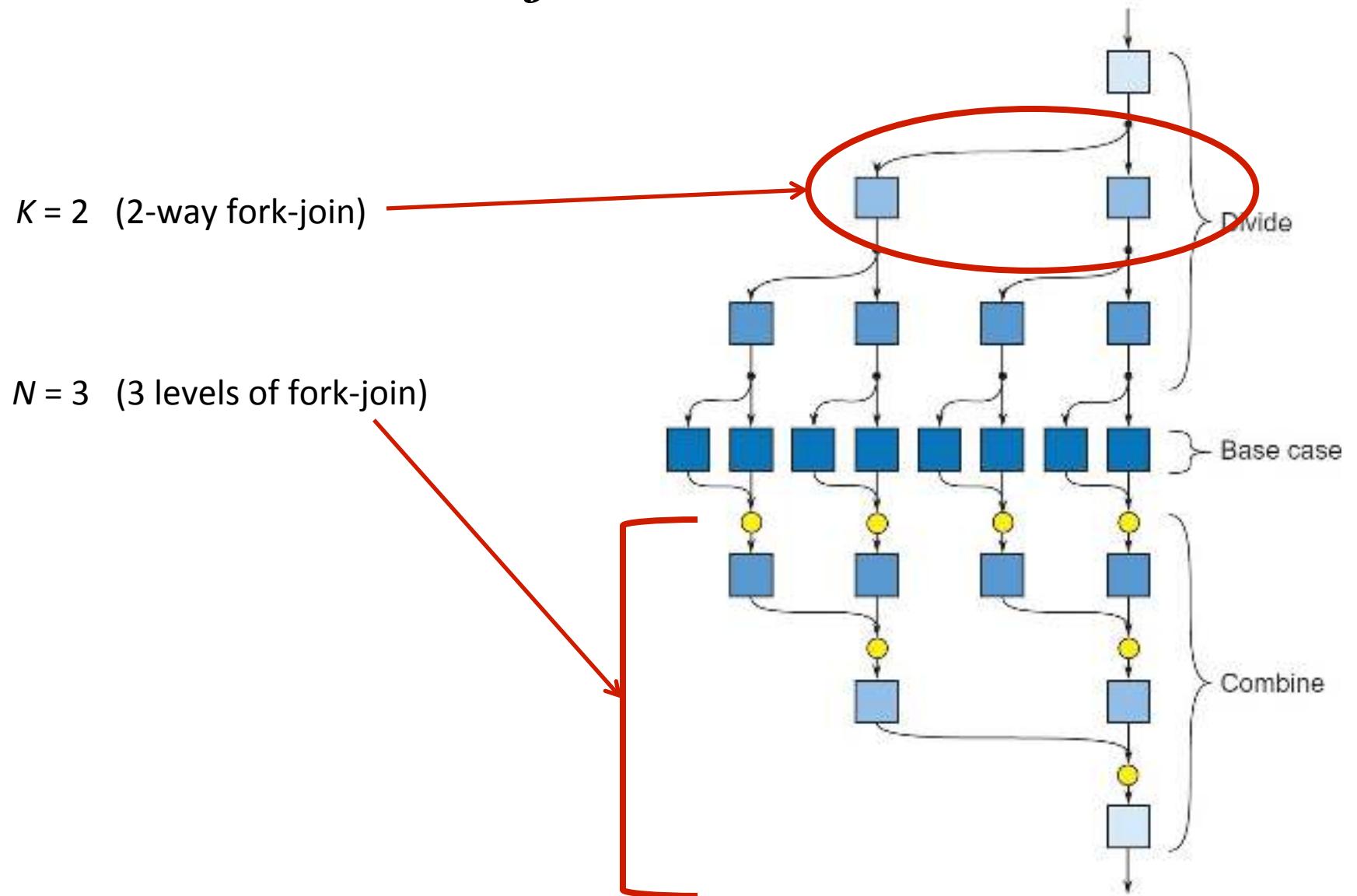
- Typical **divide-and-conquer** algorithm implemented with fork-join:

```
void DivideAndConquer( Problem P ) {  
    if( P is base case ) {  
        Solve P;  
    } else {  
        Divide P into K subproblems;  
        Fork to conquer each subproblem in parallel;  
        Join;  
        Combine subsolutions into final solution;  
    }  
}
```

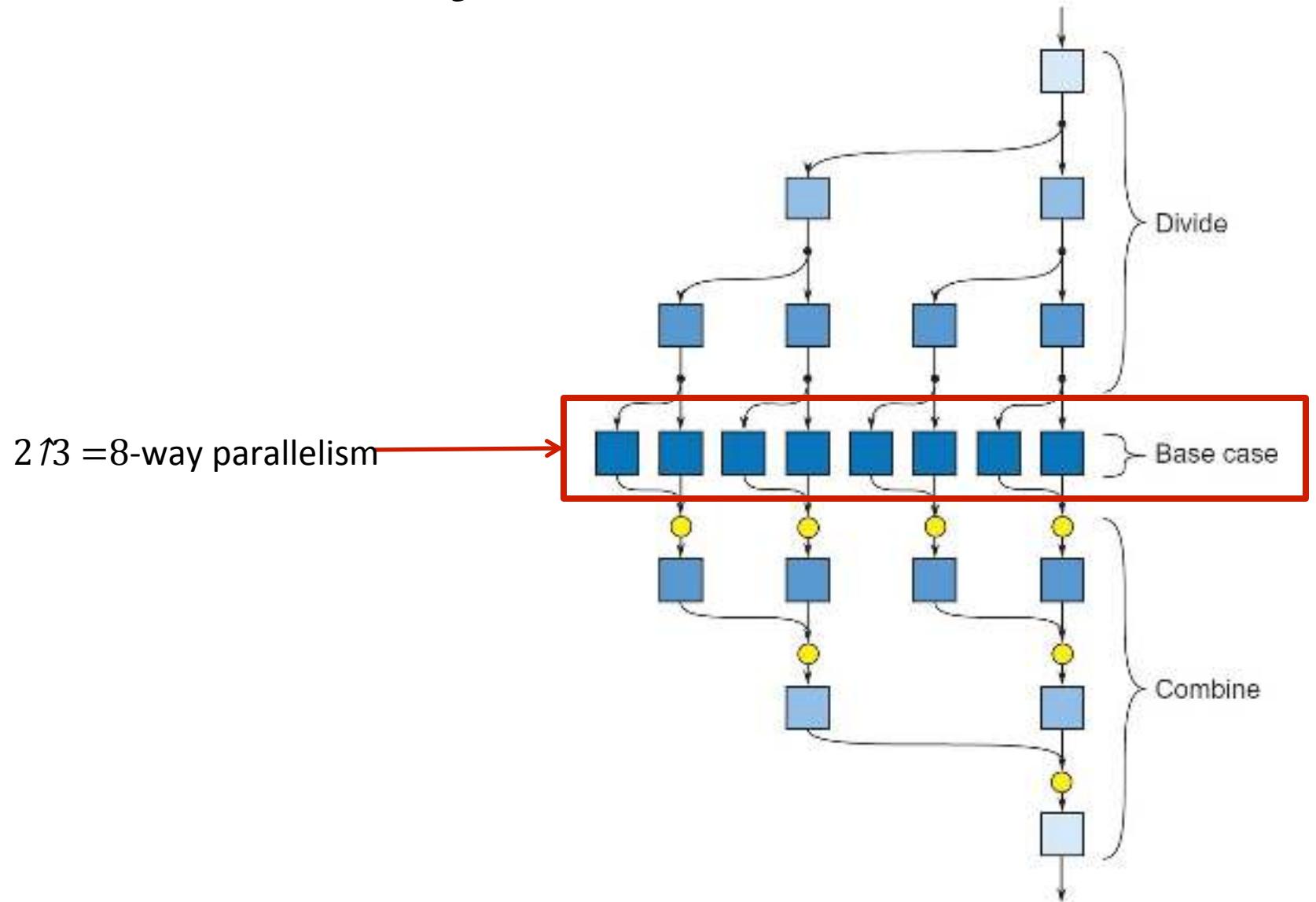
Fork-Join Pattern for Divide-Conquer



Fork-Join Pattern for Divide-Conquer



Fork-Join Pattern for Divide-Conquer



Fork-Join Pattern

- Selecting the base case size is critical
- Recursion must go deep enough for plenty of parallelism
- Too deep, and the granularity of sub-problems will be dominated by scheduling overhead
- With K-way fork-join and N levels of fork-join, can have up to K^N -way parallelism

Fibonacci Example

- Recursive Fibonacci is simple and inefficient

```
long fib ( int n ) {  
    if (n < 2) return 1;  
    else {  
        long x = fib (n-1);  
        long y = fib(n-2);  
        return x + y;  
    }  
}
```

Fibonacci Example

- Recursive Fibonacci is simple and inefficient
- Are there dependencies between the sub-calls?
- Can we parallelize it?

Fibonacci in Parallel Example

```
long fib ( int n ) {  
    if (n < 2) return 1;  
    else {  
        long x = fork fib (n-1);  
        long y = fib(n-2);  
        join;  
        return x + y;  
    }  
}
```

Programming Model Support for Fork-Join

- Cilk Plus:

```
cilk_spawn B(); — Fork  
C();  
cilk_sync; —————— Join
```

- B() executes in the child thread
- C() executes in the parent thread

Programming Model Support for Fork-Join

□ Cilk Plus:

```
cilk_spawn B();
C();
cilk_sync;
-----  
cilk_spawn A();
cilk_spawn B();
cilk_spawn C();
D();           // Not spawned, executed in spawning task
cilk_sync;    // Join
-----  
for ( int i=0; i<n; ++i )
    if ( a[i]!=0 )
        cilk_spawn f(a[i]);
cilk_sync;
```

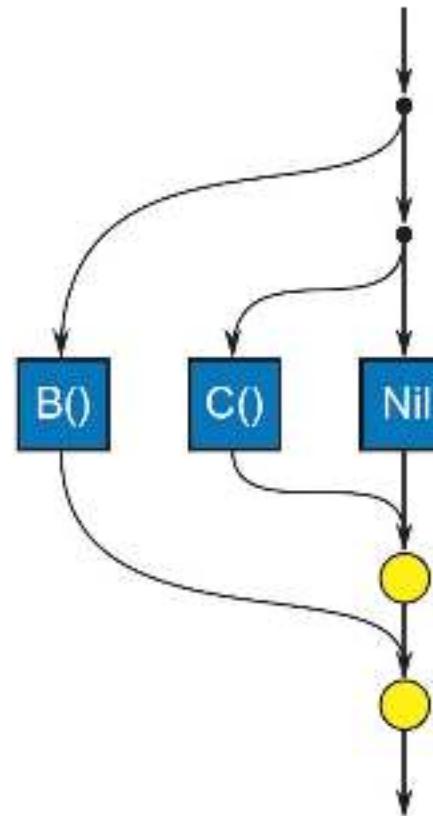
Good form!

Programming Model Support for Fork-Join

- Cilk Plus:

```
cilk_spawn B();  
cilk_spawn C();  
/* nil */  
cilk_sync;
```

Bad form! Why?



Programming Model Support for Fork-Join

- TBB
 - `parallel_invoke()`
 - ◆ For 2 to 10 way fork
 - ◆ Joins all tasks before returning
 - `Tbb::task_group`
 - ◆ For more complicated cases
 - ◆ Provides explicit join

```
task_group g;
for ( int i=0; i<n; ++i )
    if ( a[i] != 0 )
        g.run( [=,&a]{f(a[i]);} ); // Spawn f(a[i]) as child task
g.wait(); // Wait for all tasks spawned from g
```

Programming Model Support for Fork-Join

□ OpenMP:

```
#pragma omp task
B(); ← Forked task
C(); ← Performed by
#pragma omp taskwait
spawning task
```

Programming Model Support for Fork-Join

- OpenMP:

```
#pragma omp task  
→ B();  
C();  
#pragma omp taskwait
```

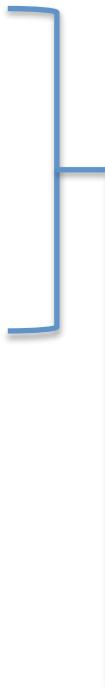
- Forked task can also be a compound statement:

```
{ B(); C(); D(); }
```

Programming Model Support for Fork-Join

- OpenMP:

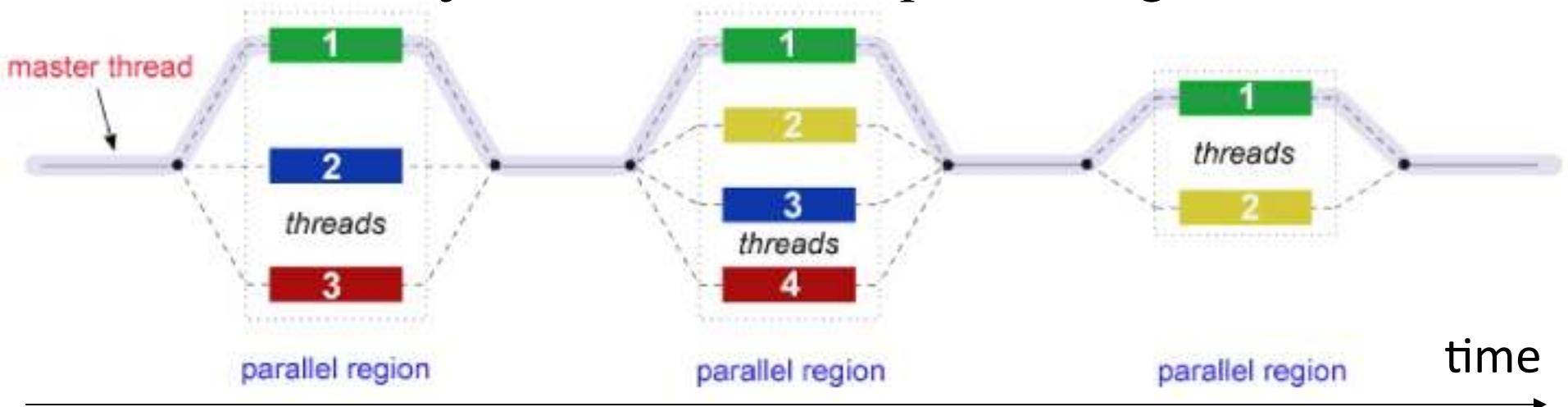
```
#pragma omp task  
B();  
C();  
#pragma omp taskwait
```



Must be enclosed in an
OpenMP parallel
construct

More to the OpenMP Fork-Join Story

- OpenMP uses a fork-join model of parallel execution as a fundamental basis of the language
- All OpenMP programs begin as a single process
 - *Master* thread executes until a parallel region is encountered
- OpenMP runtime systems executes the parallel region by forking a team of (*Worker*) parallel threads
 - Statements in parallel region are executed by worker threads
- Team threads join with master at parallel region end



OpenMP – General Rules

- Most OpenMP constructs are *compiler directives*
- Directives inform the compiler
 - Provide compiler with knowledge
 - Usage assumptions
- Directives are ignored by non-OpenMP compilers!
 - Essentially act as comment for backward compatibility
- Most OpenMP constructs apply to structured blocks
 - A block of code with one point of entry at the top and one point of exit at the bottom
 - Loops are a common example of structured blocks
 - ◆ excellent source of parallelism

OpenMP PARALLEL Directive

- Specifies what should be executed in parallel:
 - A program section (structured block)
 - If applied to a loop, what happens is:
 - ◆ iterations are executed in parallel
 - ◆ do loop (Fortran)
 - ◆ for loop (C/C++)
- PARALLEL DO is a “worksharing” directive
 - Causes work to be shared across threads
 - More on this later

PARALLEL DO: Syntax

□ Fortran

```
!$omp parallel do [clause [,] [clause ...]]  
do index = first, last [, stride]  
    body of the loop  
enddo  
[ !$omp end parallel do]
```

The loop body executes
in parallel across
OpenMP threads

□ C/C++

```
#pragma omp parallel for [clause [clause ...]]  
for (index = first; text_expr;  
     increment_expr) {  
    body of the loop  
}
```

Example: PARALLEL DO

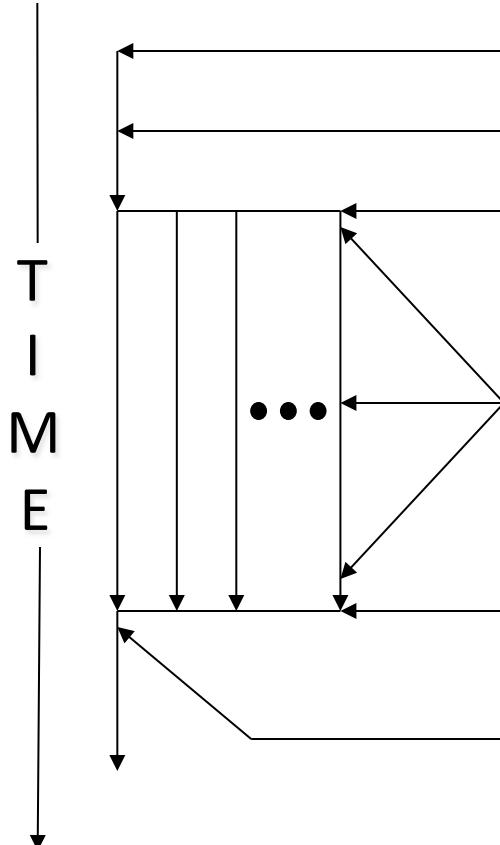
- Single precision $a^*x + y$ (*saxpy*)

```
subroutine saxpy (z, a, x, y, n)
integer i, n
real z(n), a, x(n), y(n)
!$omp parallel do
do i = 1, n
    z(i) = a * x(i) + y(i)
enddo
return
end
```

What is the degree of concurrency?

What is the degree of parallelism?

Execution Model of PARALLEL DO



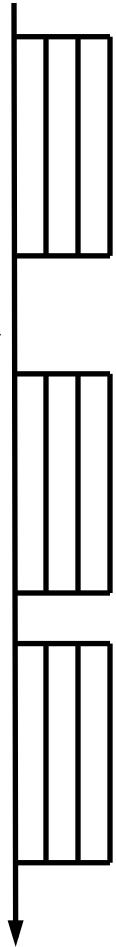
- Master thread executes serial portion of code
- Master thread enters *saxpy* routine
- Master thread encounters *parallel do* directive
- Creates slave threads (How many?)
- Master and slave threads divide iterations of parallel do loop and execute them concurrently
- Implicit synchronization: wait for all threads to finish their allocation of iterations
- Master thread resumes execution after the do loop
- Slave threads disappear

- Abstract execution model – a Fork-Join model!!!

Loop-level Parallelization Paradigm

- Execute each loop in parallel
 - Where possible
- Easy to parallelize code
- Incremental parallelization
 - One loop at a time
 - What happens between loops?
- Fine-grain overhead
 - Frequent synchronization
- Performance determined by sequential part (Why?)

```
C$OMP PARALLEL DO
do i=1, n
.....
enddo
alpha = xnorm/sum
C$OMP PARALLEL DO
do i=1, n
.....
enddo
C$OMP PARALLEL DO
do i=1, n
.....
enddo
```



Example: PARALLEL DO – Bad saxpy

- Single precision $a^*x + y$ (*saxpy*)

```
subroutine saxpy (z, a, x, y, n)
integer i, n
real z(n), a, x(n), y(n)
!$omp parallel do
do i = 1, n
    y(i) = a * x(i+1) + y(i+1)
enddo
return
end
```

What happens here?

How Many Threads?

- Use environment variable
 - `setenv OMP_NUM_THREADS 8` (Unix machines)
- Use `omp_set_num_threads()` function

```
subroutine saxpy (z, a, x, y, n)
integer i, n
real z(n), a, x(n), y(n)          Not a directive,
!$omp call omp_set_num_threads(4)  but a call to the
!$omp parallel do                  OpenMP library
do i = 1, n
    z(i) = a * x(i) + y(i)
enddo
return
end
```

Assigning Iterations to Threads

- A parallel loop in OpenMP is a worksharing directive
- The manner in which iterations of a parallel loop are assigned to threads is called the loop's *schedule*
- Default schedule assigns iterations to threads as evenly as possible (good enough for saxpy)
- Alternative user-specified schedules possible
- More on scheduling later

PARALLEL DO: The Small Print

- The programmer has to make sure that the iterations can in fact be executed in parallel
 - No automatic verification by the compiler

```
subroutine nopalallel (z, a, x, y, n)
integer i, n
real z(n), a, x(n), y(n)
!$omp parallel do
do i = 2, n
    z(i) = a * x(i) + y(i) + z(i-1)
enddo
return
end
```

PARALLEL Directive

□ Fortran

```
!$omp parallel [clause [,] [clause ...]]  
structured block  
!$omp end parallel
```

□ C/C++

```
#pragma omp parallel [clause [clause ...]]  
structured block
```

Parallel Directive: Details

- When a parallel directive is encountered, threads are spawned which execute the code of the enclosed structured block (i.e., the parallel region)
- The number of threads can be specified just like for the PARALLEL DO directive
- The parallel region is replicated and each thread executes a copy of the replicated region

Example: Parallel Region

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_thread_num(); ★
    pooh(ID, A);
}
printf("all done\n");
```

ID = omp_thread_num() ... ID = omp_thread_num()
pooh(0, A) pooh(1, A) pooh(2, A) pooh(3, A)

printf("all done\n");

Is this ok?

Parallel versus Parallel Do

- Arbitrary structured blocks versus loops
- Coarse grained versus fine grained
- Replication versus work division (work sharing)

```
!$omp parallel do
do I = 1,10
    print *, 'Hello world', I
enddo
```

PARALLEL DO is a
work sharing directive

Output: 10 Hello world messages

```
!$omp parallel
do I = 1,10
    print *, 'Hello world', I
enddo
!$omp end parallel
```

Output: $10*T$ Hello world messages
where T = number of threads

Parallel: Back to Motivation

```
omp_set_num_threads(2);
#pragma omp parallel private(i, j, x, y, my_width,
    my_thread, i_start, i_end)
{
    my_width = m/2;
    my_thread = omp_get_thread_num();
    i_start = 1 + my_thread * my_width;
    i_end = i_start + my_width - 1;
    for (i = i_start; i <= i_end; i++)
        for (j = 1; j <= n; j++) {
            x = i/ (double) m;
            y = j/ (double) n;
            depth[j][i] = mandel_val(x, y, maxiter);
        }
    for (i = i_start; i <= i_end; i++)
        for (j = 1; j <= n; j++)
            dith_depth[j][i] = 0.5*depth[j][i]
                + 0.25*(depth[j-1][i] + depth[j+1][i])
}
```

What is going on here?

Work Sharing in Parallel Regions

- Manual division of work (previous example)
- OMP *worksharing* constructs
 - Simplify the programmers job in dividing work among the threads that execute a parallel region
 - ◆ **do** directive
have different threads perform different iterations of a loop
 - ◆ **sections** directive
identify sections of work to be assigned to different threads
 - ◆ **single** directive
specify that a section of code is to be executed by one thread only
(remember default is replicated)

DO Directive

□ Fortran

```
!$omp parallel [clause [,] [clause ...]]  
...  
!$omp do [clause [,] [clause ...]]  
    do loop  
!$omp enddo [nowait]  
...  
!$omp end parallel
```

□ C/C++

```
#pragma omp parallel [clause [clause ...]]  
{  
...  
#pragma omp for [clause [clause] ... ]  
    for-loop  
}
```

DO Directive: Details

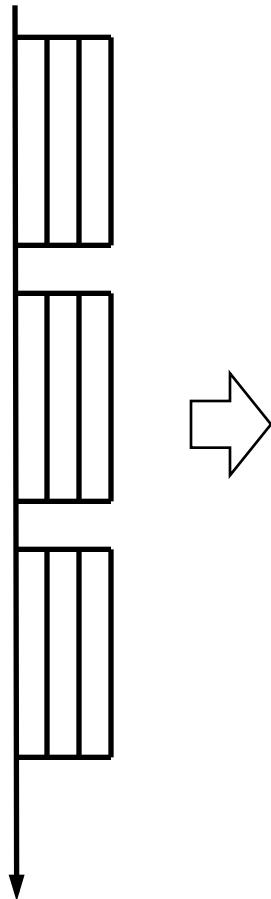
- The DO directive does not spawn new threads!
 - It just assigns work to the threads already spawned by the PARALLEL directive
- The work↔thread assignment is identical to that in the PARALLEL DO directive

```
!$omp parallel do
do I = 1,10
    print *, 'Hello world', I
enddo
```

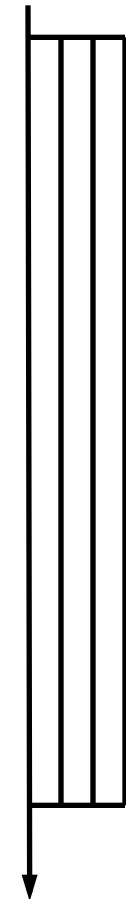
```
!$omp parallel
 !$omp do
 do I = 1,10
    print *, 'Hello world', I
enddo
 !$omp enddo
 !$omp end parallel
```

Coarser-Grain Parallelism

```
C$OMP PARALLEL DO  
do i=1,n  
.....  
enddo  
C$OMP PARALLEL DO  
do i=1,n  
.....  
enddo  
C$OMP PARALLEL DO  
do i=1,n  
.....  
enddo
```



```
C$OMP PARALLEL  
C$OMP DO  
do i=1,n  
.....  
enddo  
C$OMP DO  
do i=1,n  
.....  
enddo  
C$OMP DO  
do i=1,n  
.....  
enddo  
C$OMP PARALLEL
```



- ❑ What's going on here? Is this possible? When?
- ❑ Is this better? Why?

SECTIONS Directive

□ Fortran

```
!$omp sections [clause [,] [clause ...]]  
[ !$omp section]  
    code for section 1  
[ !$omp section  
    code for section 2]  
...  
!$omp end sections [nowait]
```

□ C/C++

```
#pragma omp sections [clause [clause ...]]  
{  
    [#pragma omp section]  
        block  
    ...  
}
```

SECTIONS Directive: Details

- Sections are assigned to threads
 - Each section executes once
 - Each thread executes zero or more sections
- Sections are not guaranteed to execute in any order

```
#pragma omp parallel
#pragma omp sections
{
    x_calculation();
#pragma omp section
    y_calculation();
#pragma omp section
    z_calculation();
}
```

OpenMP Fork-Join Summary

- OpenMP parallelism is Fork-Join parallelism
- Parallel regions have logical Fork-Join semantics
 - OMP runtime implements a Fork-Join execution model
 - Parallel regions can be nested!!!
 - ◆ can create arbitrary Fork-Join structures
- OpenMP tasks are an explicit Fork-Join construct

Recursive Implementation of Map

- Map is a simple, useful pattern that fork-join can implement
- Good to know how to implement map with fork-join if you ever need to write your own map with novel features (fusing map with other patterns)
- Cilk Plus and TBB implement their map constructs with a similar divide-and-conquer algorithm

Recursive Implementation of Map

```
cilk_for( unsigned i=lower; i<upper; ++i )  
    f(i);
```

cilk_for can be implemented with a divide-and-conquer routine...

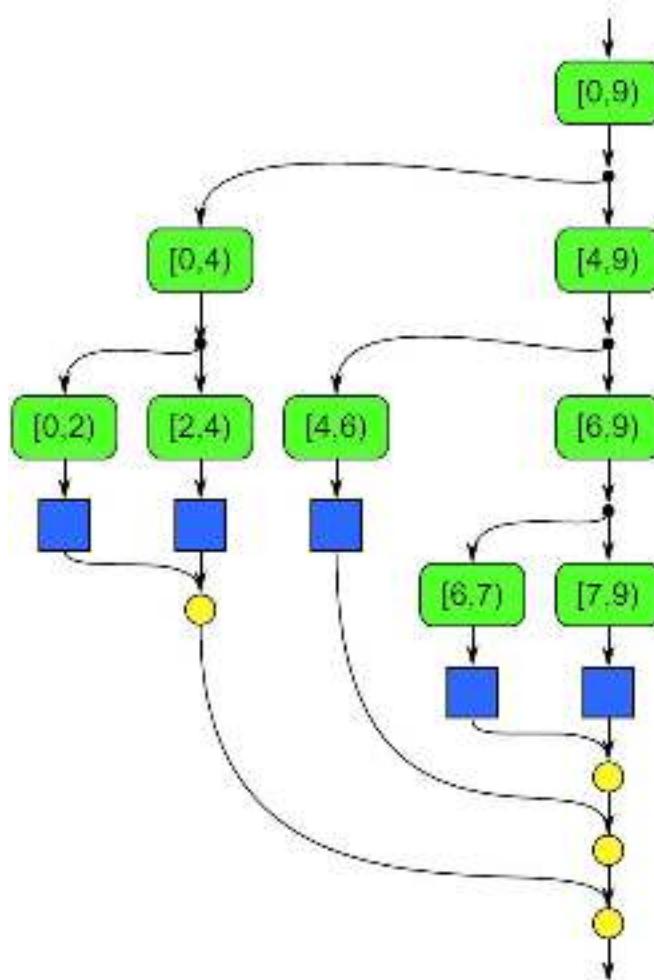
```
if( lower<upper )  
    recursive_map(lower,upper,grainsize,f)
```

Recursive Implementation of Map

```
1 template<typename Func>
2 void recursive_map( unsigned lower, unsigned upper, unsigned grainsize, Func f ) {
3     if( upper-lower<=grainsize )
4         // Parallel base case
5         for( unsigned i=lower; i<upper; ++i )
6             f(i);
7     else {
8         // Divide and conquer
9         unsigned middle = lower+(upper-lower)/2u;
10        cilk_spawn recursive_map( lower, middle, grainsize, f );
11        recursive_map( middle, upper, grainsize, f );
12    }
13    // Implicit cilk_sync when function returns
14 }
```

Recursive Implementation of Map

- recursive_map(0, 9, 2, f)



Choosing Base Cases (1)

- For parallel divide-and-conquer, two base cases:
 - Stopping parallel recursion
 - Stopping serial recursion
- For a machine with P hardware threads, we might think to have P leaves in the spawned functions tree
- This often leads to poor performance
 - Scheduler has no flexibility to balance load

Choosing Base Cases (2)

- Given leaves from spawned function tree with equal work, and equivalent processors, system effects can effect load balance:
 - Page faults
 - Cache misses
 - Interrupts
 - I/O
- Best to **over-decompose** a problem
- This creates **parallel slack**

Choosing Base Cases (3)

- **Over-decompose:** parallel programming style where more tasks are specified than there are physical workers. Beneficial in load balancing.

- **Parallel slack:** Amount of extra parallelism available above the minimum necessary to use the parallel hardware resources.

Load Balancing

- Sometimes, threads will finish their work at different rates
- When this happens, some threads may have nothing to do while others may have a lot of work to do
- This is known as a **load balancing** issue

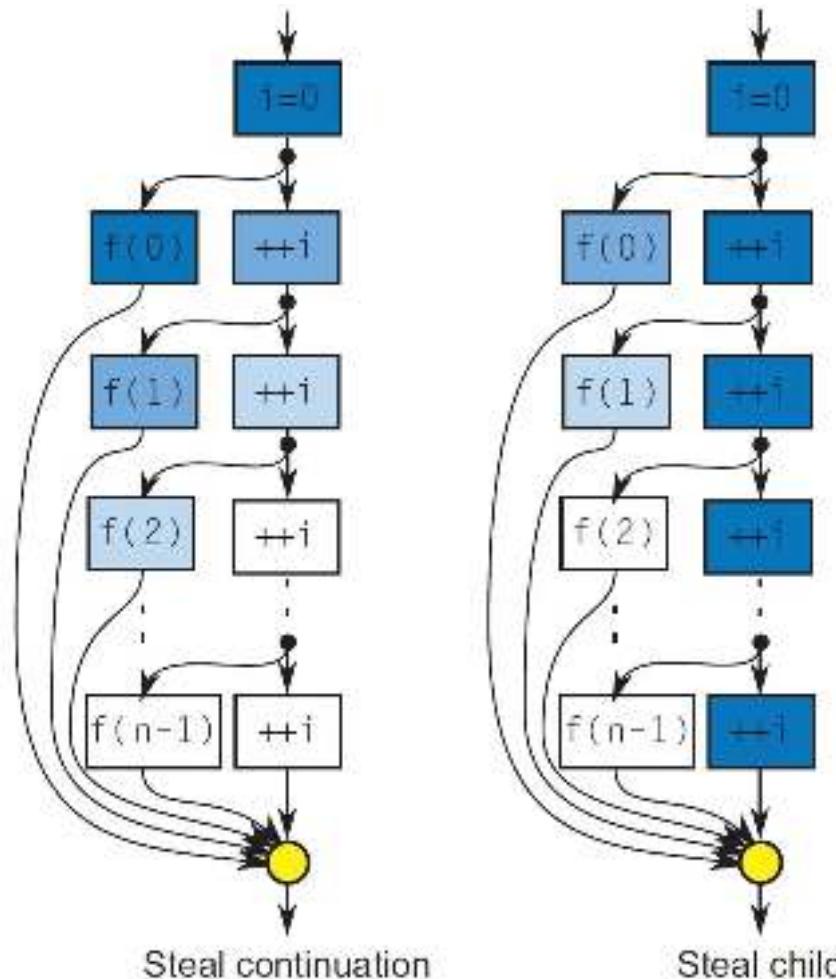
TBB and Cilk Plus Work Stealing (1)

- TBB and Cilk Plus use **work stealing** to automatically balance fork-join work
- In a work-stealing scheduler, each thread is a **worker**
- Each worker maintains a stack of tasks
- When a worker's stack is empty, it grabs from the *bottom* of another random worker
 - Tasks at the bottom of a stack are from the beginning of the call tree – tend to be a bigger piece of work
 - Stolen work will be distant from stack's owner, minimizing cache conflicts

TBB and Cilk Plus Work Stealing (2)

- TBB and Cilk Plus work-stealing differences:

Cilk Plus



TBB

Performance of Fork/Join

Let $A \parallel B$ be interpreted as “fork A, do B, and join”

Work: $T(A \parallel B)_1 = T(A)_1 + T(B)_1$

Span: $T(A \parallel B)_{\infty} = \max(T(A)_{\infty}, T(B)_{\infty})$

Cache-Oblivious Algorithms (1)

- Work/Span analysis ignores memory **bandwidth** constraints that often limit speedup
- Cache reuse is important when memory bandwidth is critical resource
- Tailoring algorithms to optimize cache reuse is difficult to achieve across machines
- **Cache-oblivious programming** is a solution for this
- Code is written to work well regardless of cache structure

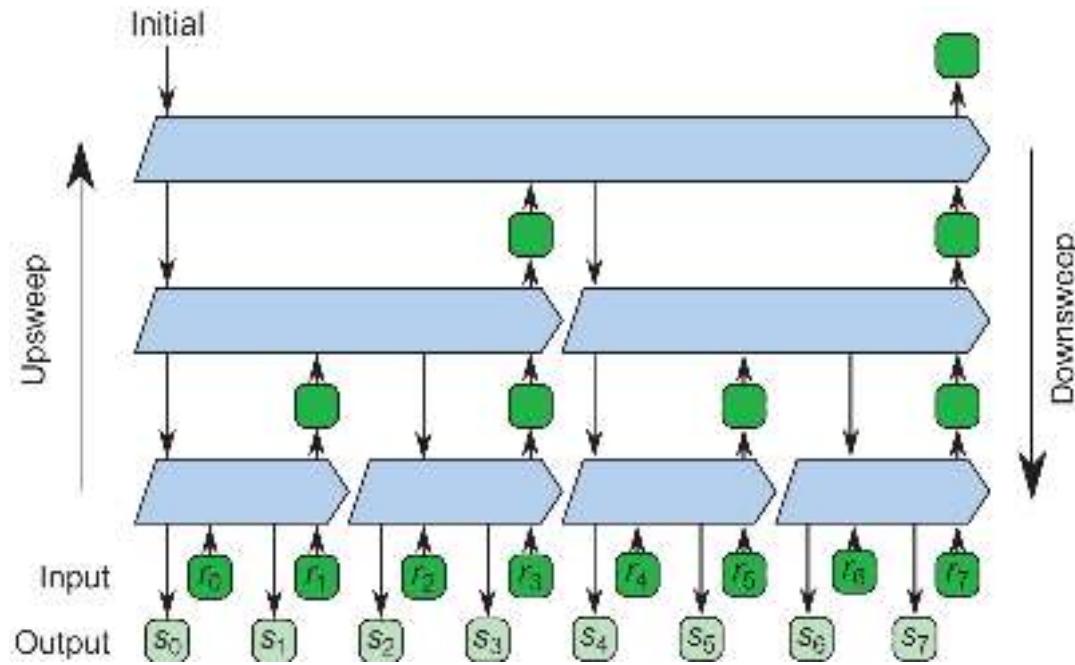
Cache-Oblivious Algorithms (2)

- Cache-oblivious programming strategy:
 - Recursive divide-and-conquer – good data locality at multiple scales
 - When a problem is subdivided enough, it can fit into the largest cache level
 - Continue subdividing to fit data into smaller and faster cache
- Example problem: matrix multiplication
 - Typical, non-recursive, algorithm uses three nested loops
 - Large matrices won't fit in cache with this approach

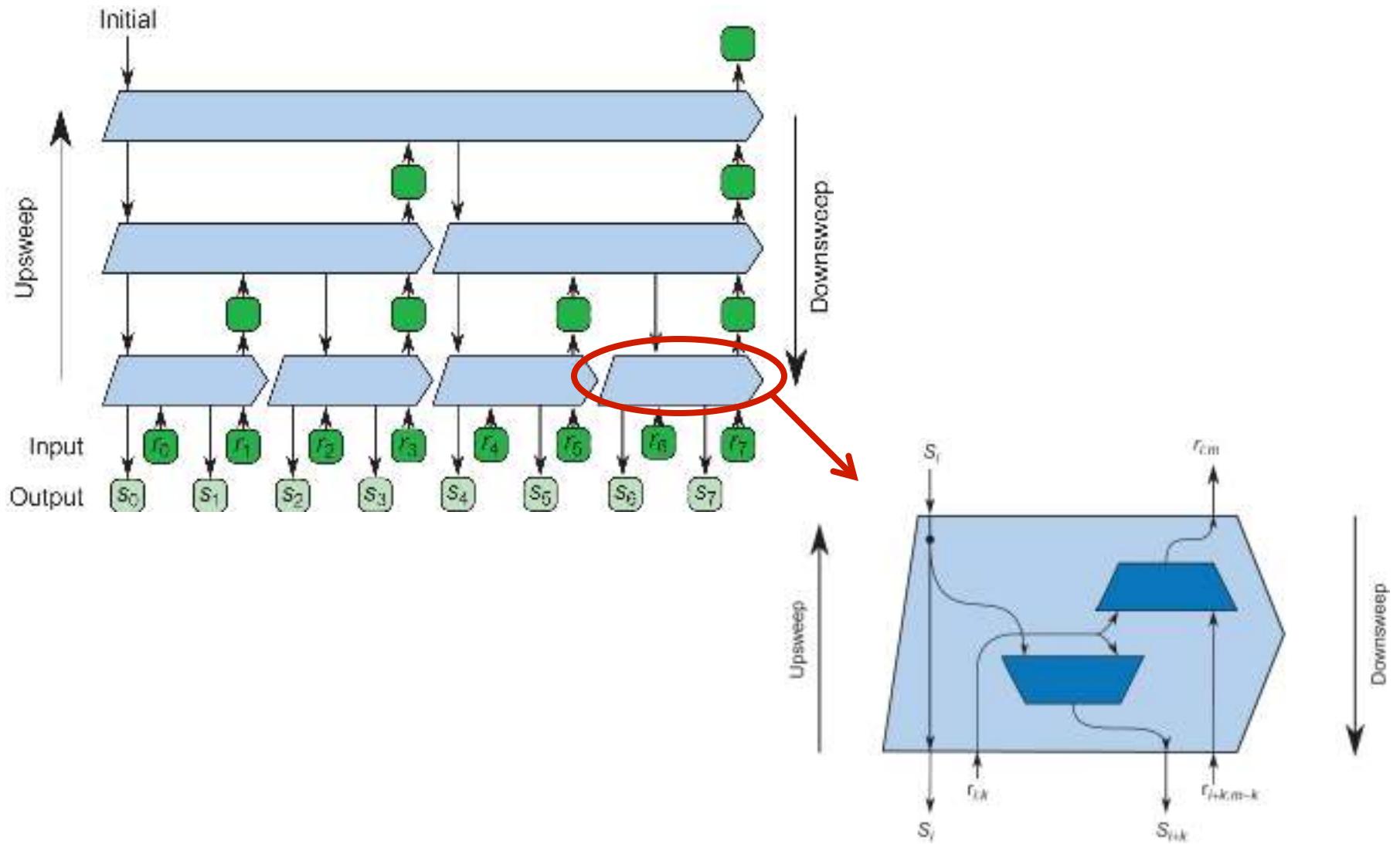
Implementing Scan with Fork-Join

- We saw that the map pattern can be implemented with the fork-join pattern
- Now we will examine how to implement the scan operation with fork-join
- Input: initial value, *initial*, and sequence, $r \downarrow 0, r \downarrow 1, \dots, r \downarrow n$
- Output: exclusive scan, $s \downarrow 0, s \downarrow 1, \dots, s \downarrow n$
- Upsweep computes a set of partial reductions on tiles of data
- Downsweep computes final scan by combining partial reductions

Implementing Scan with Fork-Join



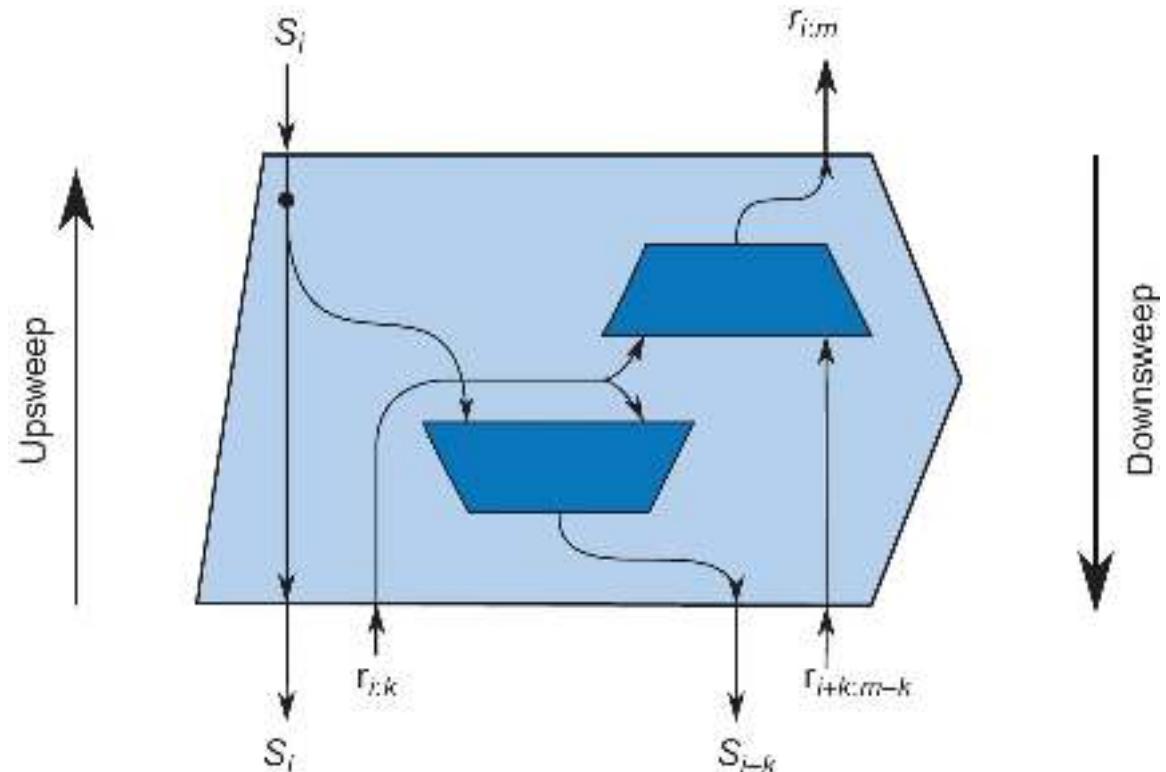
Implementing Scan with Fork-Join



Implementing Scan with Fork-Join

- During the upsweep, each node computes a partial reduction of the form:

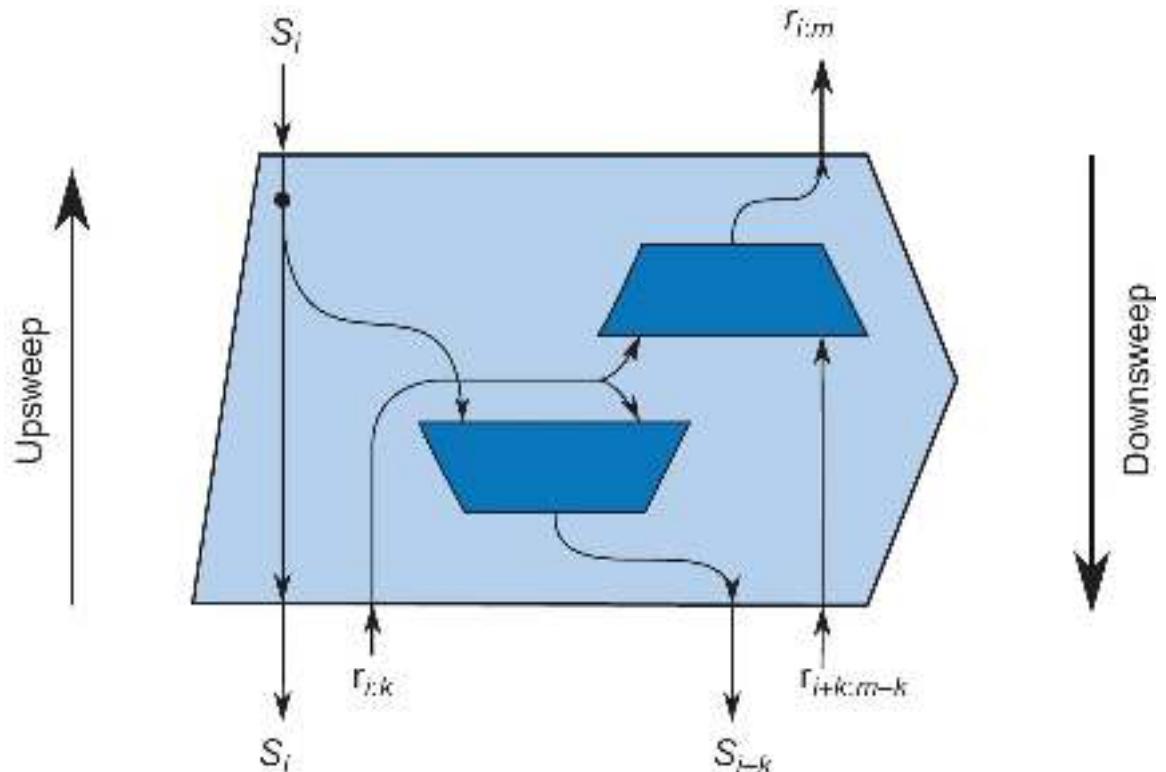
$$r \downarrow i:m = r \downarrow i:k \oplus r \downarrow i+k:m-k$$



Implementing Scan with Fork-Join

- During the downsweep, each node computes subscans of the form:

$$s \downarrow i = \text{initial} \oplus r \downarrow 0:i \quad \text{and} \quad s \downarrow i+k = s \downarrow i \oplus r \downarrow i:k$$





Pipeline Pattern

Parallel Computing

CIS 410/510

Department of Computer and Information Science



UNIVERSITY OF OREGON

Outline

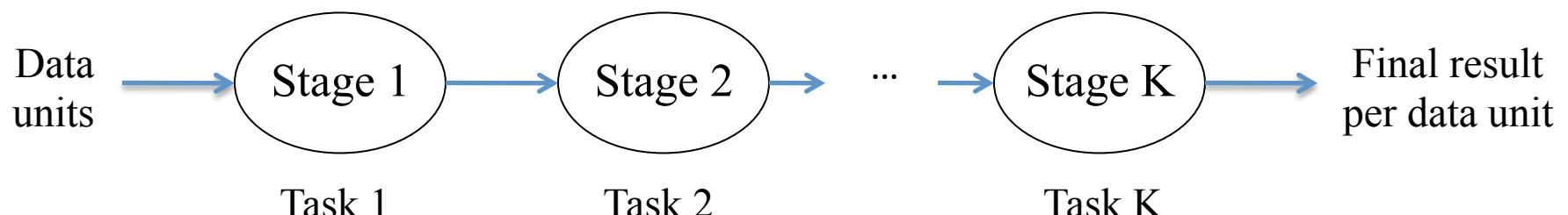
- What is the pipeline concept?
- What is the pipeline pattern?
- Example: Bzip2 data compression
- Implementation strategies
- Pipelines in TBB
- Pipelines in Cilk Plus
- Example: parallel Bzip2 data compression
- Mandatory parallelism vs. optional parallelism

Pipeline

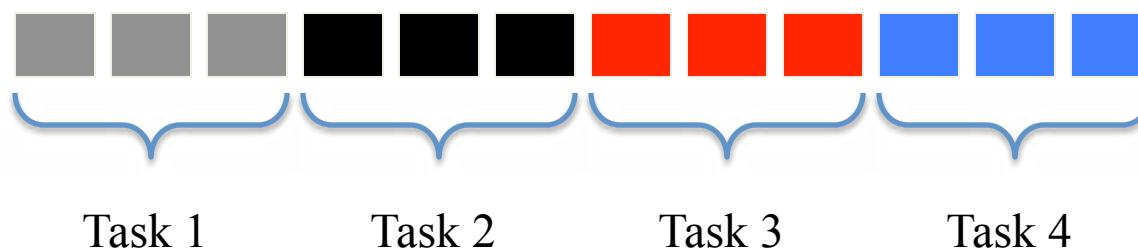
- A pipeline is a linear sequence of stages
- Data flows through the pipeline
 - From Stage 1 to the last stage
 - Each stage performs some task
 - ◆ uses the result from the previous stage
 - Data is thought of as being composed of units (items)
 - Each data unit can be processed separately in pipeline
- Pipeline computation is a special form of *producer-consumer* parallelism
 - Producer tasks output data ...
 - ... used as input by consumer tasks

Pipeline Model

- Stream of data operated on by succession of tasks
- Each task is done in a separate stage

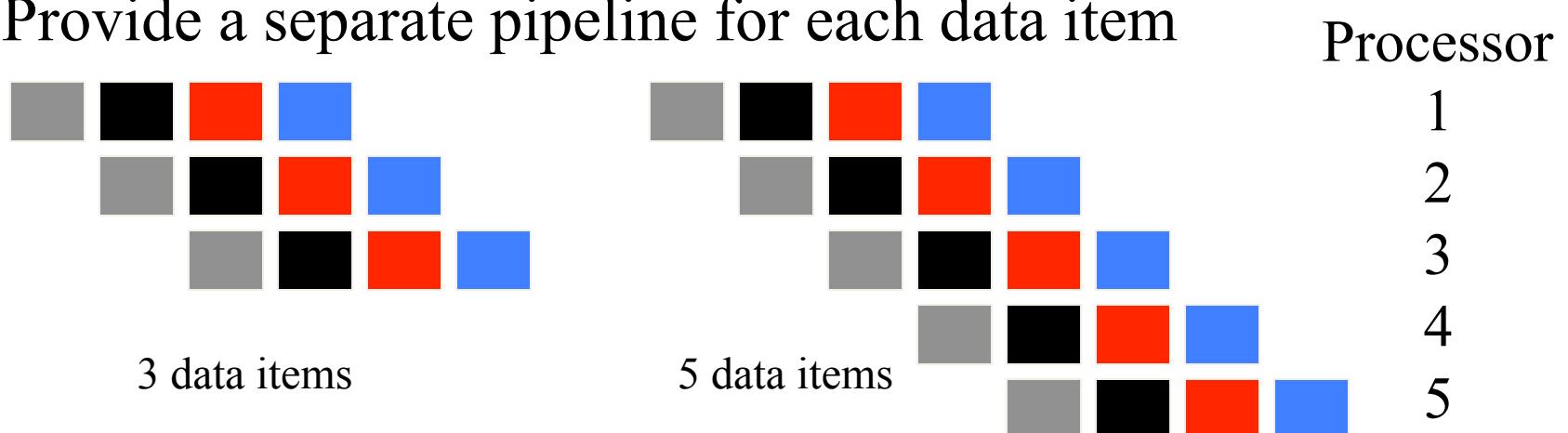


- Consider 3 data units and 4 tasks (stages) 
 - Sequential pipeline execution (no parallel execution)



Where is the Concurrency? (Serial Pipeline)

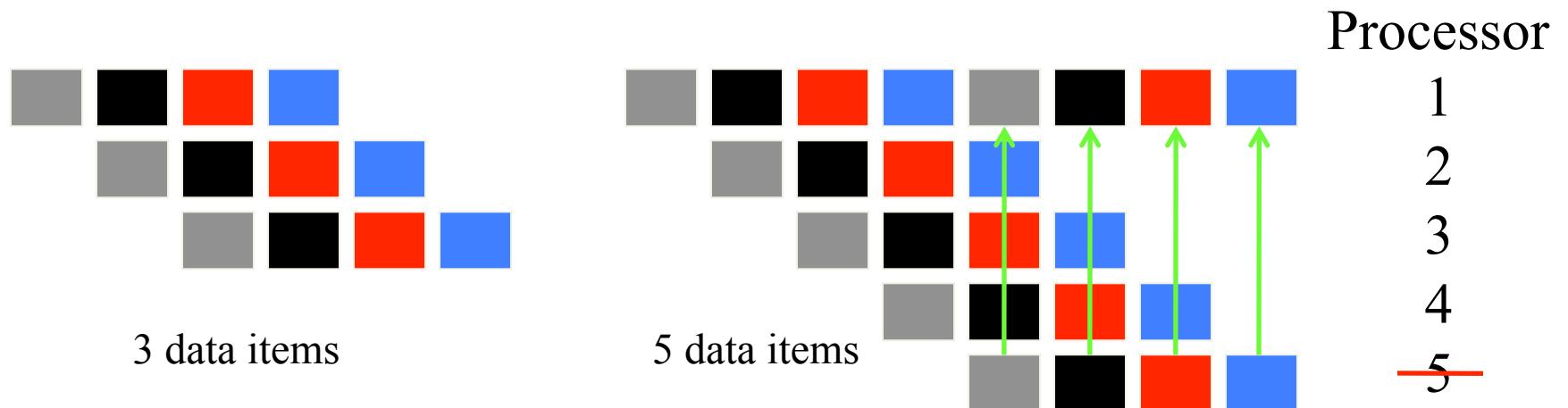
- Pipeline with serial stages
 - Each stage runs serially (i.e., can not be parallel)
 - Can not parallelize the tasks
- What can we run in parallel?
 - Think about data parallelism
 - Provide a separate pipeline for each data item



- What do you notice as we increase # data items?

Parallelizing Serial Pipelines

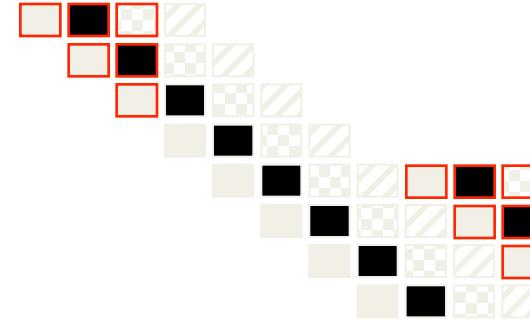
- # tasks limits the parallelism with serial tasks



- Two parallel execution choices:
 - 1) processor executes the entire pipeline
 - 2) processor assigned to execute a single task
- Which is better? Why?

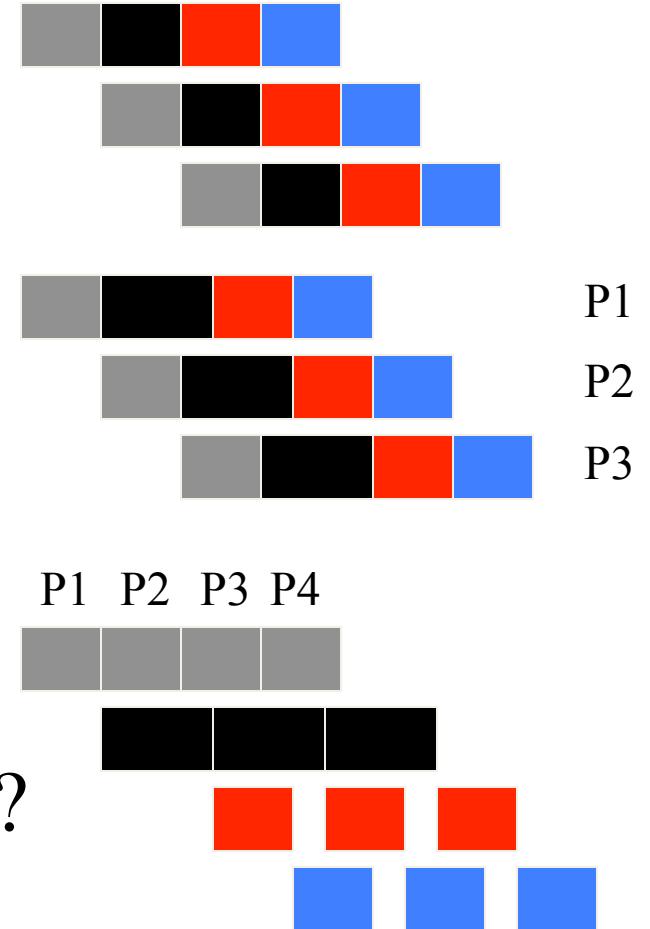
Pipeline Performance

- N data and T tasks
- Each task takes unit time t
- Sequential time = $N*T*t$
- Parallel pipeline time = start + finish + $(N-2T)/T * t$
= $O(N/T)$ (for $N \gg T$)
- Try to find a lot of data to pipeline
- Try to divide computation in a lot of pipeline tasks
 - More tasks to do (longer pipelines)
 - Break a larger task into more (shorter) tasks to do
- Interested in pipeline throughput



Pipeline Performance

- N data and T tasks
- Suppose the tasks execution times are non-uniform
- Suppose a processor is assigned to execute a task
- What happens to the throughput?
- What limits performance?
- Slowest stage limits throughput ... Why?

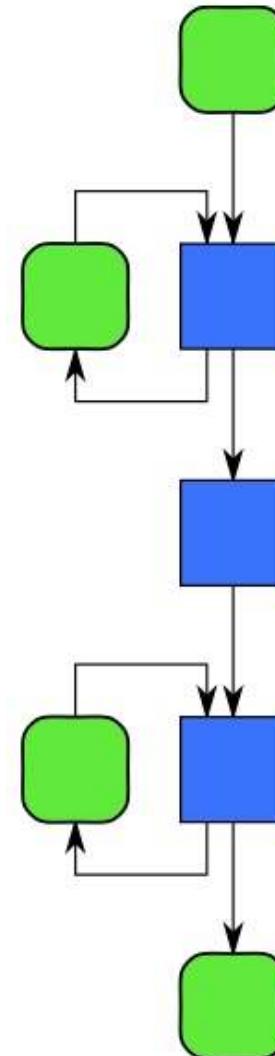


Pipeline Model with Parallel Stages

- What if we can parallelize a task?
- What is the benefit of making a task run faster?
- Book describes 3 kinds of stages (Intel TBB):
 - Parallel: processing incoming items in parallel
 - Serial out of order: process items 1 at a time (arbitrary)
 - Serial in order: process items 1 at a time (in order)

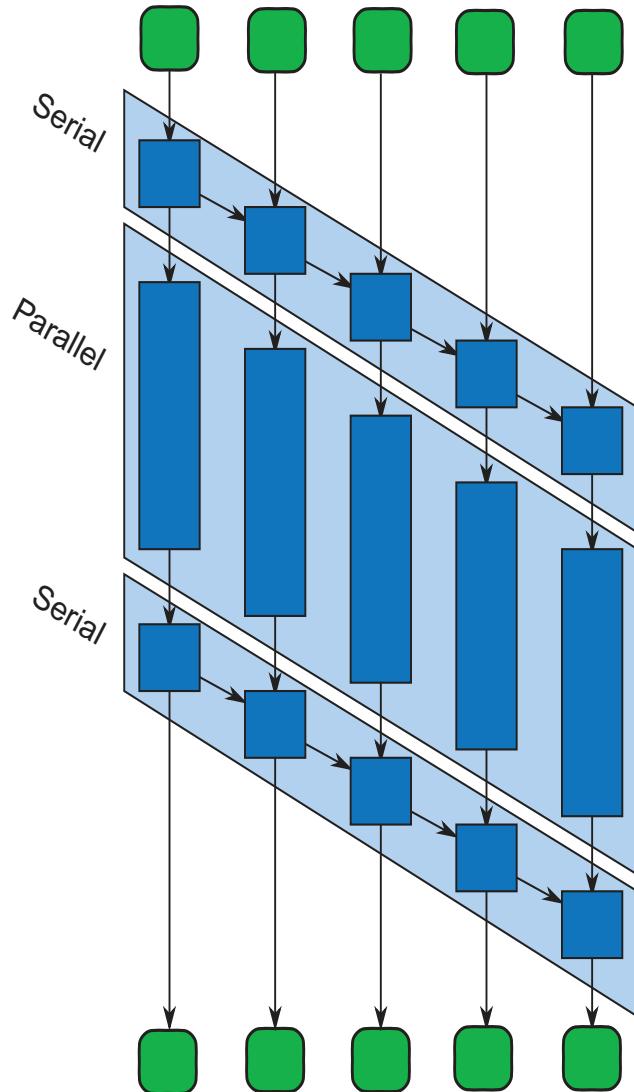
Serial-Parallel-Serial Pipeline Pattern

- Simplest common sequence of stages for a parallel pipeline
- Serial stages are in order
- Feedback in serial stage indicates that data items are processes in order
- Lack of feedback in parallel stage means data items can be processed in parallel



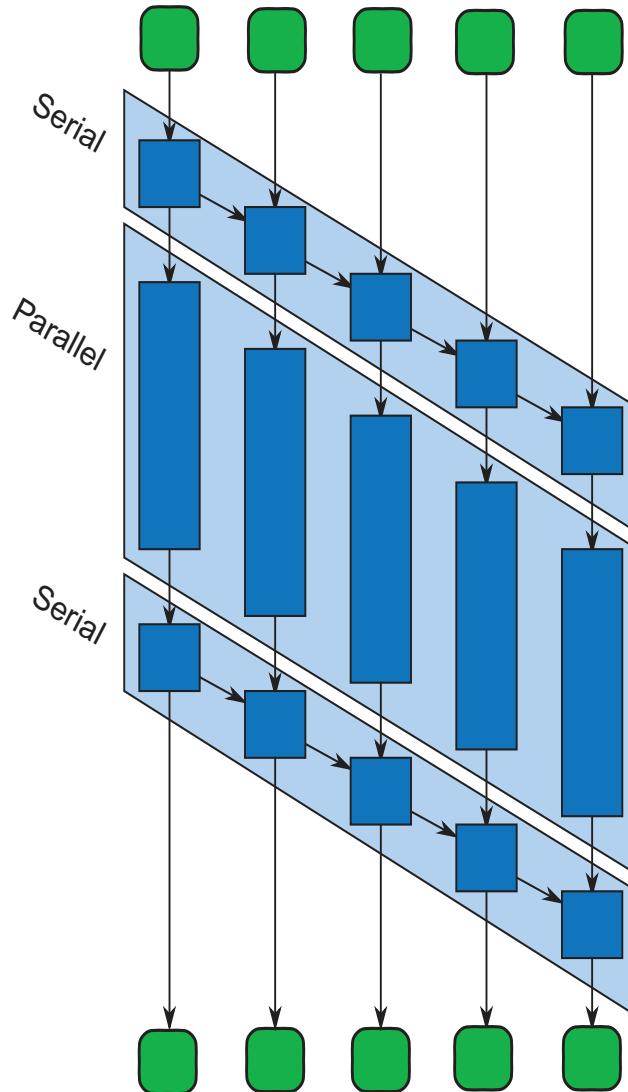
Parallel Pipeline Pattern

- A *pipeline* is composed of several computations called *stages*
 - Parallel stages run independently for each item
 - Serial stages must wait for each item in turn



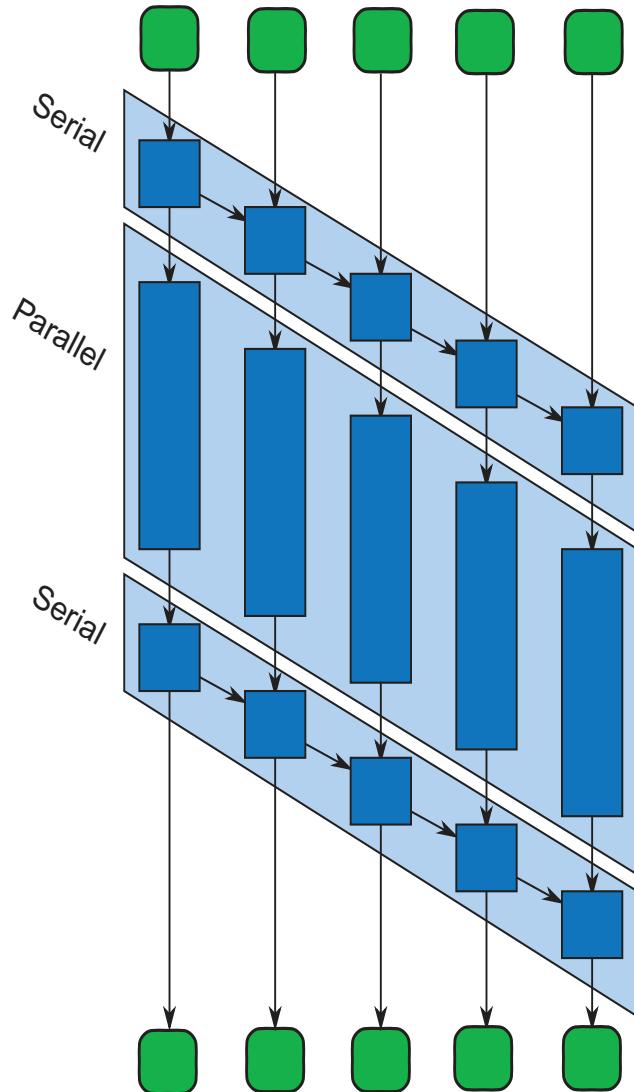
Parallel Pipeline Pattern

- Advantages:
 - Conceptually simple
 - Allows for modularity
 - Parallelizes as much as possible, even when some stages are serial, by overlapping
 - Accommodates I/O as serial stages



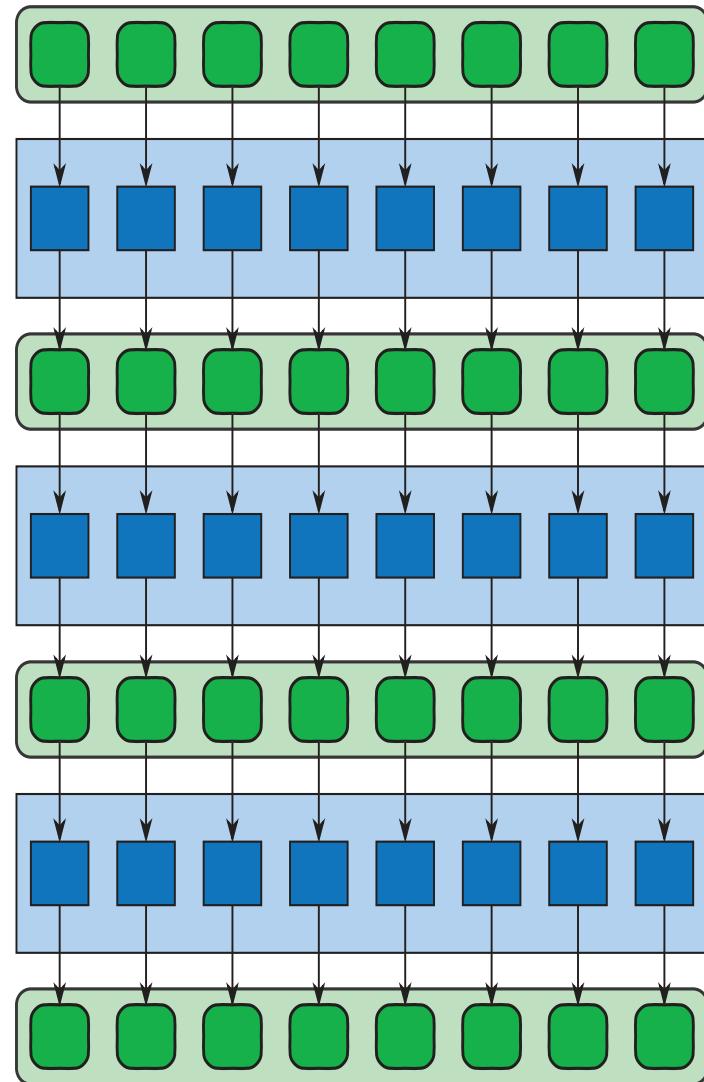
Parallel Pipeline Pattern

- Disadvantages:
 - Serial computation is still a bottleneck
 - Somewhat difficult to implement well from scratch



Combining Maps

- Map operations are often performed in sequence
- Can sometimes optimize this as one big map
- Not always feasible
 - Steps may be in different modules or libraries
 - There may be serial operations interleaved



Example: Bzip2 Data Compression

- `bzip2` utility provides general-purpose data compression
 - Better compression than `gzip`, but slower
- The algorithm operates in blocks
 - Blocks are compressed independently
 - Some pre- and post-processing must be done serially

Three-Stage Pipeline for Bzip2

- Input (serial)
 - Read from disk
 - Perform run-length encoding
 - Divide into blocks
- Compression (parallel)
 - Compress each block independently
- Output (serial)
 - Concatenate the blocks at bit boundaries
 - Compute CRC
 - Write to disk

Implementation Strategies

- Stage-bound workers
 - Each stage has a number of workers
 - ◆ serial stages have only one
 - Each worker takes a waiting item, performs work, then passes item to the next stage
- Essentially the same as map
 - Simple, but no data locality for each item

Stage-Bound Workers

First, each worker grabs input and begins processing it

Suppose this one finishes first

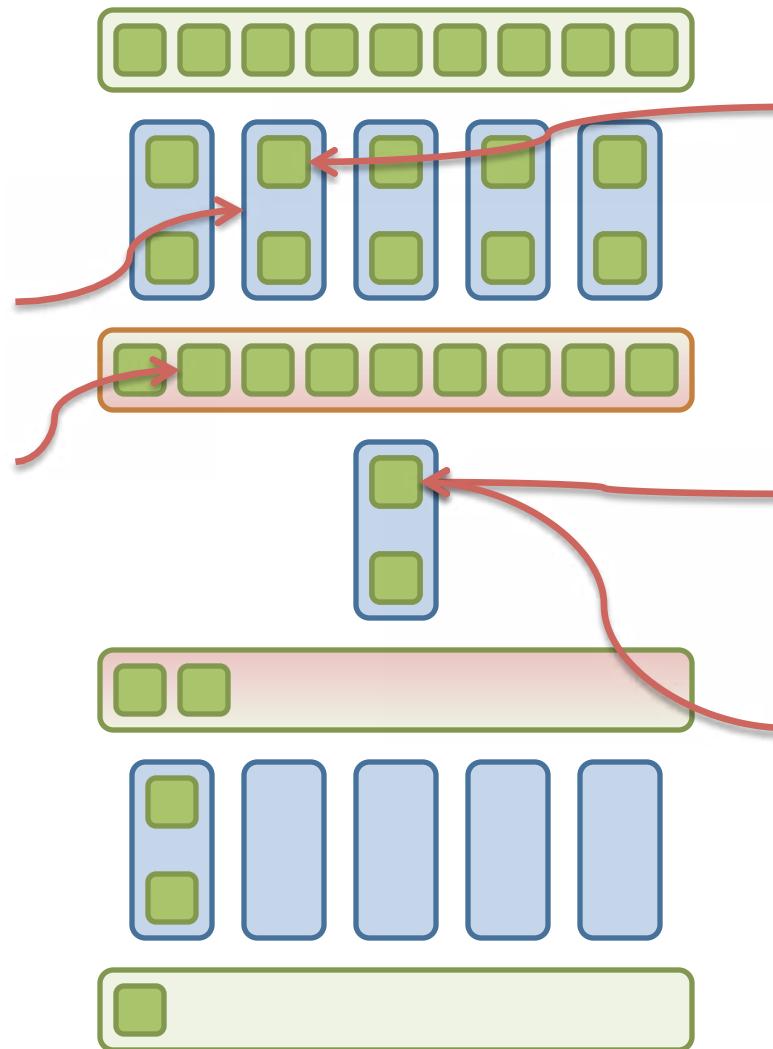
The item gets passed to the serial stage

Since it's out of order, it must wait to be processed

Meanwhile, the finished worker grabs more input

The serial stage accepts the first item

Now that the first item is processed, the second one can enter the serial stage



Implementation Strategies

- Item-bound workers
 - Each worker handles an item at a time
 - Worker is responsible for item through whole pipeline
 - On finishing last stage, loops back to beginning for next item
 - More complex, but has much better data locality for items
 - ◆ Each item has a better chance of remaining in cache throughout pipeline
 - Workers can get stuck waiting at serial stages

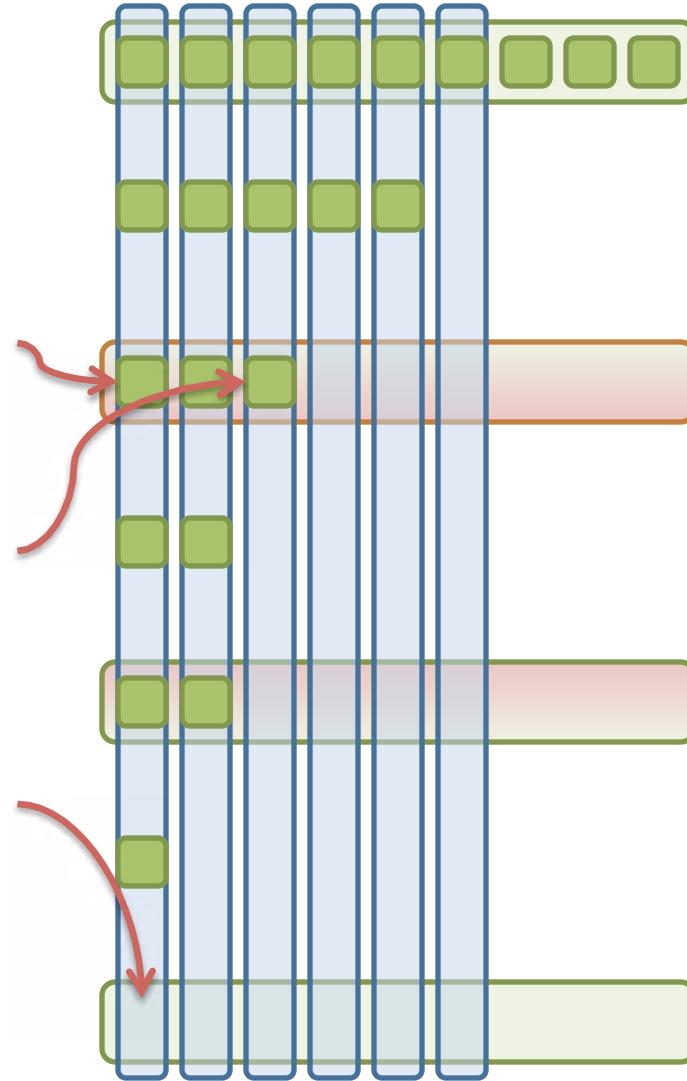
Item-Bound Workers

Each worker gets an item, which it carries through the pipeline

If an item arrives at a serial stage in order, the worker continues

Otherwise, it must block until its turn comes

When an item reaches the end, its worker starts over at the first stage



Implementation Strategies

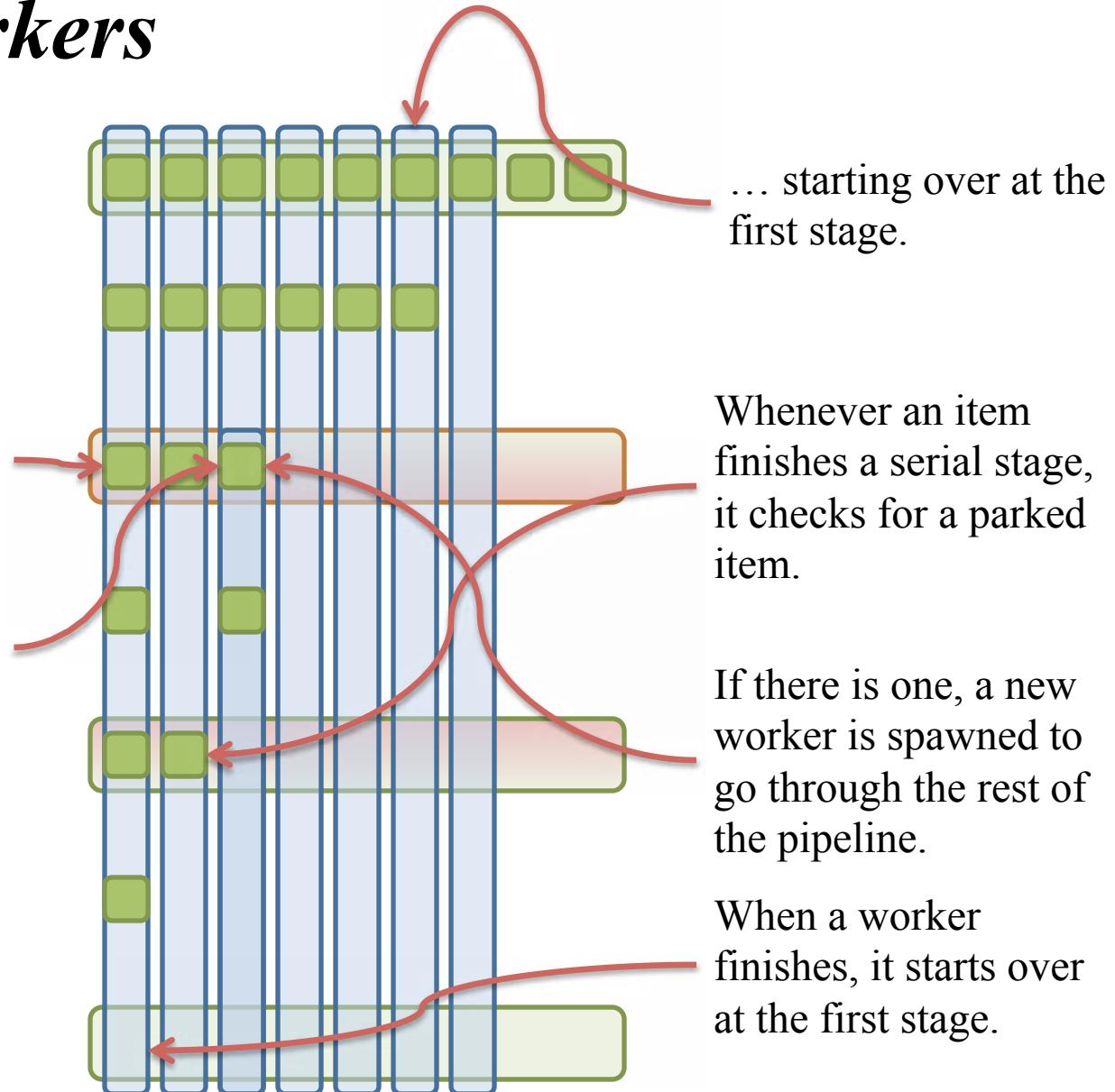
- Hybrid (as implemented in TBB)
 - Workers begin as item-bound
 - When entering a serial stage, the worker checks whether it's ready to process the item now
 - ◆ If so, the worker continues into the stage
 - ◆ Otherwise, it *parks* the item, leaving it for another worker, and starts over
 - When leaving a serial stage, the worker checks for a parked item, spawning a new worker to handle it
 - Retains good data locality without requiring workers to block at serial stages
 - No locks needed; works with greedy schedulers

Hybrid Workers

Each worker gets an item, which it intends to carry through the pipeline.

If an item arrives at a serial stage in order, its worker continues.

Otherwise, the worker “parks” the item and abandons it ...



Pipelines in TBB

- Built-in support from the `parallel_pipeline` function and the `filter_t` class template
- A `filter_t<X, Y>` takes in type X and produces Y
 - May be either a serial stage or a parallel stage
- A `filter_t<X, Y>` and a `filter_t<Y, Z>` combine to form a `filter_t<X, Z>`
- `parallel_pipeline()` executes a `filter_t<void, void>`

Pipelines in TBB: Example Code

- Three-stage pipeline with serial stages at the ends and a parallel stage in the middle
- Here, **f** is a function that returns successive items of type **T** when called, eventually returning **NULL** when done
 - Might not be thread-safe
- **g** comprises the middle stage, mapping each item of type **T** to one item of type **U**
 - Must be thread-safe
- **h** receives items of type **U**, in order
 - Might not be thread-safe

```
1 void tbb_sps_pipeline( size_t ntoken ) {
2     tbb::parallel_pipeline(
3         ntoken,
4         tbb::make_filter<void,T>(
5             tbb::filter::serial_in_order,
6             [&]( tbb::flow_control& fc ) -> T{
7                 T item = f();
8                 if( !item ) fc.stop();
9                 return item;
10            }
11        ) &
12        tbb::make_filter<T,U>(
13            tbb::filter::parallel,
14            g
15        ) &
16        tbb::make_filter<U,void>(
17            tbb::filter::serial_in_order,
18            h
19        )
20    );
21 }
```

Pipelines in TBB: Example Code

- Note the **ntoken** parameter to **parallel_pipeline**
 - Sets a cap on the number of items that can be in processing at once
 - Keeps parked items from accumulating to where they eat up too much memory
 - Space is now bound by **ntoken** times the space used by serial execution

```
1 void tbb_sps_pipeline( size_t ntoken ) {
2     tbb::parallel_pipeline(
3         ntoken,
4         tbb::make_filter<void,T>(
5             tbb::filter::serial_in_order,
6             [&]( tbb::flow_control& fc ) -> T{
7                 T item = f();
8                 if( !item ) fc.stop();
9                 return item;
10            }
11        ) &
12        tbb::make_filter<T,U>(
13            tbb::filter::parallel,
14            g
15        ) &
16        tbb::make_filter<U,void>(
17            tbb::filter::serial_in_order,
18            h
19        )
20    );
21 }
```

Pipelines in Cilk Plus

- No built-in support for pipelines
- Implementing by hand can be tricky
 - Can easily fork to move from a serial stage to a parallel stage
 - But can't simply join to go from parallel back to serial, since workers must proceed in the correct order
 - Could gather results from parallel stage in one big list, but this reduces parallelism and may take too much space

Pipelines in Cilk Plus

- Idea: A reducer can store sub-lists of the results, combining adjacent ones when possible
 - By itself, this would only implement the one-big-list concept
 - However, whichever sub-list is farthest left can process items immediately
 - ◆ the list may not even be stored as such; can “add” items to it simply by processing them
 - This way, the serial stage is running as much as possible
 - Eventually, the leftmost sub-list comprises all items, and thus they are all processed

Pipelines in Cilk Plus: Monoids

- Each view in the reducer has a sub-list and an `is_leftmost` flag
- The views are then elements of two monoids*
 - The usual list-concatenation monoid (a.k.a. the *free monoid*), storing the items
 - A monoid* over Booleans that maps $x \otimes y$ to x , keeping track of which sub-list is leftmost
 - ◆ *Not quite actually a monoid, since a monoid has to have an identity element I for which $I \otimes y$ is always y
 - ◆ But close enough for our purposes, since the only case that would break is `false` \otimes `true`, and the leftmost view can't be on the right!
- Combining two views then means concatenating adjacent sub-lists and taking the left `is_leftmost`

Pipelines in Cilk Plus: Example Code

- Thus we can implement a serial stage following a parallel stage by using a reducer to mediate them
- We call this a *consumer reducer*, calling the class template `reducer_consume`

```
1 #include <cilk/reducer.h>
2 #include <list>
3 #include <cassert>
4
5 template<typename State, typename Item>
6 class reducer_consume {
7 public:
8     // Function that consumes an Item to update a State object
9     typedef void (*func_type)(State*,Item);
10 private:
11     struct View {
12         std::list<Item> items;
13         bool is_leftmost;
14         View( bool leftmost=false ) : is_leftmost(leftmost) {}
15         ~View() {}
16     };
17
18     struct Monoid: cilk::monoid_base<View> {
19         State* state;
20         func_type func;
21         void munch( const Item& item ) const {
22             func(state,item);
23         }
24         void reduce(View* left, View* right) const {
25             assert( !right->is_leftmost );
26             if( left->is_leftmost )
27                 while( !right->items.empty() ) {
28                     munch(right->items.front());
29                     right->items.pop_front();
30                 }
31             else
32                 left->items.splice( left->items.end(), right->items );
33         }
34         Monoid( State* s, func_type f ) : state(s), func(f) {}
35     };
}
```

Pipelines in Cilk Plus: Example Code

- A `View` instance is an element of the (not-quite-)monoid.

```
1 #include <cilk/reducer.h>
2 #include <list>
3 #include <cassert>
4
5 template<typename State, typename Item>
6 class reducer_consume {
7 public:
8     // Function that consumes an Item to update a State object
9     typedef void (*func_type)(State*,Item);
10 private:
11     struct View {
12         std::list<Item> items;
13         bool is_leftmost;
14         View( bool leftmost=false ) : is_leftmost(leftmost) {}
15         ~View() {};
16     };
17
18     struct Monoid: cilk::monoid_base<View> {
19         State* state;
20         func_type func;
21         void munch( const Item& item ) const {
22             func(state,item);
23         }
24         void reduce(View* left, View* right) const {
25             assert( !right->is_leftmost );
26             if( left->is_leftmost )
27                 while( !right->items.empty() ) {
28                     munch(right->items.front());
29                     right->items.pop_front();
30                 }
31             else
32                 left->items.splice( left->items.end(), right->items );
33         }
34         Monoid( State* s, func_type f ) : state(s), func(f) {}
35     };
}
```

Pipelines in Cilk Plus: Example Code

- The **Monoid** class implements the **reduce** function.
 - The leftmost sub-list is always empty; rather than add items to it, we process all of them immediately
- The **func** field holds the function implementing the serial stage
- The **state** field is always passed to **func** so that the serial stage can be stateful

```
1 #include <cilk/reducer.h>
2 #include <list>
3 #include <cassert>
4
5 template<typename State, typename Item>
6 class reducer_consume {
7 public:
8     // Function that consumes an Item to update a State object
9     typedef void (*func_type)(State*,Item);
10 private:
11     struct View {
12         std::list<Item> items;
13         bool is_leftmost;
14         View( bool leftmost=false ) : is_leftmost(leftmost) {}
15         ~View() {}
16     };
17
18     struct Monoid: cilk::monoid_base<View> {
19         State* state;
20         func_type func;
21         void munch( const Item& item ) const {
22             func(state,item);
23         }
24         void reduce(View* left, View* right) const {
25             assert( !right->is_leftmost );
26             if( left->is_leftmost )
27                 while( !right->items.empty() ) {
28                     munch(right->items.front());
29                     right->items.pop_front();
30                 }
31             else
32                 left->items.splice( left->items.end(), right->items );
33         }
34         Monoid( State* s, func_type f ) : state(s), func(f) {}
35     };
}
```

Pipelines in Cilk Plus: Example Code

- To use the consumer reducer, the parallel stage should finish by invoking `consume` with the item
- If this worker has the leftmost view, the item will be processed immediately; otherwise it is stored for later
 - Similar to the hybrid approach, but with less control
- Following a `cilk_sync`, all items will have been processed
 - Including the implicit sync at the end of a function or `cilk_for` loop

```
36     cilk::reducer<Monoid> impl;
37
38
39 public:
40     reducer_consume( State* s, func_type f ) :
41         impl(Monoid(s,f), /*leftmost=*/true)
42     {}
43
44     void consume( const Item& item ) {
45         View& v = impl.view();
46         if( v.is_leftmost )
47             impl.monoid().munch( item );
48         else
49             v.items.push_back(item);
50     }
51 }
```

EXAMPLE: BZIP2 COMPRESSION

Parallel Bzip2 in Cilk Plus

- The `while` loop comprises the first stage
 - Reads in the file, one block at a time, spawning a call to `SecondStage` for each block
- `SecondStage` compresses its block, then passes it to the consumer reducer
 - Reducer preconfigured (line 17) to invoke `ThirdStage`
- `ThirdStage` always receives the blocks in order, so it outputs blocks as it receives them

```
1 void SecondStage( EState* s, reducer_consume<OutputState, EState*>& sink ) {
2     if( s->nblock )
3         CompressOneBlock(s);
4     sink.consume( s );
5 }
6
7 void ThirdStage( OutputState* out_state, EState* s ) {
8     if( s->nblock )
9         out_state->putOneBlock(s);
10    FreeEState(s);
11 }
12
13 int BZ2_compressFile(FILE *stream, FILE *zStream, int
14                      blockSize100k, int verbosity, int workFactor) throw()
15 {
16     @{\rm ...}
17     TmpState in_state;
18     "OutputState out_state( zStream );
19     reducer_consume<OutputState,EState*> sink(&out_state, ThirdStage);
20     while( !feof(stream) && !ferror(stream) ) {
21         EState *s = BZ2_bzCompressInit(blockSize100k, verbosity, workFactor);
22         in_state.getOneBlock(stream,s);
23         cilk_spawn SecondStage(s, sink);
24     };
25     cilk_sync;
26     ""@{\rm ...}
27 }
```

Mandatory vs. Optional Parallelism

- Consider a 2-stage (producer-consumer) pipeline
 - Produces puts items into a buffer for consumer
- There is no problem if the producer and consumer run in parallel
- The serial execution is tricky because buffer can fill up and block progress of the producer
 - Similar situation with the consumer
 - Producer and consumer must interleave to guarantee progress
- Restricting the kinds of pipelines that can be built
 - No explicit waiting because a stage invoked only when its input item is ready and must emit exactly 1 output
 - Going from parallel to serial require buffering
- Mandatory parallelism forces the system to execute operations in parallel whereas optional does not require it