

## Lecture Schedule

[Course overview](#)

[Computational science algorithms: parallelism and locality](#)

[Graph algorithms: parallelism and locality](#)

[Measurement: execution time and hardware counters](#)

[DAG scheduling](#)

[Cache models](#)

[MMM and ATLAS](#)

[Vectorization \(courtesy Professor David Padua, UIUC\)](#)

[Cache-coherent shared-memory multiprocessors](#)

[Shared-memory programming: pThreads and OpenMP](#)

[Memory consistency\(I\)](#)

[Memory consistency \(II\)](#)

[GPU programming](#)

[CUDA](#)

[Message-passing and MPI](#)

## CS 377P: Programming for Performance



## Administration

- **Instructor:**
  - Keshav Pingali (Professor, CS department & ICES)
    - 4.126 Peter O'Donnell Building (POB)
    - Email: pingali@cs.utexas.edu
- **TA:**
  - Tongliang Liao (Grad student, CS department)
    - Email: xkszltl@gmail.com

## Prerequisites

- **Basic computer architecture course**
  - (e.g.) PC, ALU, cache, memory, instruction-level parallelism (ILP)
- **Basic calculus and linear algebra**
  - differential equations and matrix operations
- **Software maturity**
  - assignments will be in C/C++ on Linux computers
  - ability to write medium-sized programs (~1000 lines)
- **Self-motivation**
  - willingness to experiment with systems

## Coursework

- **6 programming projects**
  - These will be more or less evenly spaced through the semester
  - Some assignments will also have short questions
- **One mid-semester exam**
  - March 21<sup>st</sup>, 2017
- **Final exam**

## Text-book for course

No official book for course

This book is a useful reference.

"Parallel programming in C with MPI and OpenMP", Michael Quinn, McGraw-Hill Publishers. ISBN 0-07-282256-2

Lots of material on the web

## What this course is not about

- This is not a tools/libraries course
  - We will use a small number of tools and micro-benchmarks to understand performance, but this is not a course on how to use tools and libraries
- This is not a clever hacks course
  - We are interested in general scientific principles for performance programming, not in squeezing out every last cycle for somebody's favorite program

## What this course IS about

- Architects invent many hardware features for boosting program performance
- Usually, software can benefit from these features only if it is carefully written to exploit them
- Our agenda in CS 377P:
  - Understand key performance-critical architectural features in modern computers
  - Develop general principles and techniques that can guide us in writing programs to exploit these features
- Two major concerns:
  - Exploiting multicore/manycore processors: **parallelism**
  - Exploiting the memory hierarchy: **locality**

## Parallelism

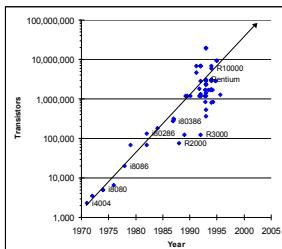
- Fundamental ongoing change in computer industry
- Moore's law(s): two versions
  1. Number of transistors on chip double every 1.5 years
    - Transistors used to build complex, superscalar processors, deep pipelines, etc. to exploit instruction-level parallelism (ILP)
  2. Processor frequency doubles every 1.5 years
    - Speed goes up by factor of 10 roughly every 5 years
    - Moore did not say this in his paper
- **Many programs ran faster if you just waited a while.**
- Fundamental change
  - Micro-architectural innovations for exploiting ILP are reaching limits
  - Clock speeds are not increasing any more because of power problems
- **Programs will not run any faster if you wait.**
- Let us understand why.



Gordon Moore

## (1) Micro-architectural approaches to improving processor performance

- Add functional units
  - Superscalar is known territory
  - Diminishing returns for adding more functional blocks
  - Alternatives like VLIW are successful only in embedded space
- Wider data paths
  - Increasing bandwidth between functional units in a core makes a difference
    - Such as comprehensive 64-bit design, but then what?

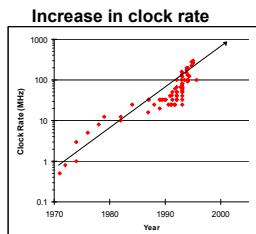


## (1) Micro-architectural approaches (contd.)

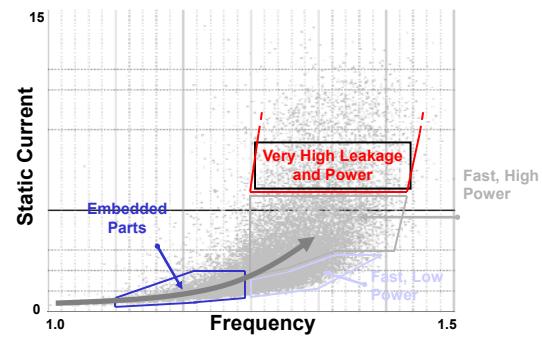
- Deeper pipeline
  - Deeper pipeline buys frequency at expense of increased branch mis-prediction penalty and cache miss penalty
  - Deeper pipelines => higher clock frequency => more power
  - Industry converging on middle ground... 9 to 11 stages
    - Successful RISC CPUs are in the same range
- More cache
  - More cache buys performance until working set of program fits in cache
  - Exploiting caches requires help from programmer/compiler as we will see

## (2) Processor clock speeds

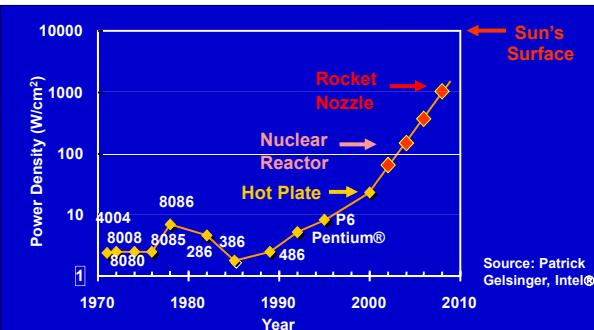
- Old picture:
  - Processor clock frequency doubled every 1.5 years
- New picture:
  - Power problems limit further increases in clock frequency (see next couple of slides)



## (2) Processor clock speeds (contd.)



## (2) Processor clock speeds (contd.)



## Recap

- Old picture:

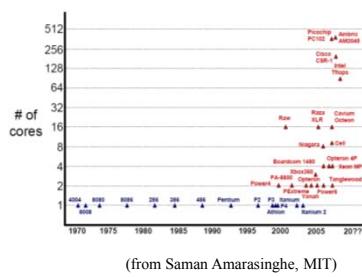
- Moore's law(s):
  1. Number of transistors doubled every 1.5 years
    - Use these to implement micro-architectural innovations for ILP
  2. Processor clock frequency doubled every 1.5 years
    - Many programs ran faster if you just waited a while.

- New picture:

- Moore's law
  1. Number of transistors still double every 1.5 years
    - But micro-architectural innovations for ILP are flat-lining
  2. Processor clock frequencies are not increasing very much
    - Programs will not run faster if you wait a while.
- Questions:
  - Hardware: What do we do with all those extra transistors?
  - Software: How do we keep speeding up program execution?

## One hardware solution: go multicore

- Use semi-conductor tech improvements to build multiple cores without increasing clock frequency
  - does not require micro-architectural breakthroughs
  - non-linear scaling of power density with frequency will not be a problem
- Predictions:
  - from now on, number of cores will double every 1.5 years



## New problem: multi-core software

- More aggregate performance for:
  - Multi-threaded apps (our focus)
  - Transactions: many instances of same app
  - Multi-tasking
- Problem
  - Most apps are not multithreaded
  - Writing multithreaded code increases software costs dramatically
    - factor of 3 for Unreal game engine (Tim Sweeney, Epic games)
- The great multicore software quest: Can we write programs so that performance doubles when the number of cores doubles?
- Very hard problem for many reasons (see later)
  - Amdahl's law
  - Overheads of parallel execution
  - Load balancing
  - .....

"We are the cusp of a transition to multicore, multithreaded architectures, and we still have not demonstrated the ease of programming the move will require... I have talked with a few people at Microsoft Research who say this is also at or near the top of their list [of critical CS research problems]." Justin Rattner, Senior Fellow, Intel

## Amdahl's Law

- Simple observation that shows that unless most of the program can be executed in parallel, the benefits of parallel execution are limited
  - serial portions of program become bottleneck
- Analogy: suppose I go from Austin to Houston at 60 mph, and return infinitely fast. What is my average speed?
  - Answer: 120 mph, not infinity

## Amdahl's Law (details)

- In general, program will have both parallel and serial portions
  - Suppose program has N operations
    - $r \cdot N$  operations in parallel portion
    - $(1-r) \cdot N$  operations in serial portion
- Assume
  - Serial execution requires one time unit per operation
  - Parallel portion can be executed infinitely fast by multicore processor, so it takes zero time to execute.
- Speed-up:  
$$\frac{\text{(execution time on single core)}}{\text{(execution time on multicore)}} = \frac{N}{(1-r) \cdot N} = \frac{1}{1-r}$$
- Even if  $r = 0.9$ , speed-up is only 10.

## Our focus

- Multi-threaded programming
  - also known as *shared-memory* programming
  - application program is decomposed into a number of “threads” each of which runs on one core and performs some of the work of the application: “many hands make light work”
  - threads communicate by reading and writing memory locations (that’s why it is called shared-memory programming)
  - we will use a popular system called [OpenMP](#)
- Key issues:
  - how do we assign work to different threads?
  - how do we ensure that work is more or less equitably distributed among the threads?
  - how do we make sure threads do not step on each other (synchronization)?
  - ....

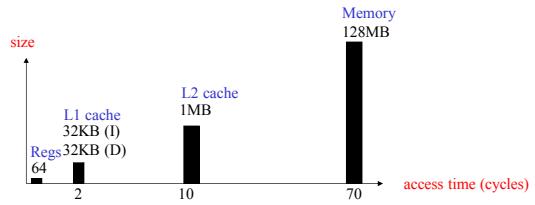
## Distributed-memory programming

- Some application areas such as computational science need more power than will be available in the near future on a multi-core processor
- Solution: connect a bunch of multicore processors together
  - (e.g.) Ranger machine at Texas Advanced Computing Center (TACC): 15,744 processors, each of which has 4 cores
- Must use a different model of parallel programming called
  - *message-passing* (or)
  - *distributed-memory programming*
- Distributed-memory programming
  - units of parallel execution are called [processes](#)
  - processes communicate by sending and receiving messages since they have no memory locations in common
  - most-commonly-used communication library: [MPI](#)
- We will study distributed-memory programming as well and you will get to run programs on Stampede

## Software problem (II): memory hierarchy

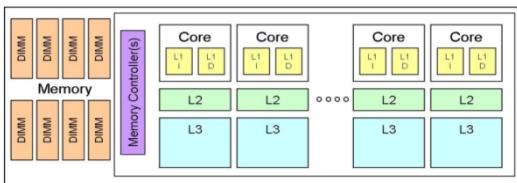
- Complication for parallel software
  - unless software also exploit caches, overall performance is usually poor
  - writing software that can exploit caches also complicates software development

## Memory Hierarchy of SGI Octane



- R10 K processor:
  - 4-way superscalar, 2 fpo/cycle, 195MHz
- Peak performance: 390 Mflops
- Experience: sustained performance is less than 10% of peak
  - Processor often stalls waiting for memory system to load data

## Memory Hierarchy of Power 7



- Eight cores on same socket/chip, 3.6 GHz
- L1 cache: 32KB data, 32KB instruction
- L2 cache: 256 KB, latency is “a few cycles”
- L3 cache: 4\*8 MB/chip, 50 cycles

## Software problem (II)

- Caches are useful only if programs have locality of reference
  - temporal locality: program references to given memory address are clustered together in time
  - spatial locality: program references clustered in address space are clustered in time
- Problem:
  - Programs obtained by expressing most algorithms in the straight-forward way do not have much locality of reference
  - How do we code applications so that they can exploit caches?

## Software problem (II): memory hierarchy

“...The CPU chip industry has now reached the point that instructions can be executed more quickly than the chips can be fed with code and data. Future chip design is memory design. Future software design is also memory design. .... Controlling memory access patterns will drive hardware and software designs for the foreseeable future.”

Richard Sites

## Abstract questions

- Do applications have parallelism?
- If so, what patterns of parallelism are there in common applications?
- Do applications have locality?
- If so, what patterns of locality are there in common applications?
- We will study sequential and parallel algorithms and data structures to answer these questions

## Course content

- Analysis of applications that need high end-to-end performance: study parallelism and locality
  - Computational science applications
  - Big-data processing
- Understanding parallel performance: DAG model of computation, Moore's law, Amdahl's law
- Measurement and the design of computer experiments
- Micro-benchmarks for abstracting performance-critical aspects of computer systems
- Memory hierarchy:
  - caches, virtual memory
  - optimizing programs for memory hierarchies
  - cache-oblivious programming
- .....

## Course content (contd.)

- .....
- Multi-core processors and shared-memory programming
  - pThreads and OpenMP
- GPUs and GPU programming
- Distributed-memory machines and message-passing programming
  - MPI
- Advanced topics:
  - Optimistic parallelism
  - Self-optimizing software
    - ATLAS,FFTW

## Computational Science Algorithms

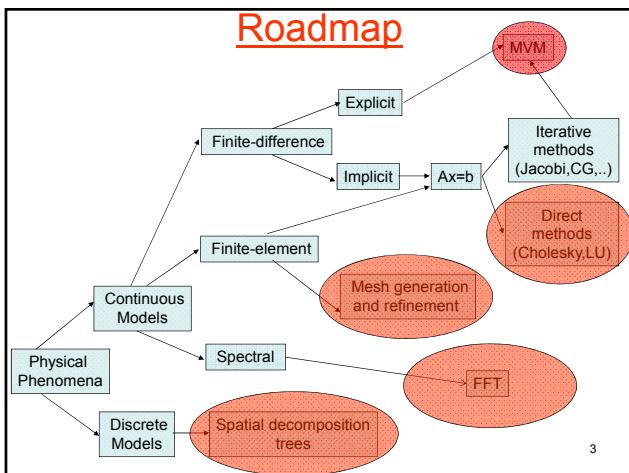
1

### Computational science

- Simulations of physical phenomena
  - fluid flow over aircraft (Boeing 777)
  - fatigue fracture in aircraft bodies
  - evolution of galaxies
  - ...
- Two main approaches
  - continuous models: fields and differential equations (eg. Navier-Stokes equations, Maxwell's equations,...)
  - discrete models: particles and forces (eg. gravitational forces)
- Paradox
  - most differential equations cannot be solved exactly
    - must use numerical techniques that convert calculus problem to matrix computations: **discretization**
    - approximation
  - n-body methods are straight-forward
    - but need to use a lot of bodies to get accuracy
    - must find a way to reduce  $O(N^2)$  complexity of obvious algorithm
    - approximate the contribution of distant bodies
- Motto:
  - "All exact science is dominated by the idea of approximation." Bertrand Russell

2

### Roadmap



3

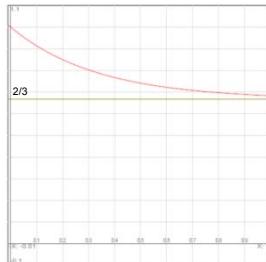
### Organization

- Finite-difference methods
  - ordinary and partial differential equations
  - discretization techniques
    - explicit methods: Forward-Euler method
    - implicit methods: Backward-Euler method
- Finite-element methods
  - mesh generation and refinement
  - weighted residuals
- N-body methods
  - Barnes-Hut
- Key algorithms and data structures
  - matrix computations
    - algorithms
      - MVM and MMM
      - solution of systems of linear equations
        - » direct methods
        - » iterative methods
    - data structures
      - dense and sparse matrices
  - graph computations
    - mesh generation and refinement
    - spatial decomposition trees

4

## Ordinary differential equations

- Consider the ode  
 $u'(t) = -3u(t)+2$   
 $u(0) = 1$
- This is called an initial value problem
  - initial value of  $u$  is given
  - compute how function  $u$  evolves for  $t > 0$
- Using elementary calculus, we can solve this ode exactly  
 $u(t) = 1/3 (e^{-3t} + 2)$



5

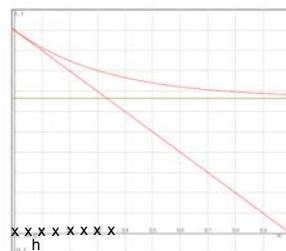
## Problem

- For general ode's, we may not be able to express solution in terms of elementary functions
- In most practical situations, we do not need exact solution anyway
  - enough to compute an approximate solution, provided
    - we have some idea of how much error was introduced
    - we can improve the accuracy as needed
- General solution:
  - convert calculus problem into algebra/arithmetic problem
    - discretization: replace continuous variables with discrete variables
    - in finite differences,
      - time will advance in fixed-size steps:  $t=0, h, 2h, 3h, \dots$
      - differential equation is replaced by difference equation

6

## Forward-Euler method

- Intuition:
  - we can compute the derivative at  $t=0$  from the differential equation  
 $u'(t) = -3u(t)+2$
  - so compute the derivative at  $t=0$  and advance along tangent to  $t=h$  to find an approximation to  $u(h)$
- Formally, we replace derivative with forward difference to get a difference equation
  - $u'(t) \rightarrow (u(t+h) - u(t))/h$
- Replacing derivative with difference is essentially the inverse of how derivatives were probably introduced to you in elementary calculus



7

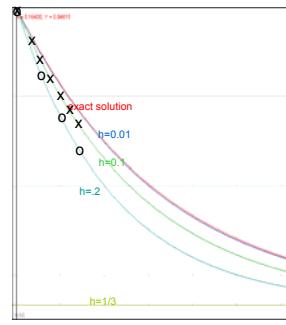
## Back to ode

- Original ode  
 $u'(t) = -3u(t)+2$
- After discretization using Forward-Euler:  
 $(u(t+h) - u(t))/h = -3u(t)+2$
- After rearrangement, we get difference equation  
 $u_f(t+h) = (1-3h)u_f(t)+2h$
- We can now compute values of  $u_f$  at  $t = h, 2h, 3h, \dots$ 
  - $u_f(0) = 1$
  - $u_f(h) = (1-h)$
  - $u_f(2h) = (1-2h+3h^2)$
  - .....

8

## Tabulation

- Numerical solution
  - Choose a value for  $h$
  - Tabulate the values of  $u_i$  at  $t = nh$  for  $n = 0, 1, 2, \dots$ , using the recurrence formula
- Question: how do you choose the step size  $h$ ?
  - Small  $h$  is more accurate but also more computationally intensive
  - If we assume we want to estimate the value of  $u$  at  $t = T$ , we will need  $O(T/h)$  evaluations of the recurrence formula
- Important property of forward-Euler:
  - Numerical solution is stable only if  $h$  is "small enough"
  - If  $h$  is too big, numerical estimate will blow up
  - Recurrence formula is a feedback loop and error introduced at one time step gets amplified by the recurrence formula



9

## Analysis of recurrence formula

- Understanding notions like stability of finite-difference formulas is complex in general
- In this particular case, we can do the analysis easily because we can solve difference equation exactly
- It is not hard to show that if difference equation is
 
$$u_i(t+h) = a \cdot u_i(t) + b$$

$$u_i(0) = 1$$
 the solution is
 
$$u_i(nh) = a^n + b \cdot (1-a^n)/(1-a)$$
- For our difference equation,
 
$$u_i(t+h) = (1-3h)u_i(t) + 2h$$
 the exact solution is
 
$$u_i(nh) = 1/3 \cdot ((1-3h)^n + 2)$$

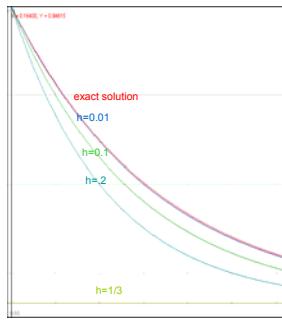
10

## Comparison

- Exact solution
 
$$u(t) = 1/3 (e^{-3t} + 2)$$

$$u(nh) = 1/3(e^{-3nh} + 2)$$
 (at time-steps)
- Forward-Euler solution
 
$$u_i(nh) = 1/3((1-3h)^i + 2)$$
- Use series expansion to compare
 
$$u(nh) = 1/3(1-3nh + 9h^2/2! - n^2h^2/2! + \dots + 2)$$

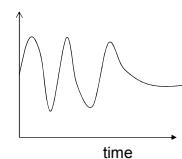
$$u(nh) = 1/3(1-3nh + n(n-1)/2 \cdot 9h^2 + \dots + 2)$$
 So error =  $O(nh^2)$
- Conclusion:
  - error per time step (local error) =  $O(h^2)$
  - error at time  $nh$  =  $O(nh^2)$
- In general, Forward-Euler converges only if time step is "small enough"



11

## Choosing time step

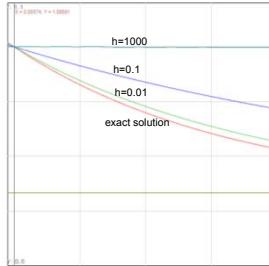
- Time-step needs to be small enough to capture highest frequency phenomenon of interest
- Nyquist's criterion
  - sampling frequency must be at least twice highest frequency to prevent aliasing
  - for most finite-difference formulas, you need sampling frequencies (much) higher than the Nyquist criterion
- In practice, most functions of interest are not band-limited, so use
  - insight from application or
  - reduce time-step repeatedly till changes are not significant
- Fixed-size time-step can be inefficient if frequency varies widely over time interval
  - other methods like finite-elements permit variable time-steps as we will see later



12

## Backward-Euler method

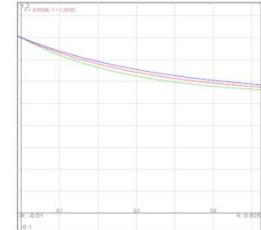
- Replace derivative with backward difference  
 $y'(t) \rightarrow (y(t) - y(t-h))/h$
- For our ode, we get  
 $y_b(t) - y_b(t-h)/h = -3y_b(t) + 2$   
which after rearrangement  
 $y_b(t) = (2 + y_b(t-h))/(1+3h)$
- As before, this equation is simple enough that we can write down the exact solution:  
 $y_b(nh) = ((1/(1+3h))^n + 2)/3$
- Using series expansion, we get  
 $y_b(nh) = (1-3nh + (-n(n-1)/2)9h^2 + ... + 2)/3$   
 $y_b(nh) = (1-3nh + 9/2 n^2h^2 + 9/2 nh^2 + ... + 2)/3$   
So error =  $O(nh^2)$  (for any value of h)



13

## Comparison

- Exact solution  
 $y(t) = 1/3 (e^{-3t} + 2)$   
 $y(nh) = 1/3(e^{-3nh} + 2)$  (at time-steps)
- Forward-Euler solution  
 $y_f(nh) = 1/3(1-3h)^n + 2$   
error =  $O(nh^2)$  (provided  $h < 2/3$ )
- Backward-Euler solution  
 $y_b(nh) = 1/3 ((1/(1+3h))^n + 2)$   
error =  $O(nh^2)$  ( $h$  can be any value you want)
- Many other discretization schemes have been studied in the literature
  - Runge-Kutta
  - Crank-Nicolson
  - Upwind differencing
  - ...

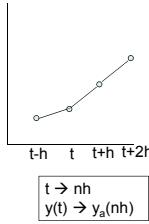


Red: exact solution  
Blue: Backward-Euler solution ( $h=0.1$ )  
Green: Forward-Euler solution ( $h=0.1$ )

14

## Higher-order difference formulas

- First derivatives:
  - Forward-Euler:  $y'(t) \rightarrow y_b(t+h) - y_b(t)/h$
  - Backward-Euler:  $y'(t) \rightarrow y_b(t) - y_b(t-h)/h$
  - Centered:  $y'(t) \rightarrow y_c(t+h) - y_c(t-h)/2h$
- Second derivatives:
  - Forward:  $y''(t) \rightarrow (y_b(t+2h) - y_b(t+h)) - (y_b(t+h) - y_b(t))/h^2$   
 $= y_b(t+2h) - 2y_b(t+h) + y_b(t)/h^2$
  - Backward:  $y''(t) \rightarrow y_b(t) - 2y_b(t-h) + y_b(t-2h)/h^2$
  - Centered:  $y''(t) \rightarrow y_c(t+h) - 2y_c(t) + y_c(t-h)/h^2$
- Explicit methods: derivatives at t are approximated by values at  $t, t+h, t+2h$ , etc.
  - Forward-Euler is example
  - Key operation is matrix-vector product
- Implicit methods: derivatives at t are approximated using some values before t such  $t-h, t-2h, \dots$ 
  - Backward-Euler, centered differences are examples
  - Key operation is linear solve



15

## Finite-differences: partial differential equations

16

### Finite-difference methods for solving partial differential equations

- Basic ideas carry over unchanged
- Example: 2-d heat equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y)$$

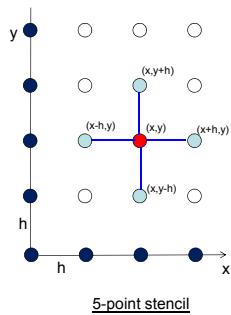
- Assume temperature at boundary is fixed
- Discretize domain using a regular NxN grid of pitch h
- Approximate derivatives as centered differences

$$\frac{\partial^2 u}{\partial y^2} \rightarrow \frac{u(x,y+h) - u(x,y) - u(x,y-h)}{h^2}$$

$$\frac{\partial^2 u}{\partial x^2} \rightarrow \frac{u(x+h,y) - u(x,y) - u(x-h,y)}{h^2}$$

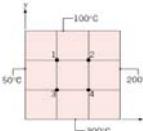
- So we get a system of  $(N-1) \times (N-1)$  difference equations in terms of the unknowns at the  $(N-1) \times (N-1)$  interior points

for all interior points  $(ih,jh)$   
 $u(ih,jh+h) + u(ih,jh-h) + u(ih+h,jh) + u(ih-h,jh) - 4u(ih,jh) = h^2 f(ih,jh)$



17

### Example



$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y)$$

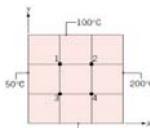
Assume  $f(x,y) = 0$

- Unknown temperatures are  $T_1, T_2, T_3, T_4$
- Discretized equation at point 1:

$$\frac{T_2 - T_1}{h} + \frac{T_1 - 50}{h} + \frac{100 - T_1}{h} + \frac{T_1 - T_3}{h} = 0$$

18

### Example (contd)



$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y)$$

Assume  $f(x,y) = 0$

$$-4T_1 + T_2 + T_3 = -150$$

$$T_1 - 4T_2 + T_4 = -300$$

$$T_1 - 4T_3 + T_4 = -350$$

$$T_2 + T_3 - 4T_4 = -500$$

$$\begin{bmatrix} -4 & 1 & 1 & 0 \\ 1 & -4 & 0 & 1 \\ 1 & 0 & -4 & 1 \\ 0 & 1 & 1 & -4 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{bmatrix} = \begin{bmatrix} -150 \\ -300 \\ -350 \\ -500 \end{bmatrix}$$

Solution:  $T_1 = 119, T_2 = 156, T_3 = 169, T_4 = 206$

How do we solve large systems of linear equations?

19

### General picture: matrix notation

- System of  $(N-1) \times (N-1)$  difference equations in terms of the unknowns at the  $(N-1) \times (N-1)$  interior points

for all interior points  $(ih,jh)$

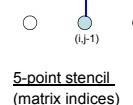
$$u(ih,jh+h) + u(ih,jh-h) + u(ih+h,jh) + u(ih-h,jh) - 4u(ih,jh) = h^2 f(ih,jh)$$

- Matrix notation: use natural order for u's

$$\begin{bmatrix} \dots & \dots & \dots & \dots & \dots \\ \dots & u(i-1,j) & \dots & \dots & \dots \\ \dots & u(i,j-1) & \dots & \dots & \dots \\ \dots & u(i,j) & \dots & \dots & \dots \\ \dots & u(i+1,j) & \dots & \dots & \dots \end{bmatrix} = h^2 f(ih,jh)$$

Pentadiagonal sparse matrix

Matrix is known at compile-time and has simple structure.



20

## Solving linear systems

- Linear system:  $\underline{A}\underline{x} = \underline{b}$
- Two approaches
  - direct methods: Cholesky, LU with pivoting
    - factorize A into product of lower and upper triangular matrices  $A = LU$
    - solve two triangular systems  
 $L\underline{y} = \underline{b}$   
 $U\underline{x} = \underline{y}$
    - problems:
      - even if A is sparse, L and U can be quite dense ("fill")
      - no useful information is produced until the end of the procedure
  - iterative methods: Jacobi, Gauss-Seidel, CG, GMRES
    - guess an initial approximation  $\underline{x}_0$  to solution
    - error is  $A\underline{x}_0 - \underline{b}$  (called residual)
    - repeatedly compute better approximation  $\underline{x}_{i+1}$  from residual  $(A\underline{x}_i - \underline{b})$
    - terminate when approximation is "good enough"

21

## Iterative method: Jacobi iteration

- Linear system  
 $4x+2y=8$   
 $3x+4y=11$
- Exact solution is  $(x=1, y=2)$
- Jacobi iteration for finding approximations to solution
  - guess an initial approximation
  - iterate
    - use first component of residual to refine value of x
    - use second component of residual to refine value of y
- For our example
 
$$\begin{aligned}x_{i+1} &= (8 - 2y_i)/4 \\y_{i+1} &= (11 - 3x_i)/4\end{aligned}$$
  - for initial guess  $(x_0=0, y_0=0)$

i	0	1	2	3	4	5	6	7
x	0	2	0.625	1.375	0.8594	1.1406	0.9473	1.0527
y	0	2.75	1.250	2.281	1.7188	2.1055	1.8945	2.0396

22

## Jacobi iteration: matrix notation

- Linear system  
 $4x+2y=8$   
 $3x+4y=11$
- Jacobi iteration
 
$$\begin{aligned}x_{i+1} &= (8 - 2y_i)/4 \\y_{i+1} &= (11 - 3x_i)/4\end{aligned}$$
- Useful to write Jacobi iteration in terms of residual (error):
 
$$\begin{aligned}x_{i+1} &= x_i - \frac{1}{4}(4x_i + 2y_i - 8) \\y_{i+1} &= y_i - \frac{1}{4}(3x_i + 4y_i - 11)\end{aligned}$$
- In matrix terms, this is

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} x_i \\ y_i \end{pmatrix} - \begin{pmatrix} 1/4 & 0 \\ 0 & 1/4 \end{pmatrix} \begin{pmatrix} 4x_i + 2y_i - 8 \\ 3x_i + 4y_i - 11 \end{pmatrix}$$

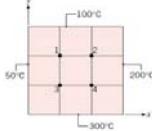
23

## Jacobi iteration: general picture

- Linear system  $\underline{A}\underline{x} = \underline{b}$
- Jacobi iteration
 
$$\underline{x}_{i+1} = \underline{x}_i - M^{-1}(\underline{A}\underline{x}_i - \underline{b})$$
 (where M is the diagonal of A)
- Key operation:
  - matrix-vector multiplication
  - important to exploit sparsity structure of A to reduce storage and computation
- Caveat:
  - Jacobi iteration does not always converge
  - even when it converges, it usually converges slowly
  - there are faster iterative methods available: CG, GMRES,...
  - what is important from our perspective is that key operation in all these iterative methods is **matrix-vector multiplication**

24

## Example (contd)



$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y)$$

Assume  $f(x,y) = 0$

$$\begin{aligned} -4T_1 + T_2 + T_3 &= -150 \\ T_1 - 4T_2 + T_4 &= -300 \\ T_1 - 4T_3 + T_4 &= -350 \\ T_2 + T_3 - 4T_4 &= -500 \end{aligned}$$

$$\begin{aligned} -4T_1^{n+1} + T_2^n + T_3^n + 0 &= -150 \\ T_1^n - 4T_2^{n+1} + 0 + T_4^n &= -300 \\ T_1^n + 0 - 4T_3^{n+1} + T_4^n &= -350 \\ 0 + T_2^n + T_3^n - 4T_4^{n+1} &= -500 \end{aligned}$$

$$\begin{aligned} T_1^{n+1} &= \frac{1}{4}(T_2^n + T_3^n + 100 + 50) \\ T_2^{n+1} &= \frac{1}{4}(T_1^n + T_4^n + 100 + 200) \\ T_3^{n+1} &= \frac{1}{4}(T_1^n + T_4^n + 300 + 50) \\ T_4^{n+1} &= \frac{1}{4}(T_2^n + T_3^n + 300 + 200) \end{aligned}$$

- Initialize the interior temperatures to some values
- Iterate until some measure of convergence is met
- Key operation is a sparse MVM
- However, code is implemented without explicit sparse matrices

25

## Implementation

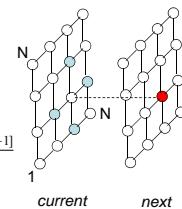
### Data structures

- temperature values at a given iteration can be stored in a NxN matrix
- use two matrices, *current* and *next*

### Algorithm

- compute values in *next* using values in *current*
- operator: five-point stencil

$$next[i, j] = \frac{current[i-1, j] + current[i, j-1] + current[i+1, j] + current[i, j+1]}{4}$$



### Questions:

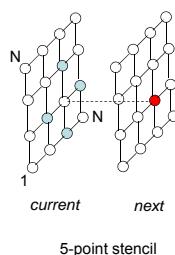
- where is the parallelism in this algorithm?
  - All grid points in *next* can be computed in parallel
- can we exploit locality?
  - Divide current and next into 2D blocks
  - Modern prefetchers will also do a fine job without blocking

5-point stencil

26

## Operator formulation of algorithms

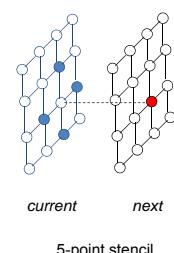
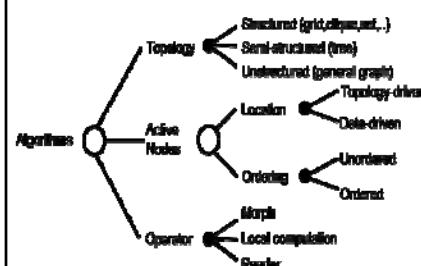
- Data-centric abstraction of algorithms
- Data structure: usually a graph
- Active element
  - Node /edge where computation is needed
  - Jacobi: all nodes of *next* grid
- Operator
  - Computation at active element
  - Jacobi: five-point stencil
- Activity: application of operator to active element
- Neighborhood
  - Set of nodes/edges read/written by activity
  - Jacobi: active node in *next* grid and neighbors in *current* grid
- Ordering : scheduling constraints on execution order of activities
  - Unordered algorithms: no semantic constraints but performance may depend on schedule
  - Ordered algorithms: problem-dependent order
  - Jacobi: unordered algorithm



5-point stencil

27

## TAO analysis: algorithm abstraction



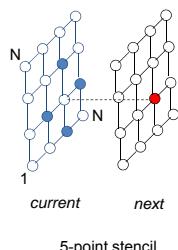
5-point stencil

- Jacobi
- Topology: structured (grid)
  - Active nodes: topology-driven (all nodes of *next* grid), unordered
  - Operator: local computation for *next*, reader for *current*

### Parallelism in unordered algorithms

- Work on multiple active nodes simultaneously
- Constraint:
  - final state must be identical to state produced by processing active nodes serially in some order
  - amorphous data-parallelism
- One implementation:
  - activities can be executed in parallel if and only if their neighborhoods are disjoint (otherwise, activities conflict)
  - correct but conservative: nearby active nodes in grid cannot be processed in parallel
- Another implementation:
  - if neighborhoods of concurrent activities overlap, graph elements in intersection of neighborhoods are read-only (more refined notion of conflict)
  - satisfactory for Jacobi
  - most general picture: commutativity of activities (we won't worry about this)
- Data parallelism:
  - topology-driven algorithm
  - no conflicts between activities

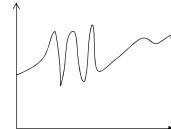
29



### Summary

- Finite-difference methods
  - can be used to find approximate solutions to ode's and pde's
  - Explicit methods: (e.g.) forward-Euler require matrix-vector multiplication
  - Implicit methods: (e.g.) backward-Euler or centered differences require solving linear system
- Many large-scale computational science simulations use these methods
- Time step or grid step needs to be constant and is determined by highest-frequency phenomenon
  - can be inefficient for when frequency varies widely in domain of interest
  - one solution: structured AMR methods

30



### Big picture

```

graph TD
    PM[Physical Models] --> CM[Continuous Models]
    CM --> FD[Finite-difference]
    CM --> FE[Finite-element]
    CM --> DM[Discrete Models]
    FD --> E[Explicit]
    FD --> I[Implicit]
    E --> MVM[MVM]
    I --> AxB[Ax=b]
    FE --> IM[Iterative methods<br/>(Jacobi,CG,...)]
    FE --> DM
    IM --> MVM
    AxB --> IM
    AxB --> DM
    DM --> D[Direct methods<br/>(Cholesky,LU)]
    D --> MVM
  
```

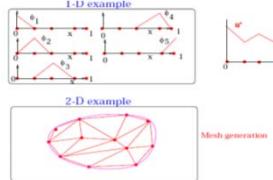
31

### Finite-element methods

- Express approximate solution to pde as a linear combination of certain basis functions
- Similar in spirit to Fourier analysis
  - express periodic functions as linear combinations of sines and cosines
- Questions:
  - what should be the basis functions?
    - mesh generation: discretization step for finite-elements
    - mesh defines basis functions  $\psi_3, \psi_4, \psi_5, \dots$  which are low-degree piecewise polynomial functions
    - given the basis functions, how do we find the best linear combination of these for approximating solution to pde?
    - $u = \sum_i c_i \psi_i$
    - weighted residual method: similar in spirit to what we do in Fourier analysis, but more complex because basis functions are not necessarily orthogonal

32

## Mesh generation and refinement



- **1-D example:**
  - mesh is a set of points, not necessarily equally spaced
  - basis functions are "hats" which
    - have a value of 1 at a mesh point,
    - decay down to 0 at neighboring mesh points
    - 0 everywhere else
  - linear combinations of these produce piecewise linear functions in domain, which may change slope only at mesh points
- In 2-D, mesh is a triangulation of domain, while in 3-D, it might be a tetrahedralization
- Mesh refinement: called h-refinement
  - add more points to mesh in regions where discretization error is large
  - irregular nature of mesh makes this easy to do this locally
  - finite-differences require global refinement which can be computationally expensive

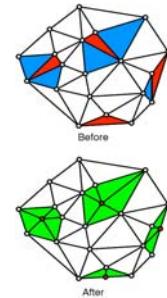
33

## Delaunay Mesh Refinement

- Iterative refinement to remove bad triangles with lots of discretization error:

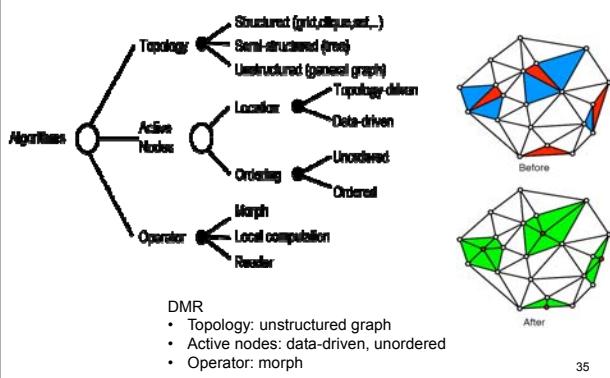
```
while there are bad triangles do {
    pick a bad triangle;
    find its cavity;
    retriangulate cavity;
    // may create new bad triangles
}
```

- Don't-care non-determinism:
  - final mesh depends on order in which bad triangles are processed
  - applications do not care which mesh is produced
- Data structure:
  - graph in which nodes represent triangles and edges represent triangle adjacencies
- Parallelism:
  - bad triangles with cavities that do not overlap can be processed in parallel
  - parallelism is dependent on runtime values
    - compilers cannot find this parallelism



34

## TAO analysis



35

## Finding coefficients

### Weighted residual technique

- similar in spirit to what we do in Fourier analysis, but basis functions are not necessarily orthogonal

### Key idea:

- problem is reduced to solving a system of equations  $Ax = b$
- solution gives the coefficients in the weighted sum
- because basis functions are zero almost everywhere in the domain, matrix A is usually very sparse
  - number of rows/columns of A  $\sim O(\text{number of points in mesh})$
  - number of non-zeros per row  $\sim O(\text{connectivity of mesh point})$
- typical numbers:
  - A is  $10^9 \times 10^9$
  - only about  $\sim 100$  non-zeros per row

36

**Finding the best choices of the coefficients:**

**Analogy with Fourier series:**

$$f(x) = a_0 + \sum_i a_i \cos(ix) + \sum_i b_i \sin(ix)$$

How do you find 'best' choices for a's and b's?

$$\int_{-\pi}^{+\pi} f(x) \cos(kx) dx = \int_{-\pi}^{+\pi} (a_0 + \sum_i a_i \cos(ix) + \sum_i b_i \sin(ix)) \cos(kx) dx$$

$$= \int_{-\pi}^{+\pi} a_k \cos(kx) \cos(kx) dx$$

$$= a_k \pi$$

**Key idea:** - residual  $f(x) - a_0 - \sum_i a_i \cos(ix) - \sum_i b_i \sin(ix)$   
- weight residual by known function and integrate to find corresponding coefficient



37

### Weighted residuals: intuitive idea

**ODE:**  $\frac{du}{dx} = -3u + 2$

**Approximate solution:**  $u^*(t) = \sum_{i=1}^N c_i \phi_i(t)$

**Residual(t) =**  $\frac{d(u^*(t))}{dt} + 3u^*(t) - 2$

**Write this as**  

$$\text{Residual}(t) = L(u^*(t)) - f(t)$$

- Residual depends on choice of  $c_i$
- Choose  $c_i$  so that integral of residual, weighted by each  $\phi_k$  is zero.
- This gives N equations in N unknowns, which can be solved to find values for  $c_i$

38

**Weighted Residual Technique:**

Residual:  $(L u^* - f) = (L(\sum_{i=1}^N c_i \phi_i) - f)$

Weighted Residual:  $= (L(\sum_{i=1}^N c_i \phi_i) - f) \cdot \phi_k$

Equation for  $k^{\text{th}}$  unknown:  $\int_{\Omega} \phi_k (L(\sum_{i=1}^N c_i \phi_i) - f) dV = 0 \Rightarrow$

If the differential equation is linear:

$$c_1 \int_{\Omega} \phi_k \cdot L \phi_1 dV + \dots + c_N \int_{\Omega} \phi_k \cdot L \phi_N dV = \int_{\Omega} \phi_k f dV \quad k = 1, 2, \dots, N$$

This system can be written as

$$Kc = b \text{ where } K(i,j) = \int_{\Omega} \phi_i \cdot L \phi_j dV \quad b(i) = \int_{\Omega} \phi_i f dV$$

**Key insight:** Calculus problem of solving pde is converted to linear algebra problem of solving  $K c = b$  where  $K$  is sparse

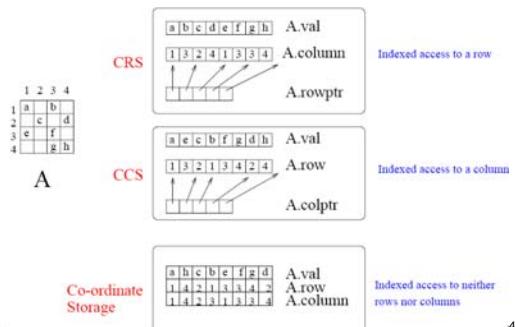
39

### Sparse matrices in finite-element method

- Sparsity pattern is complex and irregular
  - Pattern and values of non-zeros depends on the mesh and basis functions, and is not known at compile-time
  - Cannot be inlined into code like we did for heat equation
- Solution:
  - represent sparse matrix explicitly
  - Use sparse MVM code specialized to that representation

40

## Sparse matrix representations



41

## MVM with sparse matrices

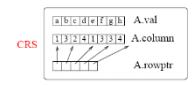
- Coordinate storage

```
for P = 1 to NZ do
    Y(A.row(P))=Y(A.row(P)) + A.val(P)*X(A.column(P))
```



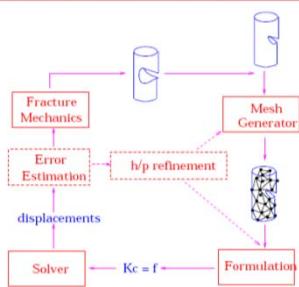
- CRS storage

```
for I = 1 to N do
    for JJ = A.rowptr(I) to A.rowptr(I+1)-1 do
        Y(I)=Y(I)+A.val(JJ)*X(A.column(JJ))
```



42

Flow-chart of Adaptive Finite-element Simulation of Fracture



43

## Barnes Hut N-body Simulation

44

## Introduction

- Physical system simulation (time evolution)
  - System consists of **bodies**
  - “**n**” is the number of bodies
  - Bodies interact via **pair-wise forces**
- Many systems can be modeled in these terms
  - Galaxy clusters (gravitational force)
  - Particles (electric force, magnetic force)

45

## Barnes Hut Idea

- Precise force calculation
  - Requires  $O(n^2)$  operations ( $O(n^2)$  body pairs)
- Barnes and Hut (1986)
  - Algorithm to approximately compute forces
    - Bodies' initial position & velocity are also approximate
  - Requires only  $O(n \log n)$  operations
  - Idea is to “combine” far away bodies
  - Error should be small because  $force \sim 1/r^2$

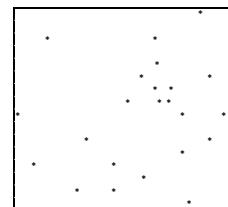
46

## Barnes Hut Algorithm

- Set bodies' initial position and velocity
- Iterate over time steps
  - Subdivide space until at most one body per cell
    - Record this spatial hierarchy in an octree
  - Compute mass and center of mass of each cell
  - Compute force on bodies by traversing octree
    - Stop traversal path when encountering a leaf (body) or an internal node (cell) that is far enough away
  - Update each body's position and velocity

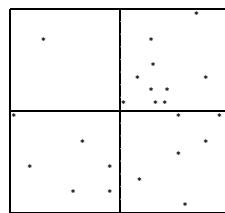
47

## Build Tree (Level 1)



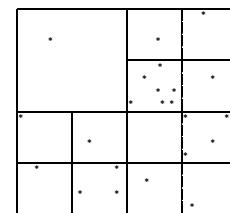
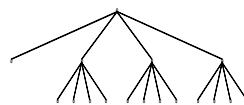
Subdivide space until at most one body per cell

48

Build Tree (Level 2)

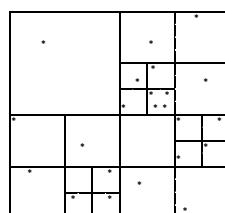
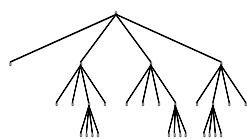
Subdivide space until at most one body per cell

49

Build Tree (Level 3)

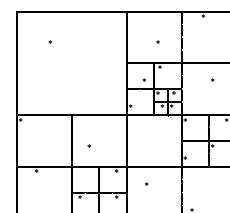
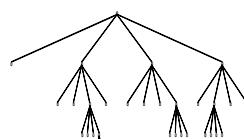
Subdivide space until at most one body per cell

50

Build Tree (Level 4)

Subdivide space until at most one body per cell

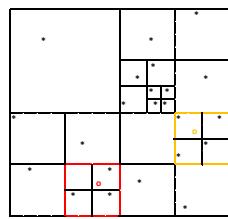
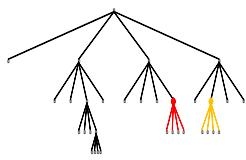
51

Build Tree (Level 5)

Subdivide space until at most one body per cell

52

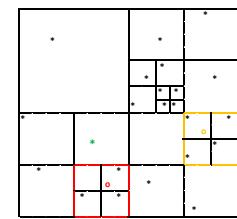
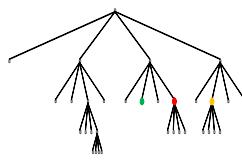
### Compute Cells' Center of Mass



For each internal cell, compute sum of mass and weighted average of position of all bodies in subtree; example shows two cells only

53

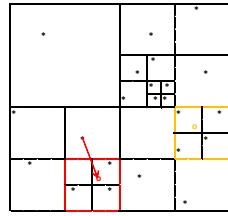
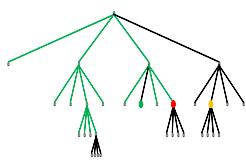
### Compute Forces



Compute force, for example, acting upon green body

54

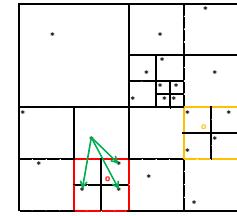
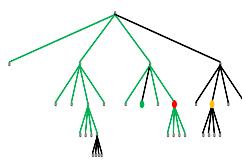
### Compute Force (short distance)



Scan tree depth first from left to right; green portion already completed

55

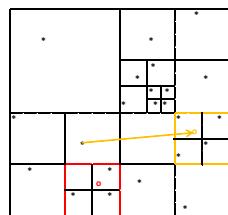
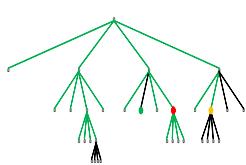
### Compute Force (down one level)



Red center of mass is too close, need to go down one level

56

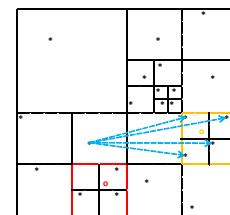
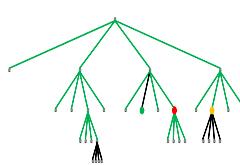
### Compute Force (long distance)



Yellow center of mass is far enough away

57

### Compute Force (skip subtree)



Therefore, entire subtree rooted in the yellow cell can be skipped

58

### Pseudocode

```
Set bodySet = ...
foreach timestep do {
    Octree octree = new Octree();
    foreach Body b in bodySet {
        octree.Insert(b);
    }
    OrderedList cellList = octree.CellsByLevel();
    foreach Cell c in cellList {
        c.Summarize();
    }
    foreach Body b in bodySet {
        b.ComputeForce(octree);
    }
    foreach Body b in bodySet {
        b.Advance();
    }
}
```

59

### Complexity

```
Set bodySet = ...
foreach timestep do {           // O(n log n)
    Octree octree = new Octree();
    foreach Body b in bodySet {   // O(n log n)
        octree.Insert(b);
    }
    OrderedList cellList = octree.CellsByLevel();
    foreach Cell c in cellList { // O(n)
        c.Summarize();
    }
    foreach Body b in bodySet { // O(n log n)
        b.ComputeForce(octree);
    }
    foreach Body b in bodySet { // O(n)
        b.Advance();
    }
}
```

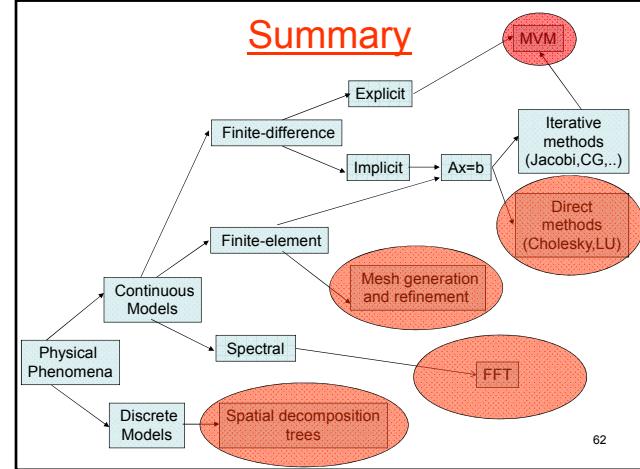
60

## Parallelism

```
Set bodySet = ...
foreach timestep do {           // sequential
    Octree octree = new Octree();
    foreach Body b in bodySet {   // tree building
        octree.Insert(b);
    }
    OrderedList cellList = octree.CellsByLevel();
    foreach Cell c in cellList { // tree traversal
        c.Summarize();
    }
    foreach Body b in bodySet { // fully parallel
        b.ComputeForce(octree);
    }
    foreach Body b in bodySet { // fully parallel
        b.Advance();
    }
}
```

61

## Summary



62

## Summary (contd.)

- Some key computational science algorithms and data structures
  - MVM:
    - Source: explicit finite-difference methods for ode's, iterative linear solvers, finite-element methods
    - Both dense and sparse matrices
  - Stencil computations:
    - Source: explicit finite-difference methods for pde's
    - Dense matrices
  - $A=LU$ :
    - Source: implicit finite-difference methods
    - Direct methods for solving linear systems: factorization
    - Usually only dense matrices
    - High-performance factorization codes use MMM as a kernel
  - Mesh generation and refinement
    - Finite-element methods
    - Graph computations

63

## Extra material

64

## Systems of ode's

- Consider a system of coupled ode's of the form
$$\begin{aligned} u'(t) &= a_{11}u(t) + a_{12}v(t) + a_{13}w(t) + c_1(t) \\ v'(t) &= a_{21}u(t) + a_{22}v(t) + a_{23}w(t) + c_2(t) \\ w'(t) &= a_{31}u(t) + a_{32}v(t) + a_{33}w(t) + c_3(t) \end{aligned}$$
- If we use Forward-Euler method to discretize this system, we get the following system of simultaneous equations

$$\begin{aligned} u(t+h) - u(t) / h &= a_{11}u(t) + a_{12}v(t) + a_{13}w(t) + c_1(t) \\ v(t+h) - v(t) / h &= a_{21}u(t) + a_{22}v(t) + a_{23}w(t) + c_2(t) \\ w(t+h) - w(t) / h &= a_{31}u(t) + a_{32}v(t) + a_{33}w(t) + c_3(t) \end{aligned}$$

65

## Forward-Euler (contd.)

- Rearranging, we get

$$\begin{aligned} u(t+h) &= (1+ha_{11})u(t) + ha_{12}v(t) + ha_{13}w(t) + hc_1(t) \\ v(t+h) &= ha_{21}u(t) + (1+ha_{22})v(t) + ha_{23}w(t) + hc_2(t) \\ w(t+h) &= ha_{31}u(t) + ha_{32}v(t) + (1+a_{33})w(t) + hc_3(t) \end{aligned}$$

- Introduce vector/matrix notation

$$\underline{x}(t) = [u(t) \ v(t) \ w(t)]^T$$

$$A = \dots$$

$$\underline{c}(t) = [c_1(t) \ c_2(t) \ c_3(t)]^T$$

66

## Vector notation

- Our systems of equations was
$$\begin{aligned} u(t+h) &= (1+ha_{11})u(t) + ha_{12}v(t) + ha_{13}w(t) + hc_1(t) \\ v(t+h) &= ha_{21}u(t) + (1+ha_{22})v(t) + ha_{23}w(t) + hc_2(t) \\ w(t+h) &= ha_{31}u(t) + ha_{32}v(t) + (1+a_{33})w(t) + hc_3(t) \end{aligned}$$
- This system can be written compactly as follows
$$\underline{x}(t+h) = (I+hA)\underline{x}(t) + h\underline{c}(t)$$
- We can use this form to compute values of  $\underline{x}(h), \underline{x}(2h), \underline{x}(3h), \dots$
- Forward-Euler is an example of **explicit method** of discretization
  - key operation: matrix-vector (MVM) multiplication
  - in principle, there is a lot of parallelism
    - $O(n^2)$  multiplications
    - $O(n)$  reductions
  - parallelism is independent of runtime values

67

## Backward-Euler

- We can also use Backward-Euler method to discretize system of ode's
$$\begin{aligned} u_b(t) - u_b(t-h) / h &= a_{11}u_b(t) + a_{12}v_b(t) + a_{13}w_b(t) + c_1(t) \\ v_b(t) - v_b(t-h) / h &= a_{21}u_b(t) + a_{22}v_b(t) + a_{23}w_b(t) + c_2(t) \\ w_b(t) - w_b(t-h) / h &= a_{31}u_b(t) + a_{32}v_b(t) + a_{33}w_b(t) + c_3(t) \end{aligned}$$
- We can write this in matrix notation as follows
$$(I-hA)\underline{x}(t) = \underline{x}(t-h) + h\underline{c}(t)$$
- Backward-Euler is example of **implicit method** of discretization
  - key operation: solving a linear system  $A\underline{x} = b$
  - How do we solve large systems of linear equations?
  - Matrix  $(I-hA)$  is often very sparse
    - Important to exploit sparsity in solving linear systems

68

# Graph Algorithms

# Overview

- Graphs are very general data structures
  - $G = (V, E)$  where  $V$  is set of nodes,  $E$  is set of edges  $\subseteq V \times V$
  - data structures such as dense and sparse matrices, sets, multisets, etc. can be viewed as representations of graphs
- Algorithms on matrices/sets/etc. can usually be interpreted as graph algorithms
  - but it may or may not be useful to do this
  - sparse matrix algorithms can be usefully viewed as graph algorithms
- Some graph algorithms can be interpreted as matrix algorithms
  - but it may or may not be useful to do this
  - may be useful if graph structure is fixed as in graph analytics applications:
    - topology-driven algorithms can often be formulated in terms of a generalized sparse MVM

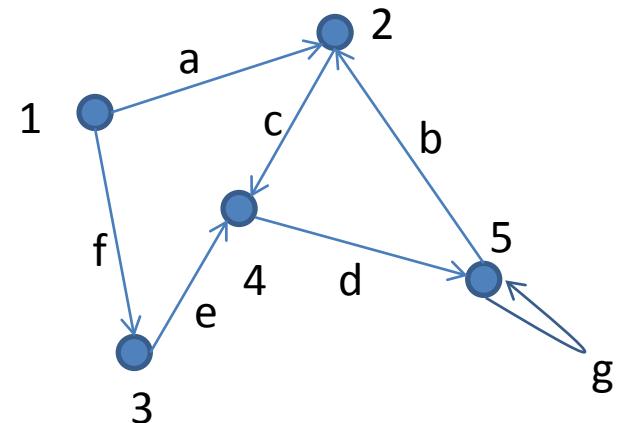
# Graph-matrix duality

- Graph  $(V, E)$  as a matrix

- Choose an ordering of vertices
- Number them sequentially
- Fill in  $|V| \times |V|$  matrix
  - $A(i,j)$  is  $w$  if graph has edge from node  $i$  to node  $j$  with label  $w$
- Called *adjacency matrix* of graph
- Edge  $(u \rightarrow v)$ :
  - $v$  is *out-neighbor* of  $u$
  - $u$  is *in-neighbor* of  $v$

- Observations:

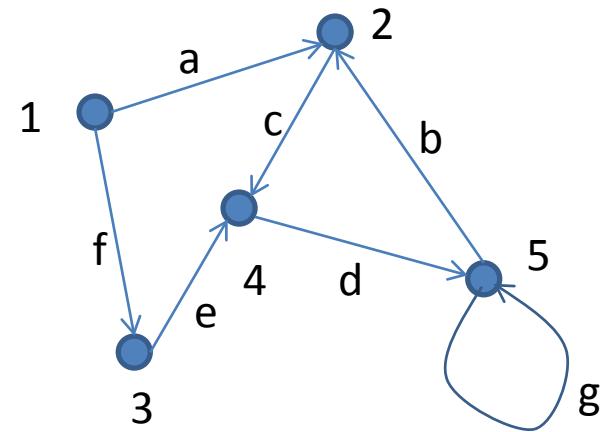
- Diagonal entries: weights on self-loops
- Symmetric matrix  $\leftrightarrow$  undirected graph
- Lower triangular matrix  $\leftrightarrow$  no edges from lower numbered nodes to higher numbered nodes
- Dense matrix  $\leftrightarrow$  clique (edge between every pair of nodes)



from \ to	1	2	3	4	5
1	0	a	f	0	0
2	0	0	0	c	0
3	0	0	0	e	0
4	0	0	0	0	d
5	0	b	0	0	g

# Matrix-vector multiplication

- Matrix computation:  $\underline{y} = A\underline{x}$
- Graph interpretation:
  - Each node  $i$  has two values (labels)  $x(i)$  and  $y(i)$
  - Each node  $i$  updates its label  $y$  using the  $x$  value from each out-neighbor  $j$ , scaled by the label on edge  $(i,j)$
  - Topology-driven, unordered algorithm
- Observation:
  - Graph perspective shows dense MVM is special case of sparse MVM
  - What is the interpretation of  $\underline{y} = A^T \underline{x}$  ?



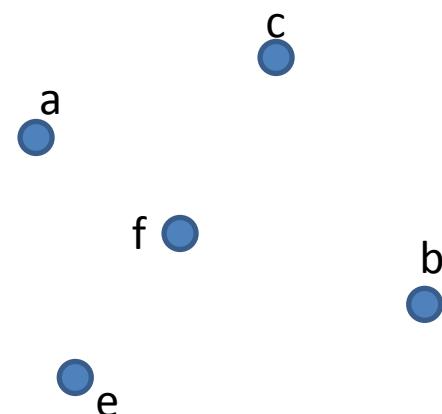
$$\begin{matrix} & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & a & f & 0 & 0 \\ 2 & 0 & 0 & 0 & c & 0 \\ 3 & 0 & 0 & 0 & e & 0 \\ 4 & 0 & 0 & 0 & 0 & d \\ 5 & 0 & b & 0 & 0 & g \end{matrix} \begin{matrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{matrix} \quad \begin{matrix} A \\ X \end{matrix}$$

# Graph set/multiset duality

- Set/multiset is isomorphic to a graph
  - labeled nodes
  - no edges
- “Opposite” of clique
- Algorithms on sets/multisets can be viewed as graph algorithms
- Usually no particular advantage to doing this but it shows generality of graph algorithms

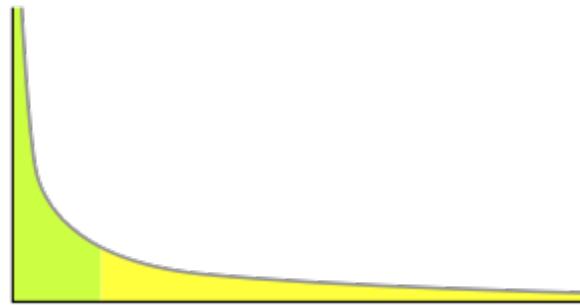
$\{a,c,f,e,b\}$

Set



Graph

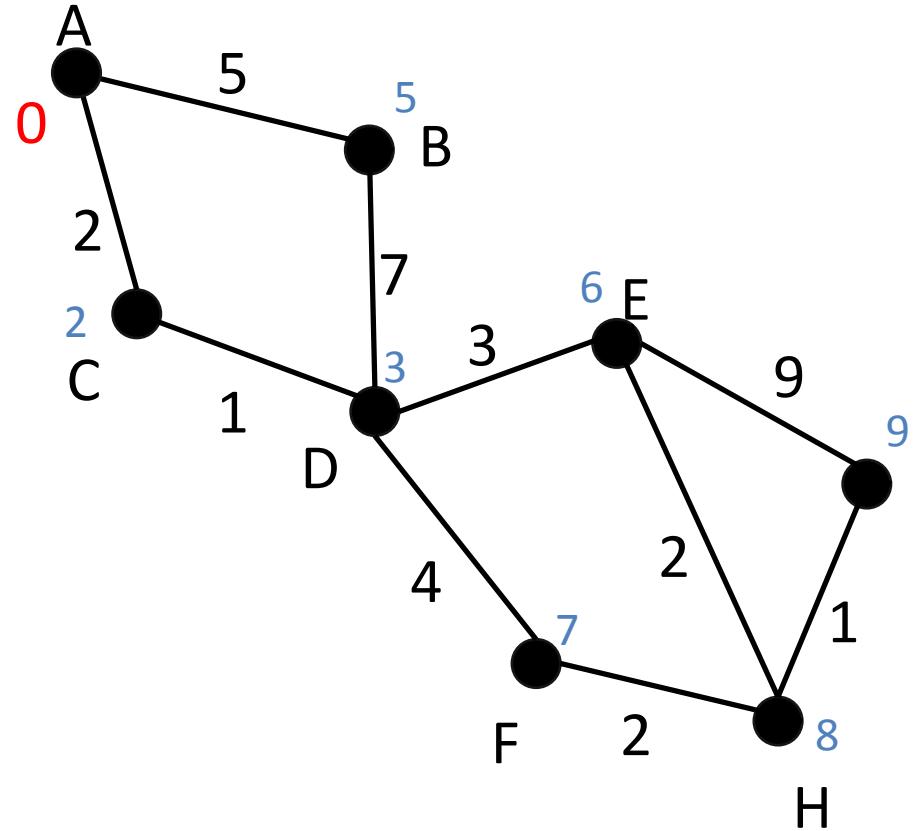
# Sparse graph types



- Power-law graphs
  - small number of very high degree nodes (hubs)
  - low diameter:
    - get from a node to any other node in  $O(1)$  hops
    - “six degrees of separation” (Karinthy 1929, Milgram 1967), on Facebook, it is 4.74
  - typical of social network graphs like the Internet graph or the Facebook graph
- Uniform-degree graphs
  - nodes have roughly same degree
  - high diameter
  - road networks, IC circuits, finite-element meshes
- Random (Erdös-Rènyi) graphs
  - constructed by random insertion of edges
  - mathematically interesting but few real-life examples

# Graph problem:SSSP

- Problem: single-source shortest-path (SSSP) computation
- Formulation:
  - Given an undirected graph with positive weights on edges, and a node called the source
  - Compute the shortest distance from source to every other node
- Variations:
  - Negative edge weights but no negative weight cycles
  - All-pairs shortest paths
  - Breadth-first search: all edge weights are 1
- Applications:
  - GPS devices for driving directions
  - social network analyses: centrality metrics



Node A is the source

# SSSP Problem

- Many algorithms
  - Dijkstra (1959)
  - Bellman-Ford (1957)
  - Chaotic relaxation (1969)
  - Delta-stepping (1998)

- Common structure:

- Each node has a label  $d$  that is updated repeatedly
  - initialized to 0 for source and  $\infty$  for all other nodes
  - during algorithm: shortest known distance to that node from source
  - termination: shortest distance from source

- All of them use the same *operator*:

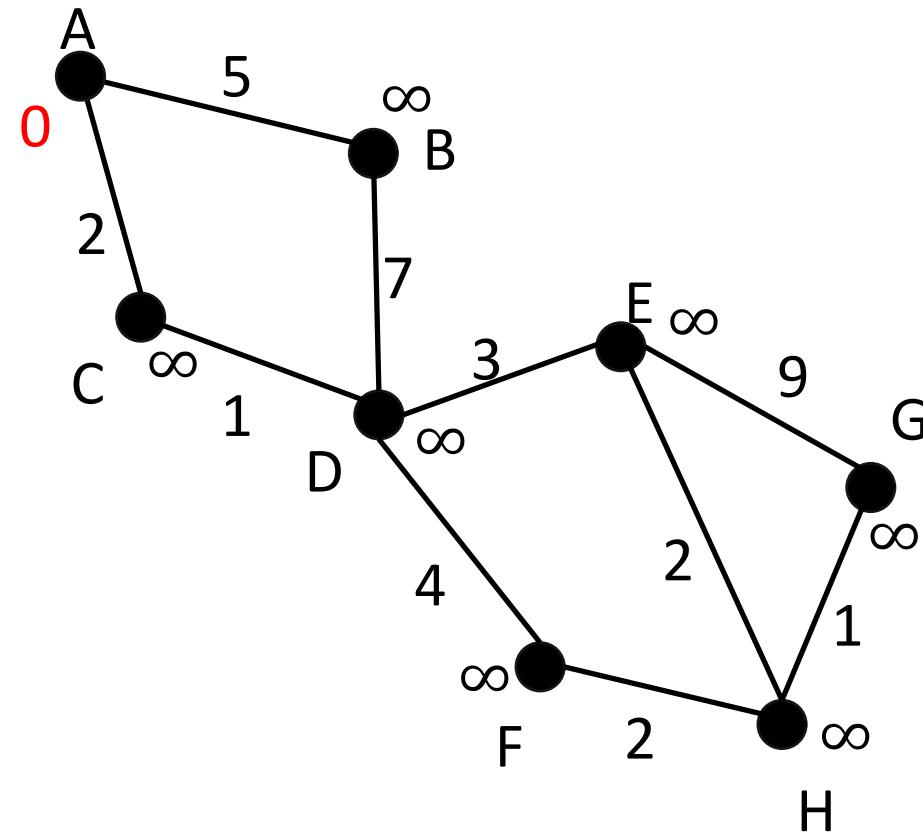
`relax-edge(u,v):`

  if  $d[v] > d[u] + w(u,v)$

  then  $d[v] \leftarrow d[u] + w(u,v)$

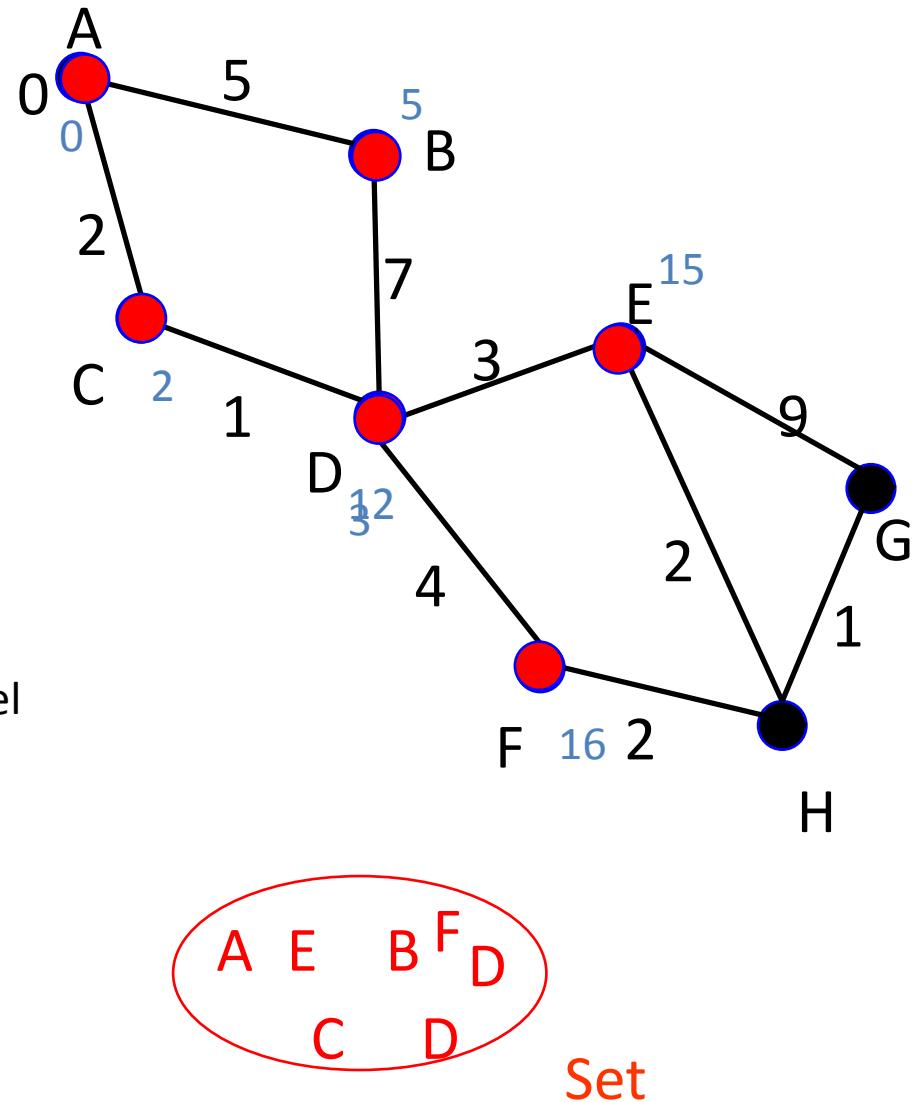
`relax-node(u):`

  relax all edges connected to u



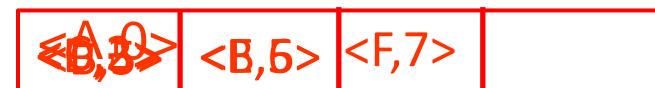
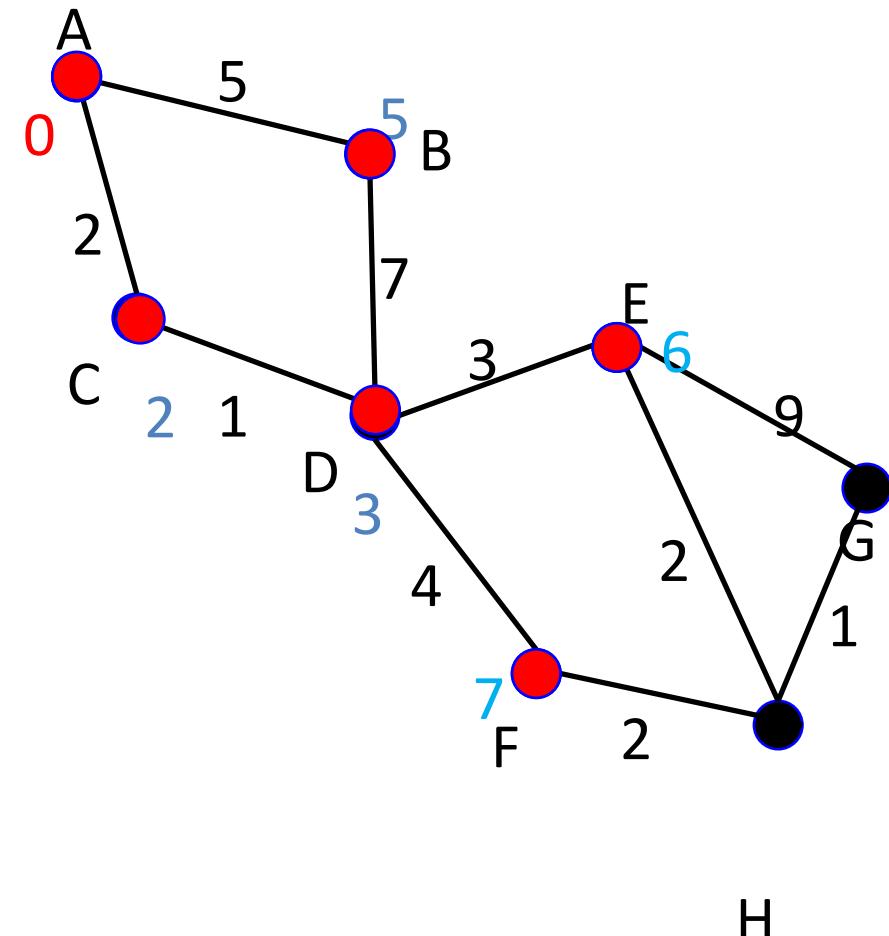
# Chaotic relaxation (1969)

- Active node:
  - node whose label has been updated
  - initially, only source is active
- Schedule for processing nodes
  - pick active nodes at random
- Implementation
  - use a (work) **set** or **multiset** to track active nodes
- TAO classification:
  - unstructured graph, data-driven, unordered, local computation
  - compare/contrast with DMR
- Parallelization:
  - process multiple work-set nodes in parallel
  - conflict: two activities may try to update label of the same node
    - eg., B and C may try to update D
  - amorphous data-parallelism
- Main inefficiency: number of node relaxations depends on the schedule
  - can be exponential in the size of graph



# Dijkstra's algorithm (1959)

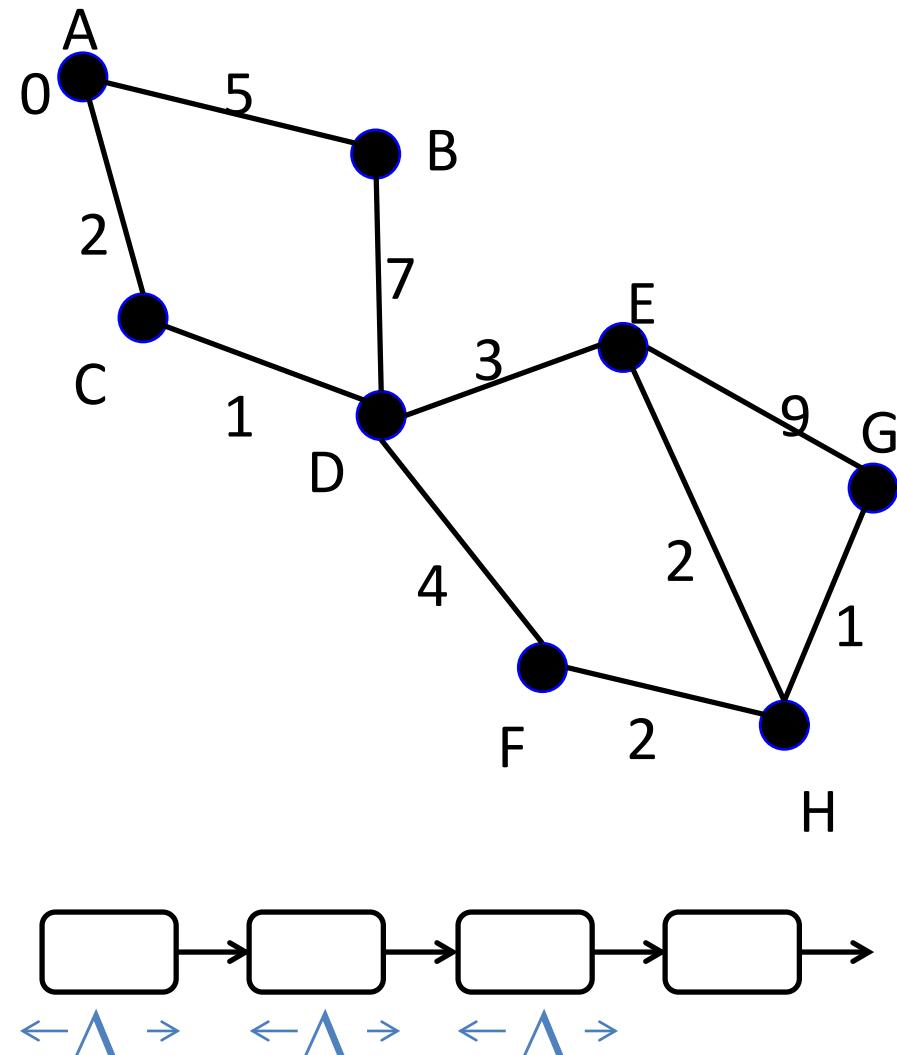
- Active nodes:
  - node whose label has been updated
  - initially, only source is active
- Schedule for processing nodes:
  - prefer nodes with smaller labels since they are more likely to have reached final values
- Implementation of work-set:
  - priority queue of nodes, ordered by label
- Work-efficient ordered algorithm
  - node is relaxed just once
  - $O(|E| * \lg(|V|))$
- TAO classification:
  - unstructured graph, data-driven, ordered, local computation
  - compare with tree summation
- Parallelism
  - nodes with minimal labels can be done in parallel if they don't conflict
  - “level-by-level” parallelization
  - limited parallelism for most graphs
- Main inefficiency:
  - little parallelism in sparse graphs



Priority queue

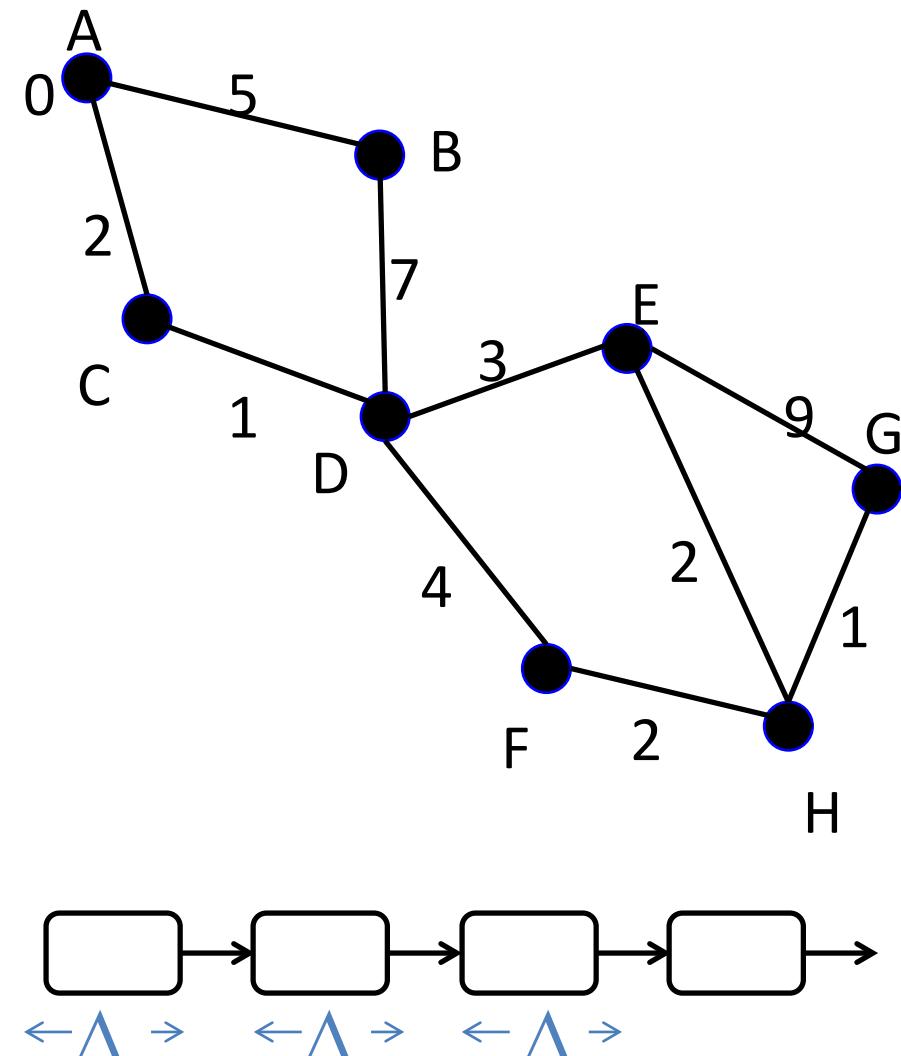
# Delta-stepping (1998)

- Controlled chaotic relaxation
  - Exploit the fact that SSSP is robust to priority inversions
  - “soft” priorities
- Implementation of work-set:
  - parameter:  $\Delta$
  - sequence of sets
  - nodes whose current distance is between  $n\Delta$  and  $(n+1)\Delta$  are put in the  $n^{\text{th}}$  set
  - nodes in each set are processed in parallel
  - nodes in set  $n$  are completed before processing of nodes in set  $(n+1)$  are started
    - implementation requires **barrier synchronization**: no worker can proceed past barrier until all workers are at the barrier
- $\Delta = 1$ : Dijkstra
- $\Delta = \infty$ : Chaotic relaxation
- Picking an optimal  $\Delta$  :
  - depends on graph and machine
  - high-diameter graph  $\rightarrow$  large  $\Delta$
  - find experimentally



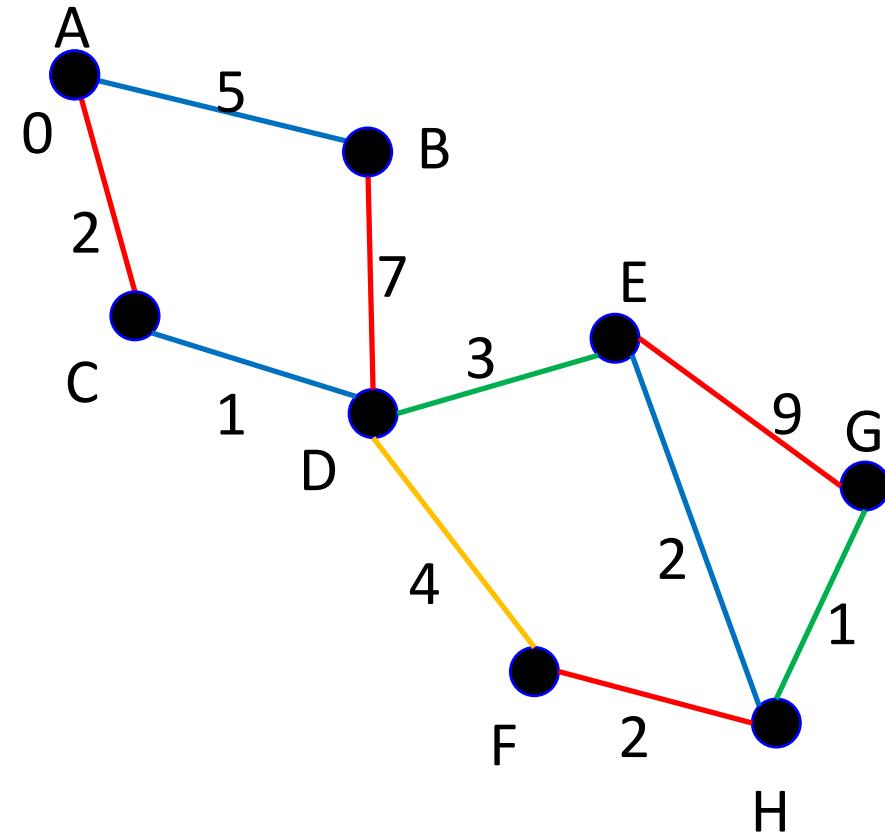
# Delta-stepping (II)

- Standard implementation of work-set:
  - nodes in set  $n$  are completed before processing of nodes in set  $(n+1)$  are started
  - barrier synchronization between processing of successive sets
- Strict barrier synchronization is not actually needed
  - once set  $n$  is empty, some threads can begin executing active nodes from set  $(n+1)$  without waiting for all threads to finish executing nodes from set  $n$



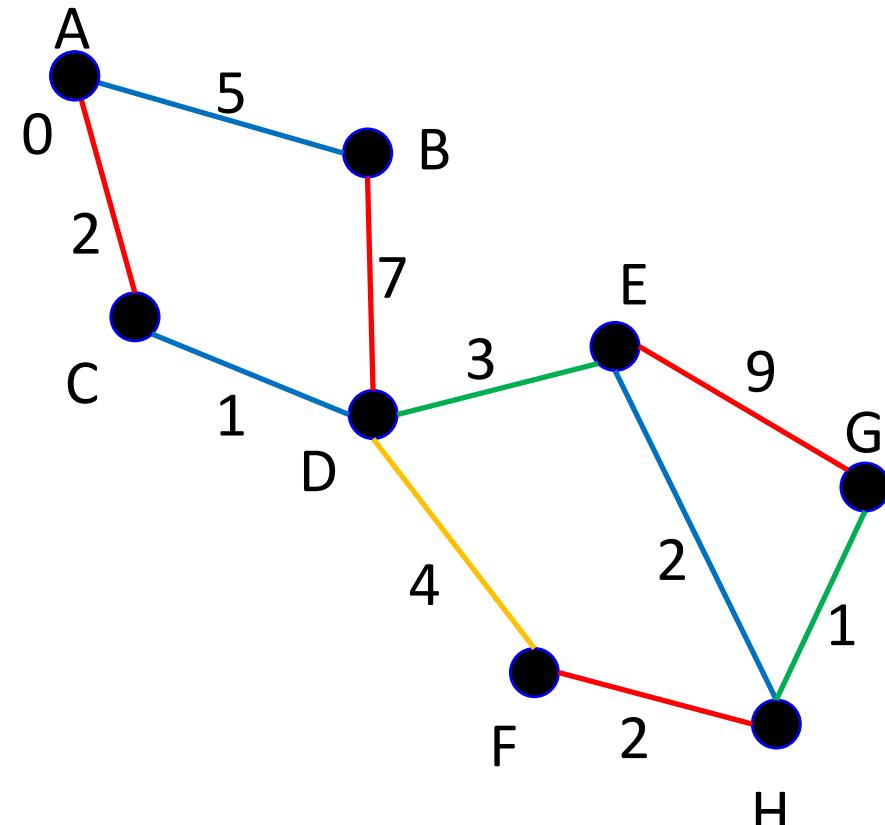
# Bellman-Ford (1957)

- Bellman-Ford (1957):
  - Iterate over all edges of graph in any order, relaxing each edge
  - Do this  $|V|$  times
  - $O(|E| * |V|)$
- TAO classification:
  - unstructured graph, topology-driven, unordered, local computation
- Parallelism
  - one approach: optimistic parallelization
  - repeat until no node label changes
    - put all edges into workset
    - workers get edges and apply relaxation operator if they can mark both nodes of edge, until workset is empty
  - can we do better?
    - since we may have to make  $O(|V|)$  sweeps over graph, it may be better to preprocess edges to avoid conflicts
    - overhead of preprocessing can be amortized over the multiple sweeps over the graph



# Matching

- Given a graph  $G = (V, E)$ , a matching is a subset of edges such that no edges in the subset have a node in common
  - (eg)  $\{(A,B), (C,D), (E,H)\}$
  - Not a matching:  $\{(A,B), (A,C)\}$
- Maximal matching: a matching to which no new edge can be added without destroying matching property
  - (eg)  $\{(A,B), (C,D), (E,H)\}$
  - (eg)  $\{(A,C), (B,D), (E,G), (F,H)\}$
  - Can be computed in  $O(|E|)$  time using a simple greedy algorithm
- Preprocessing strategy:
  - partition edges into matchings
  - many possible partitions, some better than others



Edges partitioned into matchings

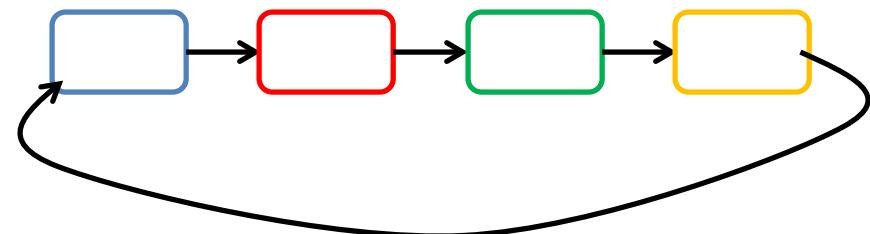
- $\{(A,B), (C,D), (E,H)\}$ ,
- $\{(A,C), (B,D), (E,G), (F,H)\}$ ,
- $\{(D,E), (G,H)\}$
- $\{(D,F)\}$

# Execution strategy

- Round-based execution
  - in each round, edges in one matching are processed in parallel w/o neighborhood marking (data parallelism)
  - barrier synchronization between rounds
- Disadvantage of round-based execution
  - all workers must wait at the barrier even if there is just one straggler
  - if we have 2 workers, round-based execution takes 6 steps
- Question: at a high level, there is some similarity to  $\Delta$ -stepping:
  - sequence of buckets
  - finish one bucket before moving on to next
  - what are the key differences in the implementations?

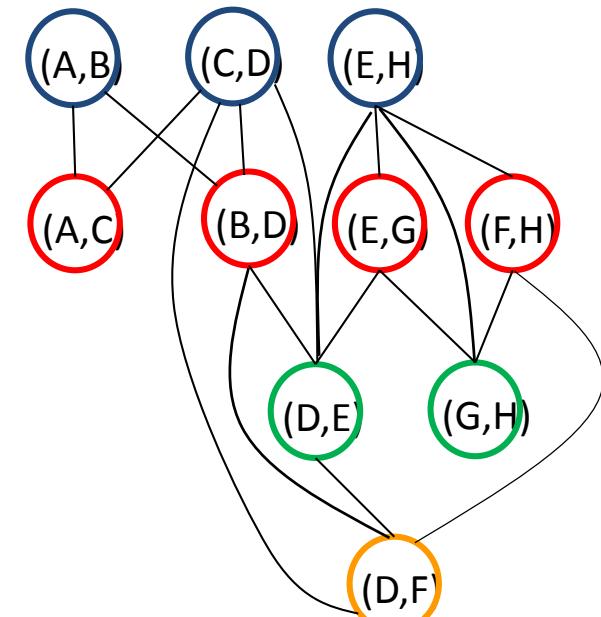
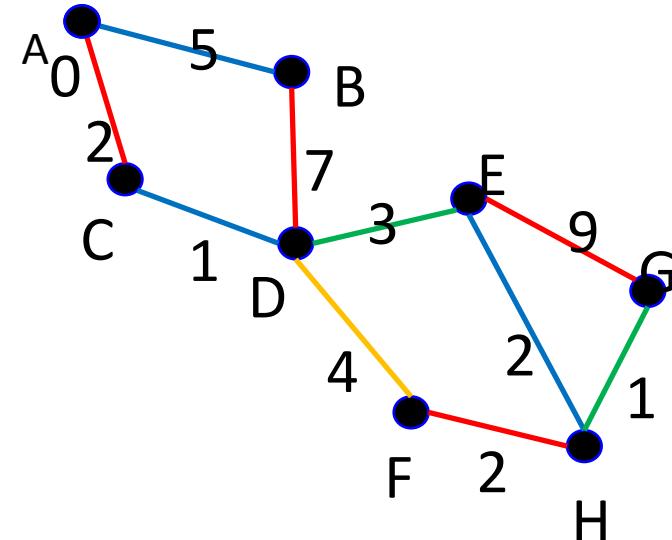
Round-based execution

1.  $\{(A,B),(C,D),(E,H)\}$
2.  $\{(A,C),(B,D),(E,G),(F,H)\}$
3.  $\{(D,E),(G,H)\}$
4.  $\{(D,F)\}$



# Another approach: interference graph

- Build interference graph (IG)
  - nodes are activities (SSSP graph edges)
  - edges represent conflicts between activities
    - for our problem, SSSP edges have a node in common
- For our problem
  - each SSSP graph edge represents a task
  - edge between task i and task j in IG if edges corresponding to tasks i and j in SSSP graph have a node in common



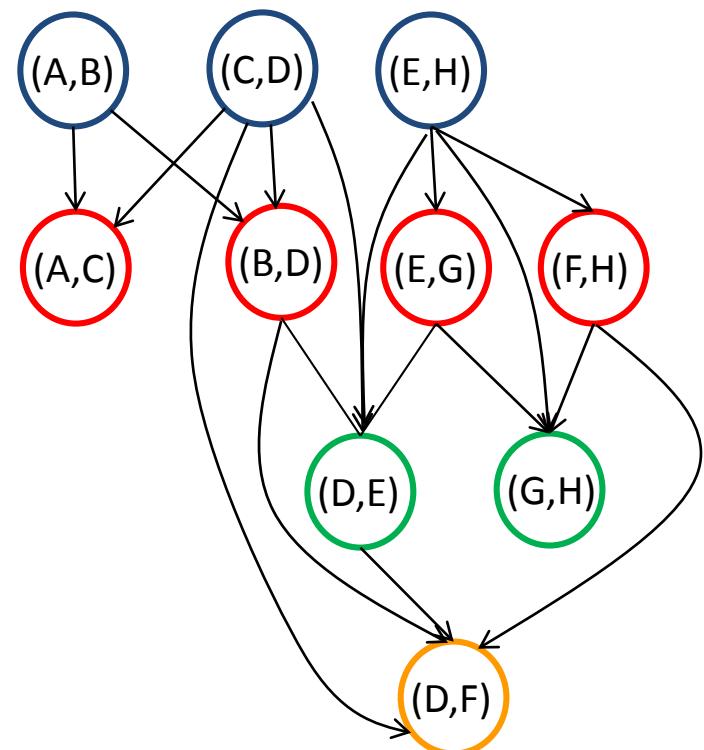
Interference graph

# Interference graph → Dependence graph

- Generate dependence graph

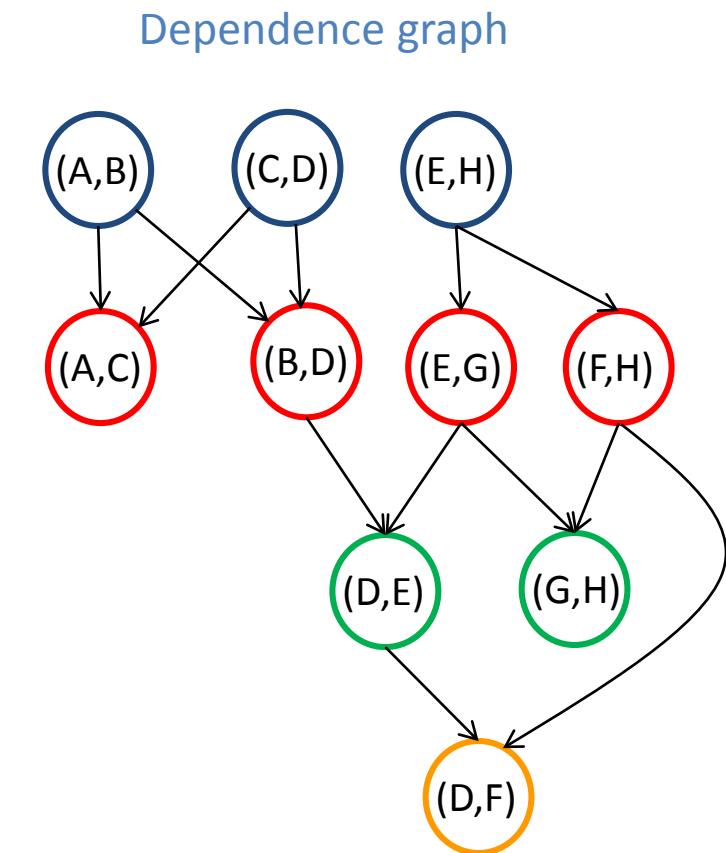
- change edges in IG to directed edges (precedence edges)
- make sure there are no cycles
- simple approach:
  - number all nodes in interference graph and direct edges from lower numbered nodes to higher numbered nodes
  - many other choices
- simplification: remove transitive edges

Interference graph



# Dependence graph

- Execution using dependence graph
  - each node has counter with number of incoming edges
  - any node with no incoming edges can be executed by a worker
  - when task is done, counters at out-neighbors are decremented, potentially making some of them sources
    - requires marks to ensure correct execution
  - execution terminates when all tasks have been completed
- Fewer ordering constraints between tasks than execution strategy based on matchings and rounds



# Inspector-executor

- When is inspector-executor parallelization possible?
  - when active nodes and neighborhoods are known as soon as input is given but before actual computation is executed
    - contrast:
      - static parallelization: active nodes and neighborhoods known at compile-time modulo problem size (example: Jacobi)
      - optimistic parallelization: active nodes and neighborhoods known only after program has been executed in parallel (example: DMR)
  - binding time analysis: when do we know some information regarding program behavior? Example: types
- When is inspector-executor parallelization useful?
  - when overhead of inspector can be amortized over many executions
    - works for Bellman-Ford because we make  $O(|V|)$  sweeps over graph
  - when overhead of inspector is small compared to executor
    - sparse Cholesky factorization: inspector is called symbolic factorization, executor is called numerical factorization

# Summary of SSSP Algorithms

- **Chaotic relaxation**
  - parallelism but amount of work depends on execution order of active nodes
  - unordered, data-driven algorithm: use sets/multisets
- **Dijkstra's algorithm**
  - work-efficient but difficult to extract parallelism
    - level-by-level parallelism
  - ordered, data-driven algorithm: use priority queues
- **Delta-stepping**
  - controlled chaotic relaxation: parameter  $\Delta$
  - $\Delta$  permits trade-off between parallelism and extra work
- **Bellman-Ford algorithm**
  - Inspector-executor parallelization:
    - inspector: use matchings or dependence graph to find parallelism after input is given

# Machine learning

- Many machine learning algorithms are sparse graph algorithms
- Examples:
  - Page rank: used to rank webpages to answer Internet search queries
  - Recommender systems: used to make recommendations to users in Netflix, Amazon, Facebook etc.

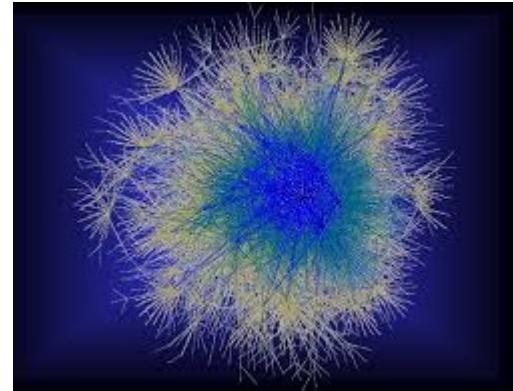
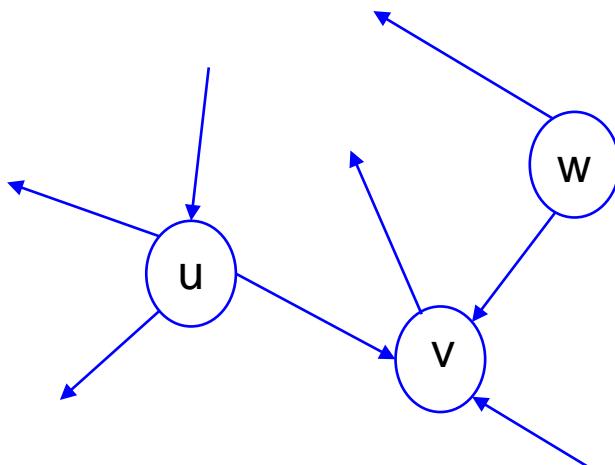
# Web search

- When you type a set of keywords to do an Internet search, which web-pages should be returned and in what order?
- Basic idea:
  - offline:
    - crawl the web and gather webpages into data center
    - build an index from keywords to webpages
  - online:
    - when user types keywords, use index to find all pages containing the keywords
  - key problem:
    - usually you end up with tens of thousands of pages
    - how do you rank these pages for the user?

# Ranking pages

- **Manual ranking**
  - Yahoo did something like this initially, but this solution does not scale
- **Word counts**
  - order webpages by how many times keywords occur in webpages
  - problem: easy to mess with ranking by having lots of meaningless occurrences of keyword
- **Citations**
  - analogy with citations to articles
  - if lots of webpages point to a webpage, rank it higher
  - problem: easy to mess with ranking by creating lots of useless pages that point to your webpage
- **PageRank**
  - extension of citations idea
  - weight link from webpage A to webpage B by “importance” of A
  - if A has few links to it, its links are not very “valuable”
  - how do we make this into an algorithm?

# Web graph

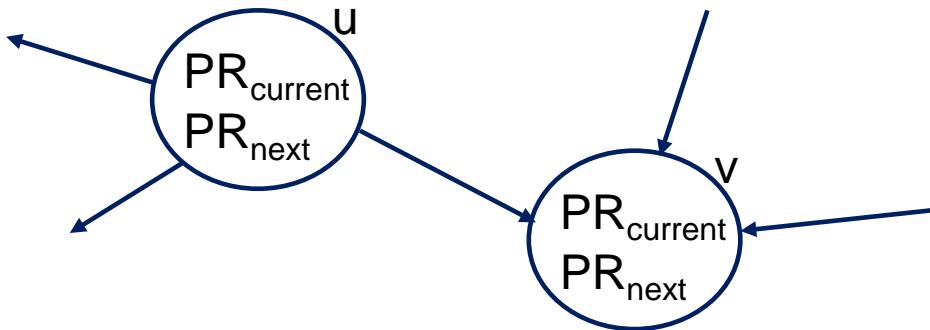


Webgraph from commoncrawl.org

- **Directed graph:** nodes represent webpages, edges represent links
  - edge from u to v represents a link in page u to page v
- **Size of graph:** commoncrawl.org (2012)
  - 3.5 billion nodes
  - 128 billion links
- **Intuitive idea of pageRank algorithm:**
  - each node in graph has a weight (pageRank) that represents its importance
  - assume all edges out of a node are equally important
  - importance of edge is scaled by the pageRank of source node

# PageRank (simple version)

Graph  $G = (V, E)$   
 $|V| = N$



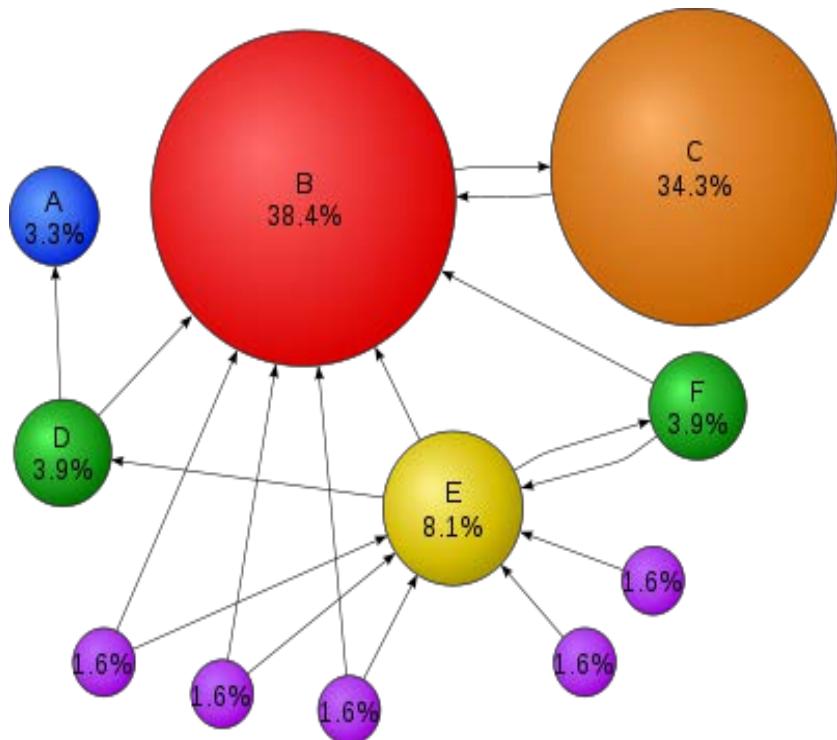
- Iterative algorithm:
  - compute a series  $PR_0, PR_1, PR_2, \dots$  of node labels
- Iterative formula:
  - $\forall v \in V. PR_0(v) = 1/N$
  - $\forall v \in V. PR_{i+1}(v) = \sum_{u \in \text{in-neighbors}(v)} \frac{PR_i(u)}{\text{out-degree}(u)}$
- Implement with two fields  $PR_{current}$  and  $PR_{next}$  in each node

# Page Rank (contd.)

- Small twist needed to handle nodes with no outgoing edges
- Damping factor:  $d$ 
  - small constant: 0.85
  - assume each node may also contribute its pageRank to a randomly selected node with probability  $(1-d)$
- Iterative formula
  - $\forall v \in V. PR_0(v) = \frac{1}{N}$
  - $\forall v \in V. PR_{i+1}(v) = \frac{1-d}{N} + d * \sum_{u \in \text{in-neighbors}(v)} \frac{PR_i(u)}{\text{out-degree}(u)}$

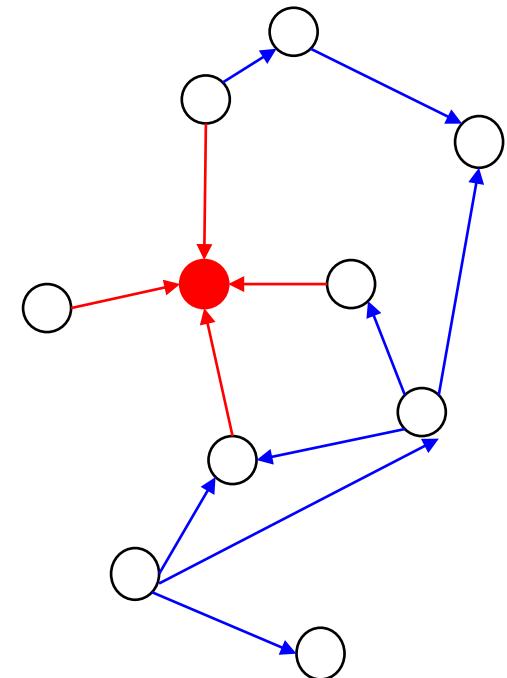
# PageRank example

- Nice example from Wikipedia
- Note
  - B and E have many in-edges but pageRank of B is much greater
  - C has only one in-edge but high pageRank because its in-edge is very valuable
- Caveat:
  - search engines use many criteria in addition to pageRank to rank webpages



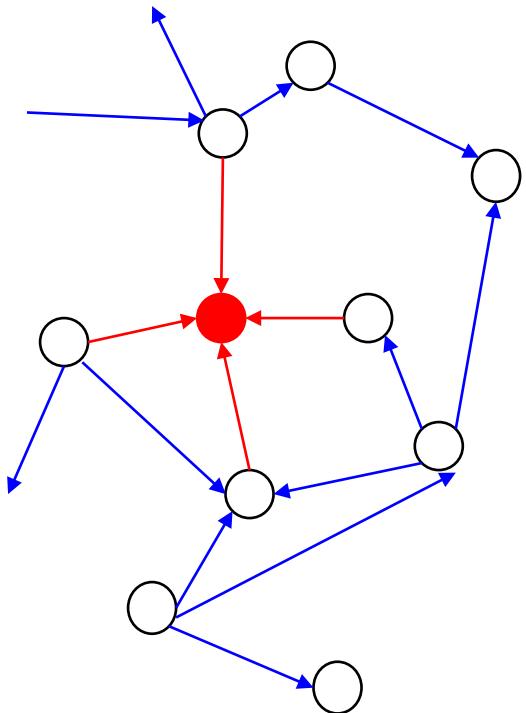
# Parallelization of pageRank

- TAO classification
  - topology: unstructured graph
  - active nodes:
    - topology-driven, unordered
  - operator: local computation
- PageRank<sub>next</sub> values at all nodes can be computed in parallel
- Which algorithm does this remind you of?
  - Jacobi iteration with 5-point stencil
  - main difference: topology
    - 5-point stencil: regular grid, uniform degree graph
    - web-graph: power-law graph
    - this has a major impact on implementation, as we will see later



# PageRank discussion

- **Vertex program (GraphLab):**
  - value at node is updated using values at immediate neighbors
  - very limited notion of neighborhood but adequate for pageRank and some ML algorithms
- **CombBlas: combinatorial BLAS**
  - generalized sparse MVM: + and \* in MVM are generalized to other operations like  $\vee$  and  $\wedge$
  - adequate for pageRank
- **Interesting application of TAO**
  - standard pageRank is topology-driven
  - can you think of a data-driven version of pageRank?

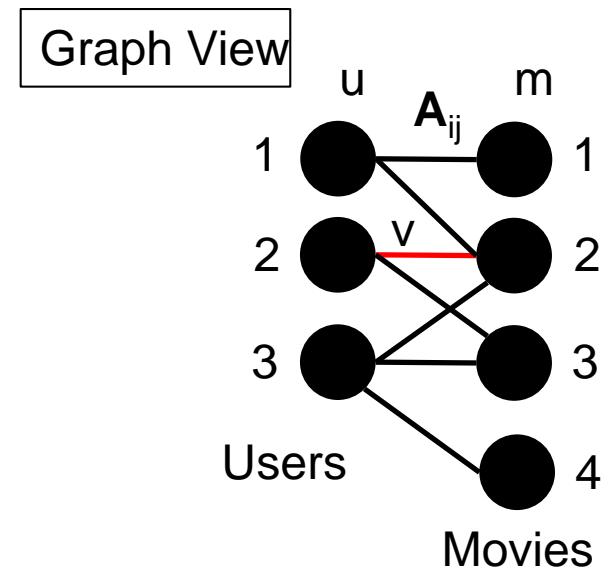
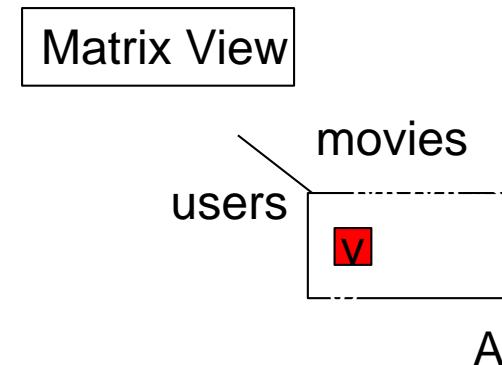


# Recommender system

- **Problem**
  - given a database of users, items, and ratings given by each user to some of the items
  - predict ratings that user might give to items he has not rated yet (usually, we are interested only in the top few items in this set)
- **Netflix challenge**
  - in 2006, Netflix released a subset of their database and offered \$1 million prize to anyone who improved their algorithm by 10%
  - triggered a lot of interest in recommender systems
  - prize finally given to BellKor's Pragmatic Chaos team in 2009

# Data structure for database

- **Sparse matrix view:**
  - rows are users
  - columns are movies
  - $A(u,m) = v$  is user u has given rating v to movie m
- **Graph view:**
  - bipartite graph
  - two sets of nodes, one for users, one for movies
  - edge  $(u,m)$  with label v
- **Recommendation problem:**
  - predict missing entries in sparse matrix
  - predict labels of missing edges in bipartite graph



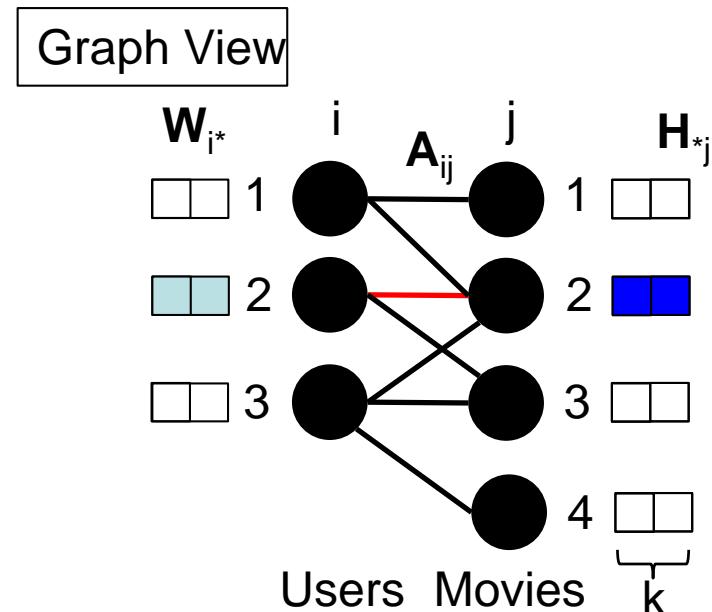
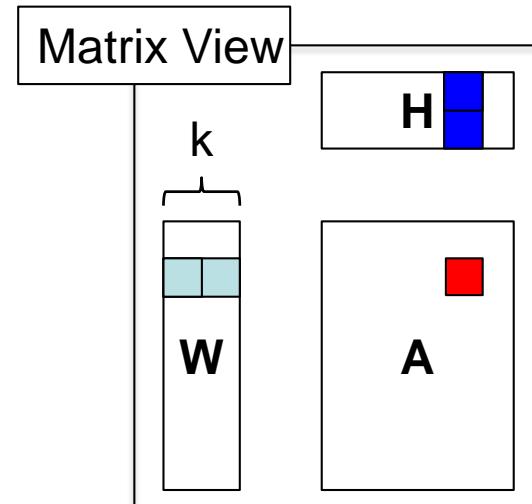
# One approach: matrix completion

- Optimization problem

- Find  $m \times k$  matrix  $\mathbf{W}$  and  $k \times n$  matrix  $\mathbf{H}$  ( $k \ll \min(m,n)$ ) such that  $\mathbf{A} \approx \mathbf{WH}$
- Low-rank approximation
- $\mathbf{H}$  and  $\mathbf{W}$  are dense so all missing values are predicted

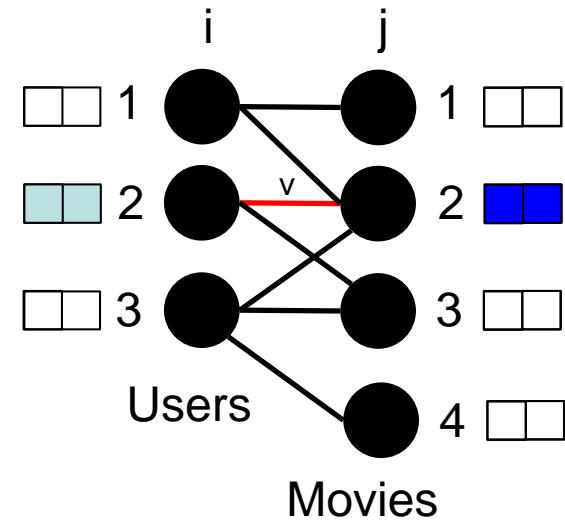
- Graph view

- Label of user nodes  $i$  is vector corresponding to row  $\mathbf{W}_{i^*}$
- Label of movie node  $j$  is vector corresponding to column  $\mathbf{H}_{*j}$
- If graph has edge  $(u,m)$ , inner product of labels on  $u$  and  $m$  must be approximately equal to label on edge



# One algorithm: SGD

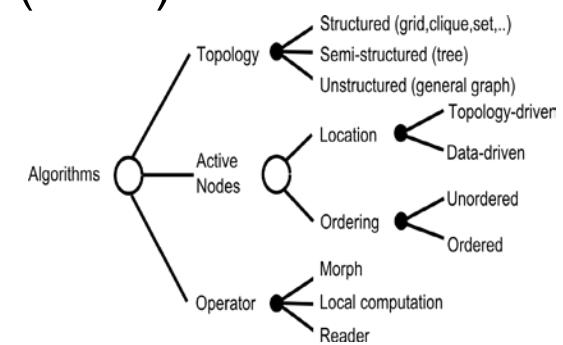
- Stochastic gradient descent (SGD)
- Iterative algorithm:
  - initialize all node labels to some arbitrary values
  - iterate until convergence
    - visit all edges  $(u,m)$  in some order and update node labels at  $u$  and  $m$  based on the residual
- TAO analysis:
  - topology: unstructured (power-law) graph
  - active edges: topology-driven, unordered
  - operator: local computation
- Parallelism in SGD:
  - edges that form a matching can be processed in parallel
- What algorithm does this remind you of?
  - Bellman-Ford



# Summary of discussion of algorithms

# What we have learned (I)

- Data-centric view of algorithms
- TAO classification
- Unordered algorithms
  - lots of parallelism for large problem sizes
  - soft priorities need in many/most algorithms (e.g. chaotic SSSP)
  - don't-care non-determinism in some algorithms (DMR)
- Topology-driven algorithms
  - iterate over data structure
  - no explicit work-list needed
- Data-driven algorithms
  - need efficient parallel work-list (put/get)
  - may need to support soft priorities (e.g. chaotic SSSP)
- Some problems
  - have both ordered and unordered algorithms (e.g. SSSP)
  - have both topology-driven and data-driven algorithms (e.g. SSSP, pageRank)
  - data-driven algorithm may be more work-efficient than topology-driven one



# What we have learned (II)

- Amorphous data-parallelism
  - data-parallelism is special case
- Parallelization strategies:
  - key question: when do you know the active nodes and neighborhoods?
    - static: known at compile-time (modulo problem size)
    - inspector-executor: after input is given but before program is executed
    - optimistic: after program has finished execution
- Implementation concepts:
  - edge matchings in graphs
  - synchronization
    - barrier synchronization: coarse-grain
    - marking graph elements, get/put on work-lists: mutual exclusion, fine-grain

# What we will study (I)

- Parallel architectures: workers can be heterogeneous and may be organized in different ways
  - vector architectures, GPUs, FPGAs
  - shared and distributed-memory architectures
- Synchronization: coordination between workers
  - coarse-grain synchronization: barriers
  - fine-grain synchronization: locks, lock-free instructions
    - application: marking of graph elements, work-lists, mutual exclusion
- Scheduling activities on workers
  - locality: temporal, spatial, network
  - load-balancing
  - minimize conflicts between concurrent activities (optimistic parallelization)
- Concurrent data structure implementations
  - graphs/sparse matrices
  - work sets/multisets
    - soft priorities
  - priority queues

# What we will study (II)

- Programming language issues
  - how do we express information about parallelism, locality, scheduling, data structure implementations?
  - how do we simplify parallel programming so most application programmers can benefit from parallelism without having to write parallel code?
- Parallel notations and libraries
  - shared-memory: pThreads, MPI, Galois
  - distributed-memory: MPI

# Lock-step execution

- With 2 workers, we can execute all tasks in 5 steps with the right schedule

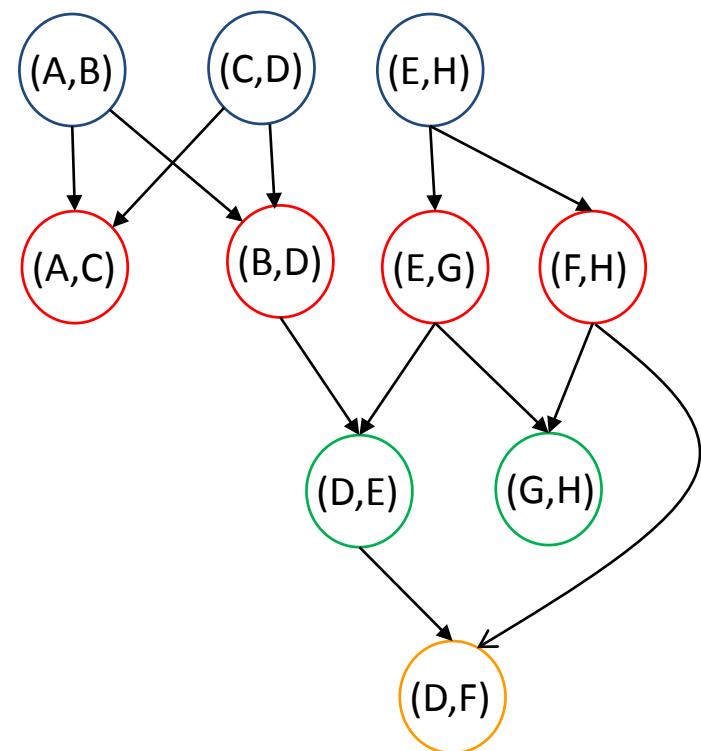
- (A,B), (E,H)
- (C,D), (E,G)
- (B,D), (F,H)
- (D,E), (G,H)
- (A,C), (D,F)

worker \ step	step →					
	0	1	2	3	4	
P0	(A,B)	(C,D)	(B,D)	(D,E)	(A,C)	
P1	(E,H)	(E,G)	(F,H)	(G,H)	(D,F)	

- Two implementations:

- synchronous execution:** assign work to workers as specified above (no need for work-lists) and use barrier synchronization between steps
- autonomous execution:**

- threads proceed independently of each other
- each node of dependence graph has an integer counter initialized to the number of in-edges
- free thread grabs any node with zero counter, executes it, and then updates counters at out-neighbors in the DAG
- updating counters needs fine-grain synchronization: must mark nodes before updating their counters



## Measurements

### Overview

- To understand and improve program performance, you need insight into program behavior on platform of interest
  - execution time of program
  - processor pipeline: stalls
  - memory hierarchy: cache accesses and misses, etc.
- Measurements (general)
  - repeatability: repeating measurement on same setup gives more or less the same results
  - replicability: performing measurement on a different but similar setup gives more or less the same results
- Measurements (computer performance)
  - basic ideas are quite simple
  - however processors are very complex so getting accurate measurements can be difficult
  - you must have a mental model of how processors execute instructions to make sensible measurements
- Libraries like PAPI simplify some measurements

## Timing your code

### Basic idea

- Assume there is a way to get “current time” on the computer
  - for now, don’t worry about precise definition of “current time”
- Timing your code
  - Use the pseudocode on right
- Problems
  - definition of “current time” can be quite subtle
  - modern computer systems are so complex that you may not be measuring what you think you are measuring
  - usually your code is written in C or some other high-level language and compiler may transform your code in unexpected ways
  - ....

```
tick = "getCurrentTime";
/*your code here */
tock = "getCurrentTime";
execTime = tock - tick;
```

## Main issues

1. **Initial conditions matter**
  - measured time may depend on state of machine when timing starts
2. **Resolution and accuracy of timer**
  - granularity of your measuring device
  - spread in measurements
3. **Heisenberg effect**
  - measurement may change quantity you are measuring
4. **Compiler optimizations**
  - may need to look at actual assembly code to make sure compiler has not modified your code in unexpected ways
5. **Context-switching by O/S and hardware interrupts**
  - you may end up measuring stuff outside your code
6. **Out-of-order execution of instructions**
  - what you measure may not be what you think you are measuring

## Main issues (1): Initial conditions

- Computers have a lot of internal state
  - caches, TLBs,...
- Internal state when measurement starts can affect execution time
  - are instructions in I-cache when measurement starts?
  - are memory locations accessed by your code in caches or memory?
    - what levels of cache?

```
tick = "getCurrentTime";
/*your code here */
tock = "getCurrentTime";
execTime = tock - tick;
```

## Main issues(2): Resolution and accuracy

- **Resolution:**
  - how small a quantity can the device measure?
  - example: you can use a tape measure to measure cloth for a suit but not to measure how wide a hydrogen atom is
- If code in R is just a few instructions, your timer may not have resolution to measure this
  - what if timer only measured milliseconds?
  - what about overhead of getCurrentTime itself?
- **Accuracy:**
  - assuming resolution is not a problem, how variable is the measurement?
  - if you repeat it ten times, how wide is the spread of measurements?

```
tick = "getCurrentTime"
/*your code here */
tock = "getCurrentTime"
execTime = tock - tick;
```

## Main issues(3): Heisenberg effect

- One solution to resolution problem:
  - put a loop around your code and execute it N times
  - divide (tock-tick) by N
- Problems:
  - loop code may change context of measurement
    - if loop counter i is allocated to a register, does that affect register allocation in your code?
    - are your instructions still in I-cache?
  - you are including loop overhead in your measurement

```
tick = "getCurrentTime"
/*your code here */
tock = "getCurrentTime"
execTime = tock - tick;
```

```
tick = "getCurrentTime"
for (int i=0;i<N;i++){
    /*your code here */
}
tock = "getCurrentTime"
execTime = (tock - tick)/N;
```

## Main issues(4): compiler optimizations

- Compiler can optimize your code in unexpected ways so you measure something different from what you are expected

- Example:

- to eliminate effect of loop overhead in previous slide, you can try to measure execTime with and without your code in the loop body
- however, compiler might optimize away the loop in the second piece of code since the loop body is empty

- Solutions

- examine assembly code to ensure compiler is not changing code in unexpected ways
- if it is, disable compiler optimizations (but this can change what you are measuring in undesirable ways)
- you can tweak code to trick compiler to stop it from doing undesirable things

```

tick = "getCurrentTime";
for (int i=0;i<N;i++){
    /*your code here */
}
tock = "getCurrentTime";
execTime1 = (tock - tick);

tick = "getCurrentTime";
for (int i=0;i<N;i++){
    /*empty loop body*/
}
tock = "getCurrentTime";
execTime2 = (tock - tick);

myCodeTime =
    (execTime1 - execTime2)/N;

```

## Main issues(5): Process-switching

- Code in R may not be executed in one shot by OS and processor

- OS may de-schedule your process while executing R, schedule code from other processes, and then get back to executing code from R

- This may happen many times during execution of R

- Analogy:

- taking an exam vs. doing an assignment
- What is getCurrentTime measuring?
- if it is elapsed time like "wall-clock time", process switches will confound your measurement

- Solutions:

- disable process switches and interrupts before executing code in R (but you may not be able to do this in user mode)
- find a timer that advances only when processor is executing your program
  - but context-switches may still pollute your caches

```

tick = "getCurrentTime"
/*your code here */
tock = "getCurrentTime"
execTime = tock - tick

```

## Main issues(6): Out-of-order execution of instructions

- Modern processors execute instructions out of program order

- but ensure dependences are satisfied

- Problem:

- code from region R may get executed outside of tick and tock
- code from outside region R may get executed between tick and tock

- Solution:

- need to insert **serializing instructions** around region R
  - "fence off" instructions being timed from other instructions
- similar to memory fences but for instructions of all types, not just memory operations

```

tick = "getCurrentTime"
/*your code here */
tock = "getCurrentTime"
execTime = tock - tick

```

## Drilling down

- Key questions:

- What can we use for "getCurrentTime" and what is its resolution?
- How do we avoid timing errors from process-switches and interrupts?
- How do we insert serialization instructions at tick and tock?

```

tick = "getCurrentTime"
/*your code here */
tock = "getCurrentTime"
execTime = tock - tick

```

- Answer is very system-dependent but we will discuss two solutions for C/Linux/x86:

- Linux call: clock\_gettime
- x86 code

## clock\_gettime

```
#include <time.h>
struct timespec { time_t tv_sec; /* seconds */ long tv_nsec; /* nanoseconds */ };
int clock_gettime(clockid_t clk_id, struct timespec *tp)
int clock_getres (clockid_t clk_id, struct timespec *res)
```

- **timespec**
  - type for time measurement
  - two fields:
    - tv\_sec (seconds)
    - tv\_nsec (nanoseconds)
  - to get total time in nanoseconds, multiple tv\_sec by a billion and add to tv\_nsec
- **clock\_gettime**
  - first argument: which clock?
  - some choices:
    - CLOCK\_REALTIME: systemwide, real-time clock
    - CLOCK\_PROCESS\_CPUTIME\_ID: high-resolution (nanosecond) timer for process
    - CLOCK\_THREAD\_CPUTIME\_ID: high-resolution (nanosecond) timer for thread

```
#include <stdio.h> /* for printf */
#include <stdint.h> /* for uint64 */
#include <time.h> /* for clock_gettime */

main(int argc, char **argv)
{ uint64_t execTime; /* time in nanoseconds */
  struct timespec tick, tock;

  clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &tick);
  /* do stuff */
  clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &tock);

  execTime = 1000000000 * (tock.tv_sec - tick.tv_sec) + tock.tv_nsec - tick.tv_nsec;
  printf("elapsed process CPU time = %llu nanoseconds\n", (long long unsigned int) execTime);
}
```

Implementation of `clock_gettime` should use serialization instructions.  
`CLOCK_PROCESS_CPUTIME_ID` measures the amount of time spent in this process.  
Resolution on systems I used is 1 nanosecond.  
Even if /\*do stuff \*/ is empty, `execTime` is about 2000 nanosec on these systems.

## x86 code

- **Getting time:**
  - **TSC**: 64-bit time-stamp counter that tracks cycles
  - **RDTSC** instruction: read time-stamp counter
    - EDX  $\leftarrow$  high-order 32 bits of counter
    - EAX  $\leftarrow$  low-order 32 bits of counter
    - no serialization guarantee
  - **RDTSCLP** instruction
    - waits until all previous instructions have been executed before reading counter
    - however following instructions may begin execution before read is performed
- **Serialization instruction:**
  - CPUID instruction
    - modifies EAX, EBX, ECX, EDX registers
    - can be executed at any privilege level

## Further reading

- **Linux man pages:**
  - describes `clock_gettime` and other clocks
  - [https://linux.die.net/man/3/clock\\_gettime](https://linux.die.net/man/3/clock_gettime)
- **Technical note from Intel:**
  - shows how to use RDTSC and CPUID for accurate timing measurements
  - [www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf](http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf)

## PAPI counters

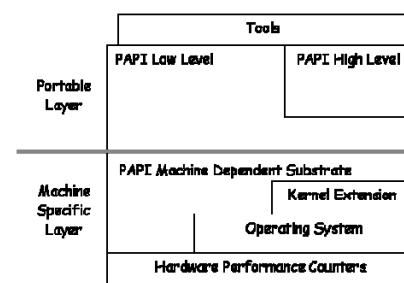
### Hardware counters

- Modern CPUs have hardware counters for many events
  - Cycles
  - Instructions
  - Floating-point instructions
  - Loads and stores
  - I-cache misses
  - L1 data cache misses
  - L2 data cache misses
  - TLB misses
  - Pipeline stalls
  - .....
- Complications
  - accessing counters directly can be complex
  - code is not portable
  - on many processors, fewer hardware counters than events you can track so only a subset of events can be measured in a given run

## PAPI

- Performance Application Programming Interface
- Two interfaces to underlying counter hardware:
  - High-level interface: provides ability to start, stop and read counters for a specified list of events
  - Low-level interface: manages hardware events in user-defined groups called EventSets
- Timers and system information
- C and Fortran bindings
- PAPI interface to performance counters supported in the Linux 2.6.31 kernel
- User guide:  
[http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI\\_USER\\_GUIDE\\_23.htm](http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI_USER_GUIDE_23.htm)

### PAPI design



## PAPI Events

- **Preset events**
  - platform-independent names for events deemed useful for performance tuning
  - examples: accesses to the memory hierarchy, cache coherence protocol events, cycle and instruction counts, functional unit and pipeline utilization
  - run PAPI papi\_avail utility to determine preset events available on platform
- **PAPI also provides access to native events through low-level interface**
  - may be platform-specific

## PAPI preset events

- PAPI\_L1\_DCM: Level 1 data cache misses
- PAPI\_L1\_DCA: Level 1 data cache accesses
- PAPI\_L1\_ICM: Level 1 I-cache misses
- PAPI\_L2\_DCM: Level 2 data cache misses
- PAPI\_L3\_DCM: Level 3 data cache misses
- PAPI\_FXU\_IDLE: cycles floating-point units are idle
- PAPI\_TOT\_INS: total instructions executed
- PAPI\_TOT\_CYC: total cycles
- PAPI\_IPS: instructions executed per second
- .....

## PAPI\_query\_event

- Check whether CPU can measure the PAPI event you are interested in

```
if (PAPI_OK != PAPI_query_event(PAPI_TOT_INS))
    ehandler("Cannot count PAPI_TOT_INS.");
if (PAPI_OK != PAPI_query_event(PAPI_L1_DCM))
    ehandler("Cannot count PAPI_L1_DCM.");
if (PAPI_OK != PAPI_query_event(PAPI_L2_DCM))
    ehandler("Cannot count PAPI_L2_DCM.");
```

## High Level API

- Meant for application programmers wanting simple but accurate measurements
  - calls the lower level API
- **Eight important functions:**
  - PAPI\_num\_counters
    - how many hardware counters are supported?
  - PAPI\_start\_counters
  - PAPI\_stop\_counters
  - PAPI\_read\_counters
  - PAPI\_accum\_counters
    - adds counters into accumulator array and zeroes them
  - PAPI\_flops
    - floating-point operations per second
  - PAPI\_flops
    - floating-point instructions per second
  - PAPI\_lpc
    - instructions per cycle

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <papi.h>

int main( int argc, char *argv[] ) {
    int i, j, k;

    long long counters[3];
    int PAPI_events[] = {
        PAPI_TOT_CYC,
        PAPI_L2_DCM,
        PAPI_L2_DCA };
    PAPI_library_init(PAPI_VER_CURRENT);

    i = PAPI_start_counters( PAPI_events, 3 );

    /* your code here */

    PAPI_read_counters( counters, 3 );

    printf("%lld L2 cache misses (%.3lf%% misses) in %lld cycles\n",
           counters[1],
           (double)counters[1] / (double)counters[2],
           counters[0] );
}

return 0;
}
```

## Summary

- **Measurement**
  - basic ideas are quite simple
  - however processors are very complex so getting accurate measurements can be difficult
- **You must have a mental model of how processors execute instructions**
- **Libraries like PAPI simplify some measurements**

## Scheduling

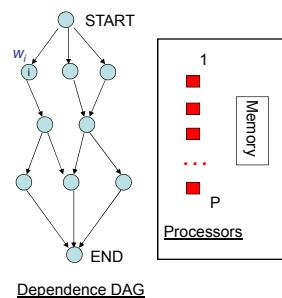
Keshav Pingali  
University of Texas, Austin

### Goal of lecture

- So far, we have studied
  - how parallelism and locality arise in programs
  - ordering constraints between tasks for correctness or efficiency
- This lecture: How do we assign tasks to workers?
  - multicore: workers might be cores
  - distributed-memory machines: workers might be hosts/machines
- Scheduling
  - rich literature exists for dependence graph scheduling
  - most of it is not very useful in practice since they use unrealistic program and machine models
    - (e.g.) assume task execution times are known
  - nevertheless, it is useful to study it since it gives us intuition for what the issues are for scheduling in practice

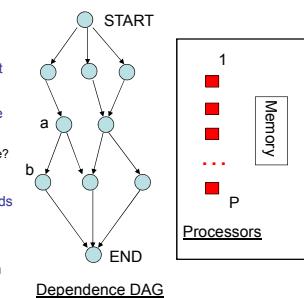
## Dependence DAG's

- DAG with START and END nodes
  - all nodes reachable from START
  - END reachable from all nodes
  - START and END are not essential
- Nodes are computations
  - each computation can be executed by a processor in some number of time-steps
  - computation may require reading/writing shared-memory
  - node weight: time taken by a processor to perform that computation
  - $w_i$  is weight of node  $i$
- Edges are precedence constraints
  - nodes other than START can be executed only after immediate predecessors in graph have been executed
  - known as dependences
- Very old model:
  - PERT charts (late 50's):
    - Program Evaluation and Review technique
    - developed by US Navy to manage Polaris submarine contracts



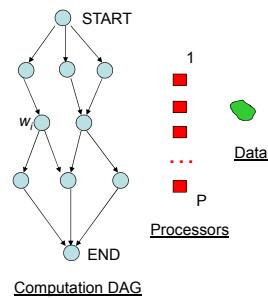
## Computer model

- P identical processors
  - processors have local memory
  - all shared-data is stored in global memory
- How does a processor know which nodes it must execute?
  - work assignment
- How does a processor know when it is safe to execute a node?
  - (eg) if P1 executes node a and P2 executes node b, how does P2 know when P1 is done?
  - synchronization
- For now, let us defer these questions
- In general, time to execute program depends on work assignment
  - for now, assume only that if there is an idle processor and a ready node, that node is assigned immediately to an idle processor
- $T_p$  = best possible time to execute program on P processors



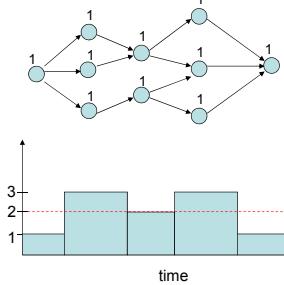
## Work and critical path

- **Work** =  $\ll_1 w_i$ 
  - time required to execute program on one processor =  $T_1$
- **Path weight**
  - sum of weights of nodes on path
- **Critical path**
  - path from START to END that has maximal weight
  - this work must be done sequentially, so you need this much time regardless of how many processors you have
  - call this  $T_4$



## Terminology

- **Instantaneous parallelism**  
 $IP(t) = \max$  number of processors that can be kept busy at each point in execution of algorithm
- **Maximal parallelism**  
 $MP = \text{highest } IP(t)$
- **Average parallelism**  
 $AP = T_1/T_4$
- These are properties of the computation DAG, not of the machine or the work assignment



## Computing critical path etc.

- **Algorithm for computing earliest start times of nodes**
  - Keep a value called minimum-start-time (mst) with each node, initialized to 0
  - Do a topological sort of the DAG
    - ignoring node weights
  - For each node  $n$  ( $\neq$  START) in topological order
    - for each node  $p$  in predecessors( $n$ )
      - $mst_n = \max(mst_n, mst_p + w_p)$
- **Complexity** =  $O(|V|+|E|)$
- Critical path and instantaneous, maximal and average parallelism can easily be computed from this

## Speed-up

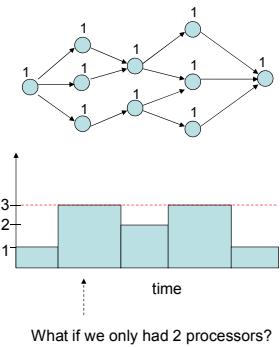
- **Speed-up( $P$ )** =  $T_1/T_P$ 
  - intuitively, how much faster is it to execute program on  $P$  processors than on 1 processor?
- **Bound on speed-up**
  - regardless of how many processors you have, you need at least  $T_4$  units of time
  - speed-up( $P$ )  $\leq T_1/T_4 = \ll_1 w_i / CP = AP$

## Amdahl's law

- **Amdahl:**
  - suppose a fraction  $p$  of a program can be done in parallel
  - suppose you have an unbounded number of parallel processors and they operate infinitely fast
  - speed-up will be at most  $1/(1-p)$ .
- **Follows trivially from previous result.**
- **Plug in some numbers:**
  - $p = 90\% \rightarrow$  speed-up  $\approx 10$
  - $p = 99\% \rightarrow$  speed-up  $\approx 100$
- **To obtain significant speed-up, most of the program must be performed in parallel**
  - serial bottlenecks can really hurt you

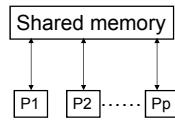
## Scheduling

- Suppose  $P \leq MP$
- There will be times during the execution when only a subset of "ready" nodes can be executed.
- Time to execute DAG can depend on which subset of  $P$  nodes is chosen for execution.
- To understand this better, it is useful to have a more detailed machine model



## Machine Model

- Processors operate synchronously (in lock-step)
  - barrier synchronization in hardware
  - if a processor has reached step  $i$ , it can assume all other processors have completed tasks in all previous steps
- Each processor has private memory



## Schedules

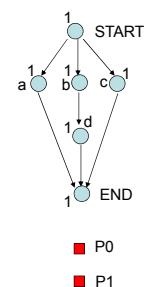
**Schedule:** function from node to (processor, start time)  
Also known as "space-time mapping"

**Schedule 1**

	0	1	2	3	4
time	0	1	2	3	4
P0	START	a	c	END	
P1		b	d		

**Schedule 2**

	0	1	2	3	4
time	0	1	2	3	4
P0	START	a	b	d	END
P1		c			



Intuition: nodes along the critical path should be given preference in scheduling

## Optimal schedules

- **Optimal schedule**
  - shortest possible schedule for a given DAG and the given number of processors
- **Complexity of finding optimal schedules**
  - one of the most studied problems in CS
- **DAG is a tree:**
  - level-by-level schedule is optimal (Aho, Hopcroft)
- **General DAGs**
  - variable number of processors (number of processors is input to problem): NP-complete
  - fixed number of processors
    - 2 processors: polynomial time algorithm
    - 3,4,...: complexity is unknown!
- **Many heuristics available in the literature**

## Heuristic: list scheduling

- **Maintain a list of nodes that are ready to execute**
  - all predecessor nodes have completed execution
- **Fill in the schedule cycle-by-cycle**
  - in each cycle, choose nodes from ready list
  - use heuristics to choose “best” nodes in case you cannot schedule all the ready nodes
- **One popular heuristic:**
  - assign node priorities before scheduling
  - priority of node n:
    - weight of maximal weight path from n to END
    - intuitively, the “further” a node is from END, the higher its priority

## List scheduling algorithm

```

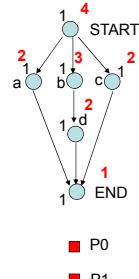
cycle c = 0;
ready-list = {START};
inflight-list = {};
while (|ready-list|+|inflight-list| > 0) {
    for each node n in ready-list in priority order { //schedule new tasks
        if (a processor is free at this cycle) {
            remove n from ready-list and add to inflight-list;
            add node to schedule at time cycle;
        }
        else break;
    }
    c = c + 1; //increment time
    for each node n in inflight-list { //determine ready tasks
        if (n finishes at time cycle) {
            remove n from inflight-list;
            add every ready successor of n in DAG to ready-list
        }
    }
}
  
```

## Example

	time	0	1	2	3	4
space	P0	START	a	c	END	
	P1		b	d		

Heuristic picks the good schedule

Not always guaranteed to produce optimal schedule  
(otherwise we would have a polynomial time algorithm!)



## Generating dependence graphs

- How do we produce dependence graphs in the first place?
- Two approaches
  - specify DAG explicitly
    - parallel programming
    - easy to make mistakes
      - data races: two tasks that write to same location but are not ordered by dependence
  - by compiler analysis of sequential programs
- Let us study the second approach
  - called **dependence analysis**

## Data dependence

- Basic blocks
  - straight-line code
- Nodes represent statements
- Edge  $S_1 \rightarrow S_2$ 
  - flow dependence (read-after-write (RAW))
    - $S_1$  is executed before  $S_2$  in basic block
    - $S_1$  writes to a variable that is read by  $S_2$
  - anti-dependence (write-after-read (WAR))
    - $S_1$  is executed before  $S_2$  in basic block
    - $S_1$  reads from a variable that is written by  $S_2$
  - output-dependence (write-after-write (WAW))
    - $S_1$  is executed before  $S_2$  in basic block
    - $S_1$  and  $S_2$  write to the same variable
  - input-dependence (read-after-read (RAR)) (usually not important)
    - $S_1$  is executed before  $S_2$  in basic block
    - $S_1$  and  $S_2$  read from the same variable



## Conservative approximation

- In real programs, we often cannot determine precisely whether a dependence exists
  - in example,
    - $i = j$ : dependence exists
    - $i \neq j$ : dependence does not exist
  - dependence may exist for some invocations and not for others
- **Conservative approximation**
  - when in doubt, assume dependence exists
  - at the worst, this will prevent us from executing some statements in parallel even if this would be legal
- **Aliasing:** two program names for the same storage location
  - (e.g.)  $X(i)$  and  $X(j)$  are *may*-aliases
  - may-aliasing is the major source of imprecision in dependence analysis

Example  
procedure f(X,i,j)  
begin  
  X(i) = 10;  
  X(j) = 5;  
end

## Putting it all together

- Write sequential program.
- Compiler produces parallel code
  - generates control-flow graph
  - produces computation DAG for each basic block by performing dependence analysis
  - generates schedule for each basic block
    - use list scheduling or some other heuristic
    - branch at end of basic block is scheduled on all processors
- **Problem:**
  - average basic block is fairly small (~ 5 RISC instructions)
- **One solution:**
  - transform the program to produce bigger basic blocks

## One transformation: loop unrolling

- Original program
- Unroll loop 4 times: not very useful!

```
for i = 1,100  
  X(i) = i  
  
for i = 1,100,4  
  X(i) = i  
  i = i+1  
  X(i) = i  
  i = i+1  
  X(i) = i  
  i = i+1  
  X(i) = i
```

## Smarter loop unrolling

- Use new name for loop iteration variable in each unrolled instance

```
for i = 1,100,4  
  X(i) = i  
  i1 = i+1  
  X(i1) = i1  
  i2 = i+2  
  X(i2) = i2  
  i3 = i+3  
  X(i3) = i3
```

## Array dependence analysis

- If compiler can also figure out that  $X(i)$ ,  $X(i+1)$ ,  $X(i+2)$ , and  $X(i+3)$  are different locations, we get the following dependence graph for the loop body

```
for i = 1,100,4  
  X(i) = i  
  i1 = i+1  
  X(i1) = i1  
  i2 = i+2  
  X(i2) = i2  
  i3 = i+3  
  X(i3) = i3
```

## Array dependence analysis (contd.)

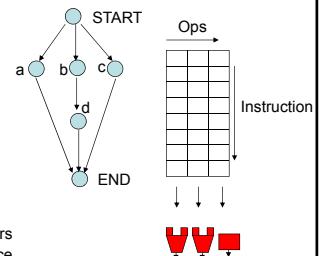
- We will study techniques for array dependence analysis later in the course
- Problem can be formulated as an integer linear programming problem:
  - Is there an integer point within a certain polyhedron derived from the loop bounds and the array subscripts?

## Two applications

- **Static scheduling**
  - create space-time diagram at compile-time
  - VLIW code generation
- **Dynamic scheduling**
  - create space-time diagram at runtime
  - multicore scheduling for dense linear algebra

## Scheduling instructions for VLIW machines

- Processors → functional units
- Local memories → registers
- Global memory → memory
- Time → instruction
- Nodes in DAG are operations (load/store/add/mul/branch/..)
  - **instruction-level parallelism**
- List scheduling
  - useful for scheduling code for pipelined, superscalar and VLIW machines
  - used widely in commercial compilers
  - loop unrolling and array dependence analysis are also used widely



## Historical note on VLIW processors

- Ideas originated in late 70's-early 80's
- Two key people:
  - Bob Rau (Stanford, UIUC, TRW, Cydrome, HP)
  - Josh Fisher (NYU, Yale, Multiflow, HP)
- **Bob Rau's contributions:**
  - transformations for making basic blocks larger:
    - predication
    - software pipelining
  - hardware support for these techniques
    - predicated execution
    - rotating register files
  - most of these ideas were later incorporated into the Intel Itanium processor
- **Josh Fisher:**
  - transformations for making basic blocks larger:
    - trace scheduling: uses key idea of **branch probabilities**
  - Multiflow compiler used loop unrolling



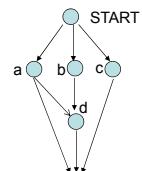
Bob Rau



Josh Fisher

## DAG scheduling for multicores

- **Reality:**
  - hard to build single cycle memory that can be accessed by large numbers of cores
- **Architectural change**
  - decouple cores so there is no notion of a global step
  - each core/processor has its own PC and cache
  - memory is accessed independently by each core
- **New problem:**
  - since cores do not operate in lock-step, how does a core know when it is safe to execute a node?
- **Solution: software synchronization**
  - counter associated with each DAG node
  - decremented when predecessor task is done
- **Software synchronization increases overhead of parallel execution**
  - cannot afford to synchronize at the instruction level
  - nodes of DAG must be coarse-grain: loop iterations



P0: a  
P1: b  
P2: c  
P3: d

How does P2 know when P0 and P1 are done?

## Increasing granularity: Block Matrix Algorithms

Original matrix multiplication

```
for I = 1,N
  for J = 1,N
    for K = 1,N
      C(I,J) = C(I,J)+A(I,K)*B(K,J)
```

*Block (tiled) matrix multiplication*

```
for IB = 1,N step B
  for JB = 1,N step B
    for KB = 1,N step B
      parallel loops
      for I = IB, IB+B-1
        for J = JB, JB+B-1
          for K = KB, KB+B-1
            C(I,J) = C(I,J)+A(I,K)*B(K,J)
```

$A_{00}$	$A_{01}$	$B_{00}$	$B_{01}$
$A_{10}$	$A_{11}$	$B_{10}$	$B_{11}$

$A_{00}$	$A_{01}$	$C_{00}$	$C_{01}$
$A_{10}$	$A_{11}$	$C_{10}$	$C_{11}$

$$\begin{aligned} C_{00} &= A_{00} * B_{00} + A_{01} * B_{10} \\ C_{01} &= A_{00} * B_{11} + A_{01} * B_{01} \\ C_{11} &= A_{11} * B_{01} + A_{10} * B_{01} \\ C_{10} &= A_{10} * B_{00} + A_{11} * B_{10} \end{aligned}$$

## New problem

- Difficult to get accurate execution times of coarse-grain nodes
  - conditional inside loop iteration
  - cache misses
  - exceptions
  - O/S processes
  - ....
- Solution: runtime scheduling**

## Example: DAGuE

- Dongarra et al (UTK)
- Programming model for specifying DAGs for parallel blocked dense linear algebra codes
  - nodes: block computations
  - DAG edges specified by programmer (see next slides)
- Runtime system**
  - keeps track of ready nodes
  - assigns ready nodes to cores
  - determines if new nodes become ready when a node completes

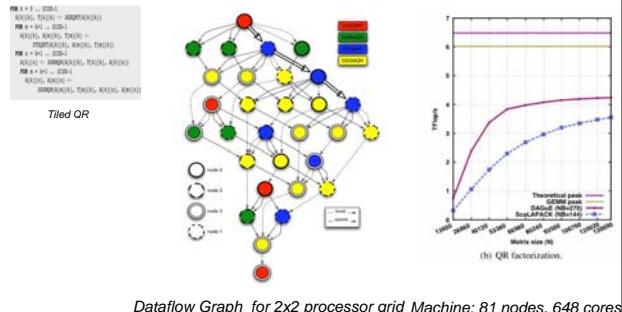
## DAGuE: Tiled QR (1)

```
FOR k = 0 .. SIZE-1
  A[k][k], T[k][k] <- DGEQRT(A[k][k])
  FOR m = k+1 .. SIZE-1
    A[k][k], A[m][k], T[m][k] <-
      DTQR(A[k][k], A[m][k], T[m][k])
  FOR n = k+1 .. SIZE-1
    A[k][n] <- DORMQR(A[k][k], T[k][k], A[k][n])
    FOR m = k+1 .. SIZE-1
      A[k][n], A[m][n] <-
        DSSMQR(A[m][k], T[m][k], A[k][n], A[m][n])
```

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,4}$	$A_{0,5}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$	$A_{1,5}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$	$A_{2,5}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$	$A_{3,5}$
$A_{4,0}$	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$	$A_{4,5}$
$A_{5,0}$	$A_{5,1}$	$A_{5,2}$	$A_{5,3}$	$A_{5,4}$	$A_{5,5}$

Tiled QR (using tiles and in/out notations)

## DAGuE: Tiled QR (2)



33

## Summary of multicore scheduling

- **Assumptions**

- DAG of tasks is known
- each task is “heavy-weight” and executing task on one worker exploits adequate locality
- no assumptions about runtime of tasks
- no lock-step execution of processors or synchronous global memory

- **Scheduling**

- keep a work-list of tasks that are ready to execute
- use heuristic priorities to choose from ready tasks

## Summary

- **Dependence graphs**
  - nodes are computations
  - edges are dependences
- **Static dependence graphs:** obtained by
  - studying the algorithm
  - analyzing the program
- **Limits on speed-ups**
  - critical path
  - Amdahl’s law
- **DAG scheduling**
  - heuristic: list scheduling (many variations)
  - static and dynamic scheduling
  - applications: VLIW code generation, multicore scheduling for dense linear algebra
- **Major limitations:**
  - works for topology-driven algorithms with fixed neighborhoods since we know tasks and dependences before executing program
  - not very useful for data-driven algorithms since tasks are created dynamically
    - one solution: work-stealing, work-sharing. Study later.

Cache Models  
and  
Program Transformations

# Goal of lecture

- Develop abstractions of real caches for understanding program performance
- Study the cache performance of matrix-vector multiplication (MVM)
  - simple but important computational science kernel
- Understand MVM program transformations for improving performance

# Matrix-vector product

- **Code:**

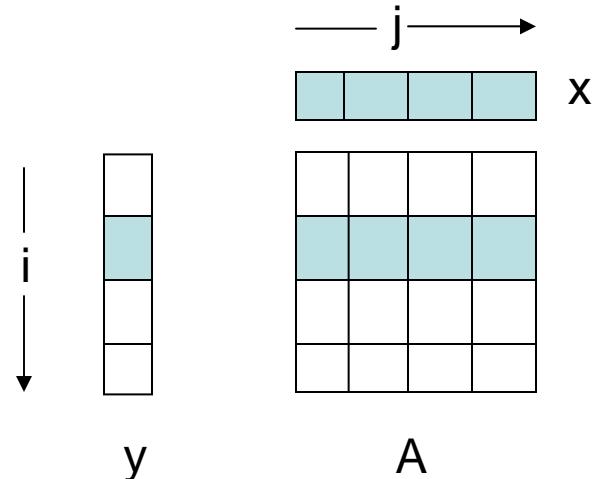
```
for i = 1,N  
  for j = 1,N  
    y(i) = y(i) + A(i,j)*x(j)
```

- **Total number of references =  $4N^2$**

- This assumes that all elements of  $A, x, y$  are stored in memory
- Smart compilers nowadays can register-allocate  $y(i)$  in the inner loop
- You can get this effect manually

```
for i = 1,N  
  temp = y(i)  
  for j = 1,N  
    temp = temp + A(i,j)*x(j)  
  y(i) = temp
```

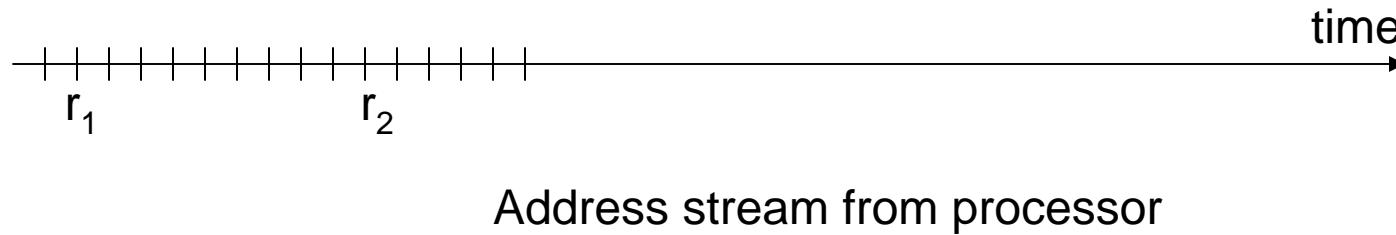
- To keep things simple, we will not do this but our approach applies to this optimized code as well



# Cache abstractions

- Real caches are very complex
- Science is all about tractable and useful abstractions (models) of complex phenomena
  - models are usually approximations
- Can we come up with cache abstractions that are both tractable and useful?
- Focus:
  - two-level memory model: cache + memory

# Stack distance



- $r_1, r_2$  : two memory references
  - $r_1$  occurs earlier than  $r_2$
- $\text{stackDistance}(r_1, r_2)$ : number of distinct cache lines referenced between  $r_1$  and  $r_2$
- Stack distance was defined by Mattson et al (IBM Systems Journal paper)
  - arguably the most important paper in locality

# Modeling approach

- First approximation:
  - ignore conflict misses
  - only cold and capacity misses
- Most problems have some notion of “problem size”
  - (eg) in MVM, the size of the matrix ( $N$ ) is a natural measure of problem size
- Question: how does the miss ratio change as we increase the problem size?
- Even this is hard, but we can often estimate miss ratios at two extremes
  - **large cache model**: problem size is small compared to cache capacity
  - **small cache model**: problem size is large compared to cache capacity
  - we will define these more precisely in the next slide.

# Large and small cache models

- Large cache model
  - no capacity misses
  - only cold misses
- Small cache model
  - cold misses: first reference to a line
  - capacity misses: possible for succeeding references to a line
    - let  $r_1$  and  $r_2$  be two successive references to a line
    - assume  $r_2$  will be a capacity miss if  $\text{stackDistance}(r_1, r_2)$  is some function of problem size
    - argument: as we increase problem size, the second reference will become a miss sooner or later
- For many problems, we can compute
  - miss ratios for small and large cache models
  - problem size transition point from large cache model to small cache model

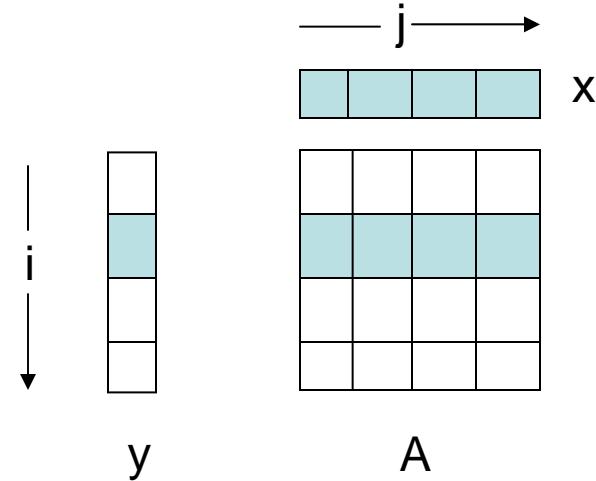
# MVM study

- We will study five scenarios
  - Scenario I
    - i,j loop order, line size = 1 number
  - Scenario II
    - j,i loop order, line size = 1 number
  - Scenario III
    - i,j loop order, line size = b numbers
  - Scenario IV
    - j,i loop order, line size = b numbers
  - Scenario V
    - blocked code, line size = b numbers

# Scenario I

- Code:

```
for i = 1,N
    for j = 1,N
        y(i) = y(i) + A(i,j)*x(j)
```
- Inner loop is known as DDOT in NA literature if working on doubles:
  - Double-precision DOT product
- Cache line size
  - 1 number
- Large cache model:
  - Misses:
    - A:  $N^2$  misses
    - x: N misses
    - y: N misses
    - Total =  $N^2+2N$
    - Miss ratio =  $(N^2+2N)/4N^2$   
 $\sim 0.25 + 0.5/N$

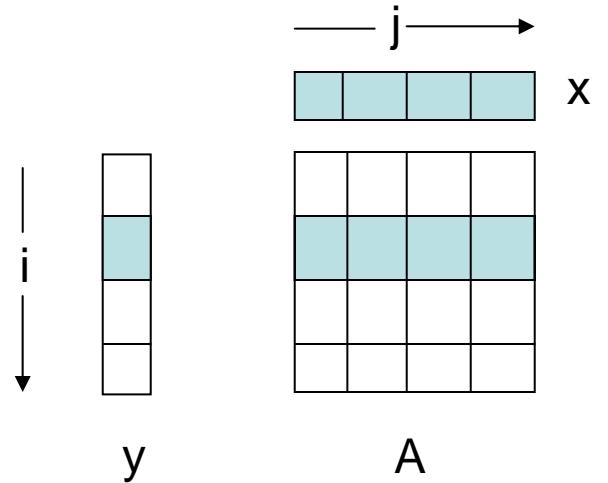


# Scenario I (contd.)

Address stream:  $y(1) \ A(1,1) \ x(1) \ y(1)$   $y(1) \ A(1,2) \ x(2) \ y(1)$  ...  $y(1) \ A(1,N) \ x(N) \ y(1)$   $y(2) \ A(2,1) \ x(1) \ y(2)$

- **Small cache model:**

- A:  $N^2$  misses
- x:  $N + N(N-1)$  misses (reuse distance= $O(N)$ )
- y:  $N$  misses (reuse distance= $O(1)$ )
- Total =  $2N^2+N$
- Miss ratio =  $(2N^2+N)/4N^2$   
 $\sim 0.5 + 0.25/N$



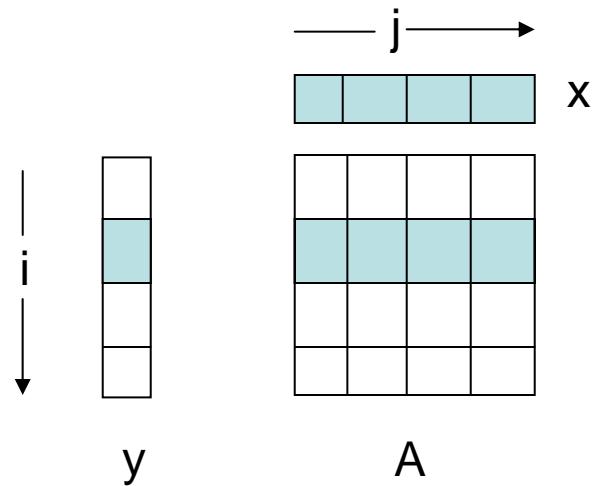
- **Transition from large cache model to small cache model**

- As problem size increases, when do capacity misses begin to occur?
- Subtle issue: depends on replacement policy (see next slide)

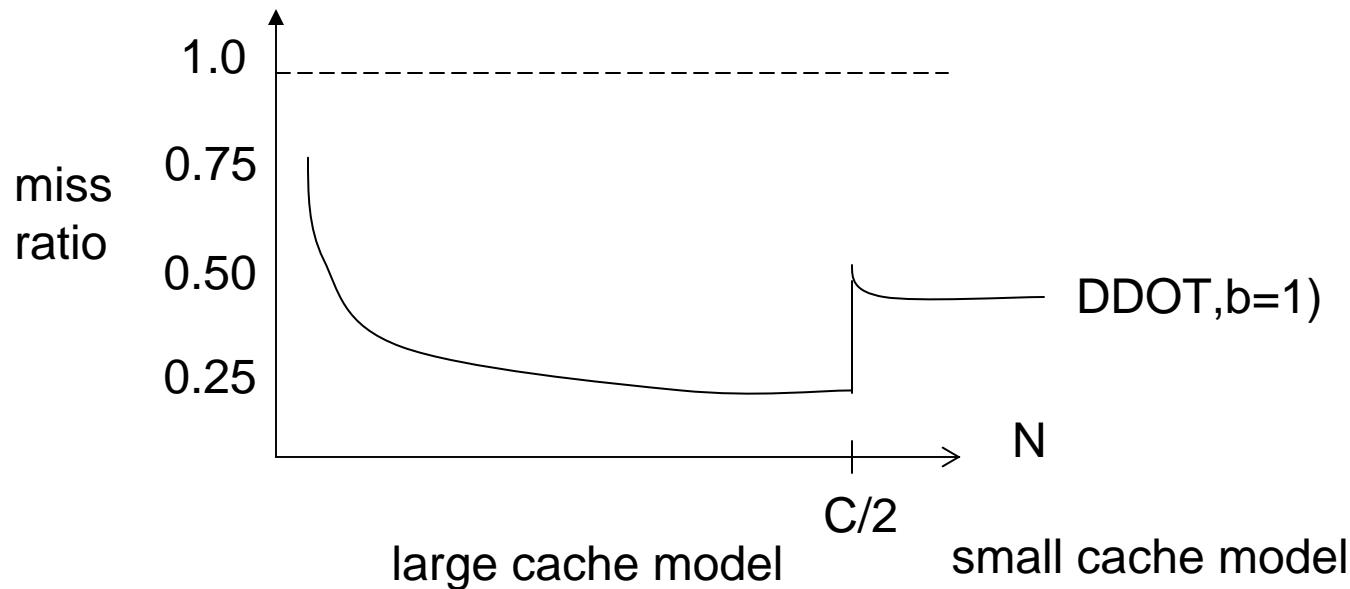
# Scenario I (contd.)

Address stream:  $y(1) \ A(1,1) \ x(1) \ y(1)$   $y(1) \ A(1,2) \ x(2) \ y(1)$  ...  $y(1) \ A(1,N) \ x(N) \ y(1)$   $y(2) \ A(2,1) \ x(1) \ y(2)$

- Question: as problem size increases, when do capacity misses begin to occur?
- Depends on replacement policy:
  - Optimal replacement:
    - do the best job you can, knowing everything about the computation
    - only  $x$  needs to be cache-resident
    - elements of  $A$  can be “streamed in” and tossed out of cache after use
    - So we need room for  $(N+2)$  numbers
    - Transition:  $N+2 > C \rightarrow N \sim C$
  - LRU replacement
    - by the time we get to end of a row of  $A$ , first few elements of  $x$  are “cold” but we do not want them to be replaced
    - Transition:  $(2N+2) > C \rightarrow N \sim C/2$
- Note:
  - optimal replacement requires perfect knowledge about future
  - most real caches use LRU or something close to it
  - some architectures support “streaming”
    - in hardware
    - in software: hints to tell processor not to cache certain references



# Miss ratio graph



- Jump from large cache model to small cache model will be more gradual in reality because of conflict misses

# Scenario II

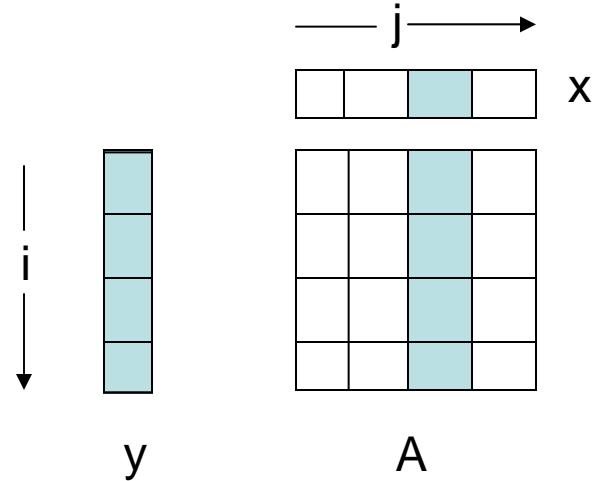
- Code:

```
for j = 1,N  
  for i = 1,N  
    y(i) = y(i) + A(i,j)*x(j)
```

- Inner loop is known as AXPY in NA literature

$$\mathbf{y} = \alpha \cdot \mathbf{x} + \mathbf{y}$$

- Miss ratio picture exactly the same as Scenario I
  - roles of x and y are interchanged



# Scenario III

- **Code:**

```
for i = 1,N  
  for j = 1,N  
    y(i) = y(i) + A(i,j)*x(j)
```

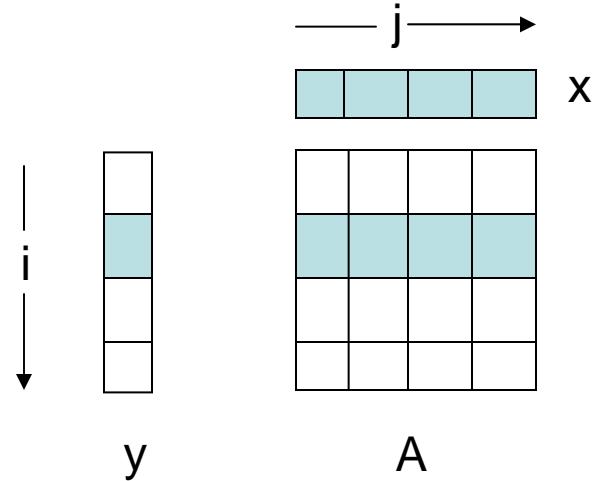
- **Cache line size**

- b numbers

- **Large cache model:**

- Misses:

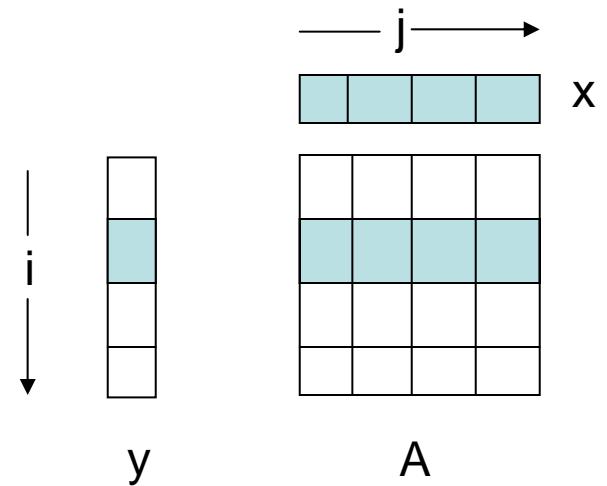
- A:  $N^2/b$  misses
    - x:  $N/b$  misses
    - y:  $N/b$  misses
    - Total =  $(N^2+2N)/b$
    - Miss ratio =  $(N^2+2N)/4bN^2$   
 $\sim 0.25/b + 0.5/bN$



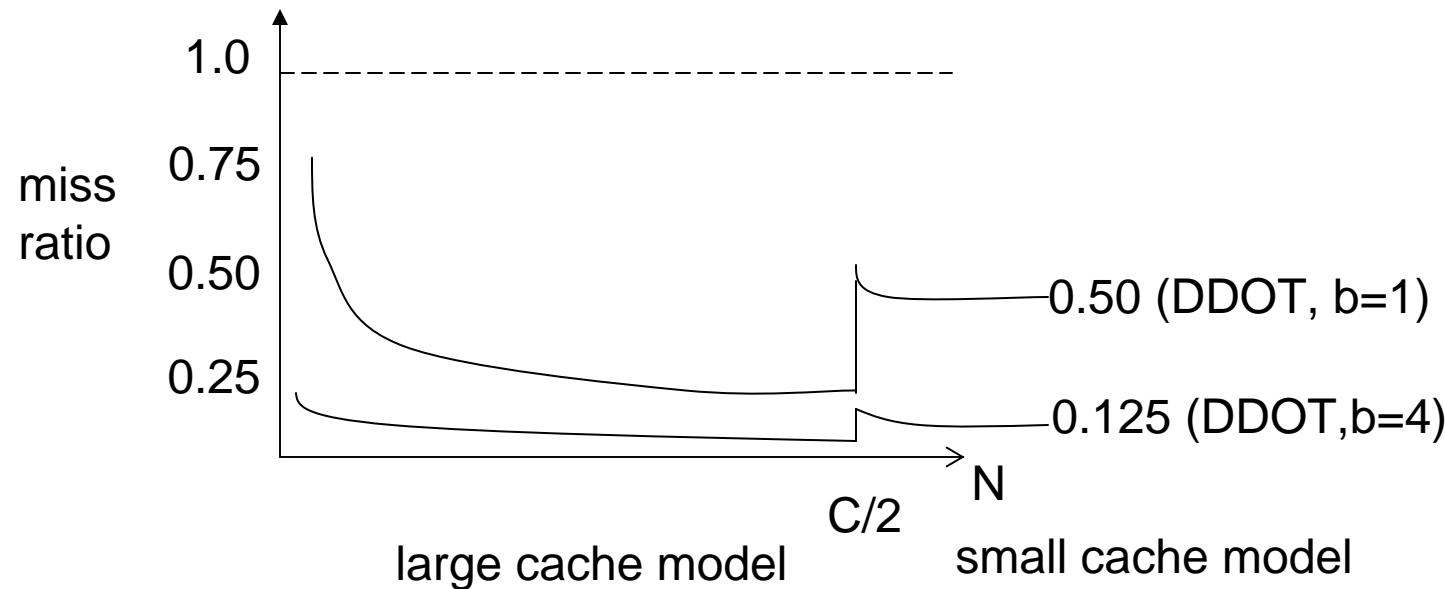
# Scenario III (contd.)

Address stream:  $y(1) \ A(1,1) \ x(1) \ y(1)$   $y(1) \ A(1,2) \ x(2) \ y(1)$  ...  $y(1) \ A(1,N) \ x(N) \ y(1)$   $y(2) \ A(2,1) \ x(1) \ y(2)$

- Small cache model:
  - A:  $N^2/b$  misses
  - x:  $N/b + N(N-1)/b$  misses (reuse distance=O(N))
  - y:  $N/b$  misses (reuse distance=O(1))
  - Total =  $(2N^2+N)/b$
  - Miss ratio =  $(2N^2+N)/4bN^2$   
 $\sim 0.5/b + 0.25/bN$
- Transition from large cache model to small cache model
  - As problem size increases, when do capacity misses begin to occur?
  - LRU: roughly when  $(2N+2b) = C$ 
    - $N \sim C/2$
  - Optimal: roughly when  $(N+2b) \sim C \rightarrow N \sim C$
- So miss ratio picture for Scenario III is similar to that of Scenario I but the y-axis is scaled down by b
- Typical value of b = 4 (SGI Octane)



# Miss ratio graph

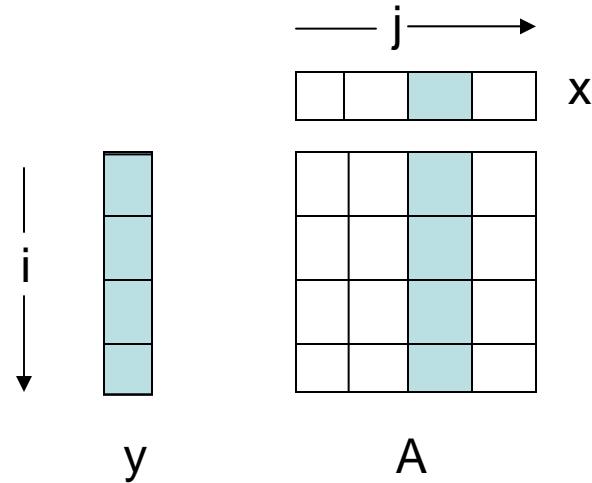


- Jump from large cache model to small cache model will be more gradual in reality because of conflict misses

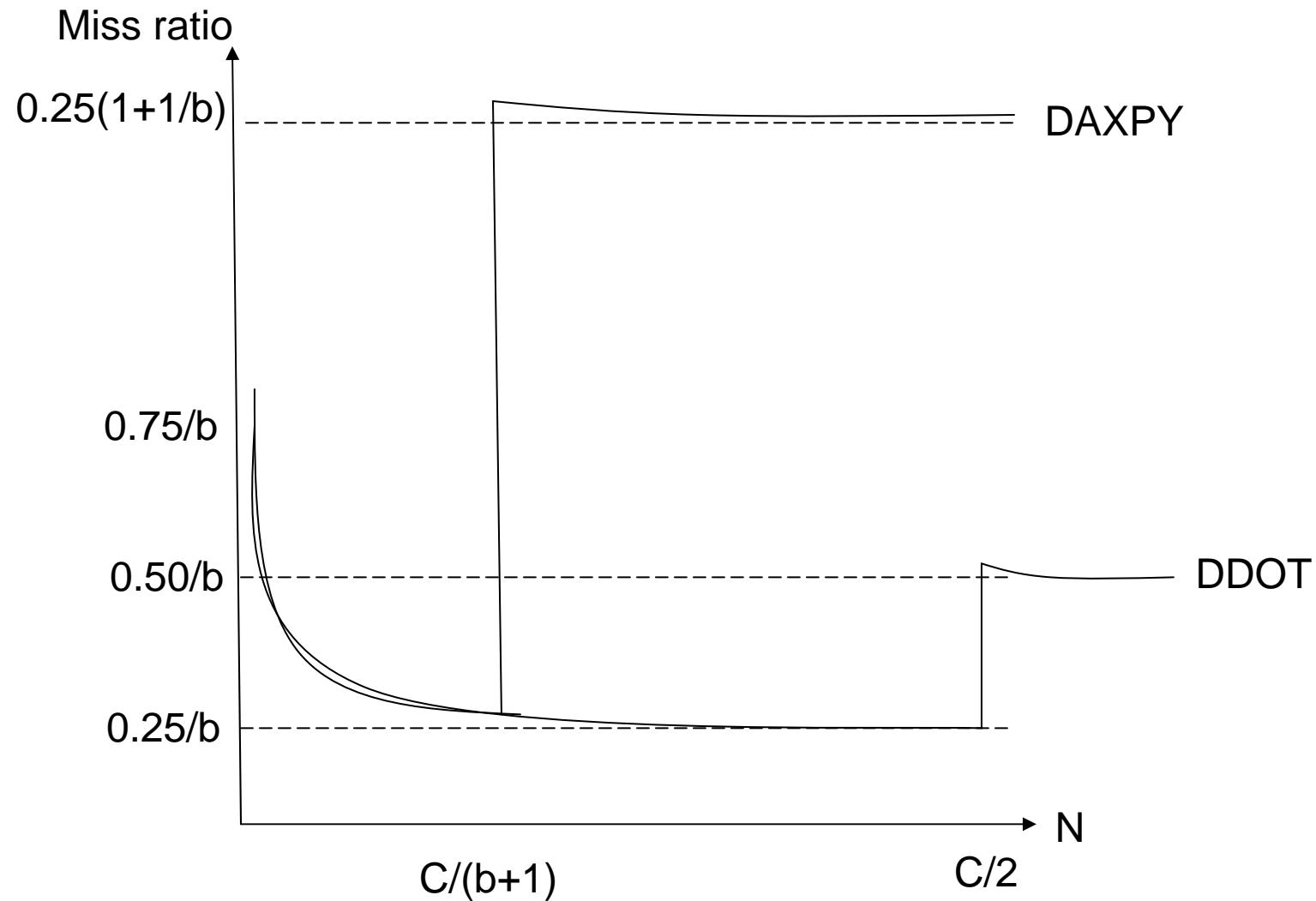
# Scenario IV

- Code:

```
for j = 1,N
    for i = 1,N
        y(i) = y(i) + A(i,j)*x(j)
```
- Large cache model:
  - Same as Scenario III
- Small cache model:
  - Misses:
    - A:  $N^2$
    - x:  $N/b$
    - y:  $N/b + N(N-1)/b = N^2/b$
    - Total:  $N^2(1+1/b) + N/b$
    - Miss ratio =  $0.25(1+1/b) + 0.25/bN$
- Transition from large cache to small cache model
  - LRU:  $Nb + N + b = C \rightarrow N \sim C/(b+1)$
  - optimal:  $N + 2b \sim C \rightarrow N \sim C$
- Transition happens much sooner than in Scenario III (with LRU replacement)



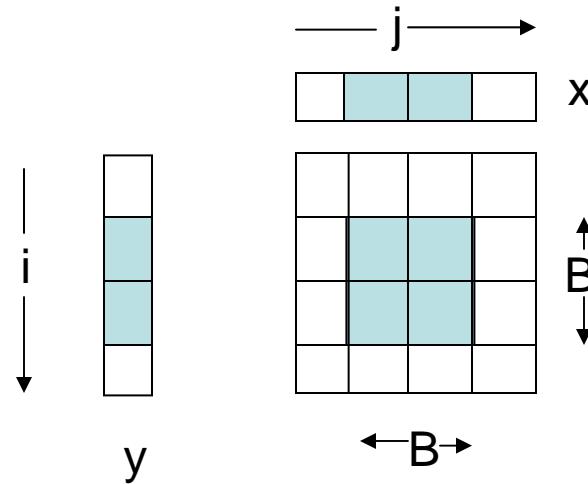
# Miss ratios



# Scenario V

- Intuition: perform blocked MVM so that data for each blocked MVM fits in cache
  - One estimate for B: all data for block MVM must fit in cache
    - ➔  $B^2 + 2B \sim C$
    - ➔  $B \sim \sqrt{C}$
  - Actually we can do better than this
- Code: blocked code

```
for bi = 1,N,B
    for bj = 1,N,B
        for i = bi,min(bi+B-1,N)
            for j = bj,min(bj+B-1,N)
                y(i)=y(i)+A(i,j)*x(j)
```
- Choose block size B so
  - you have large cache model while executing block
  - B is as large as possible (to reduce loop overhead)
  - for our example, this means  $B \sim c/2$  for row-major order of storage and LRU replacement
- Since entire MVM computation is a sequence of block MVMs, this means miss ratio will be  $0.25/b$  independent of N!



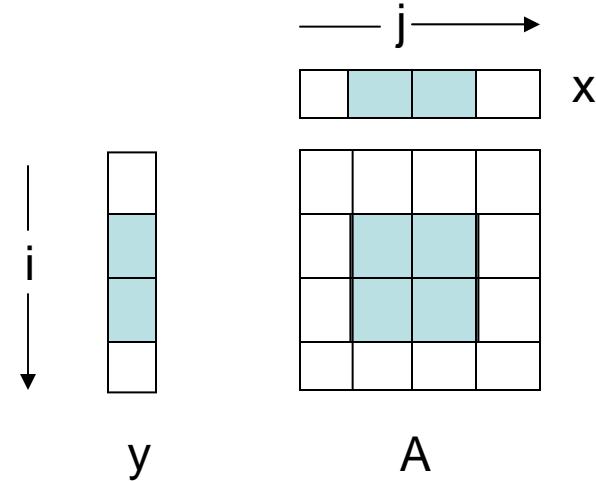
# Scenario V (contd.)

- Code: blocked code

```
for bi = 1,N,B  
  for bj = 1,N,B  
    for i = bi,min(bi+B-1,N)  
      for j = bj,min(bj+B-1,N)  
        y(i)=y(i)+A(i,j)*x(j)
```

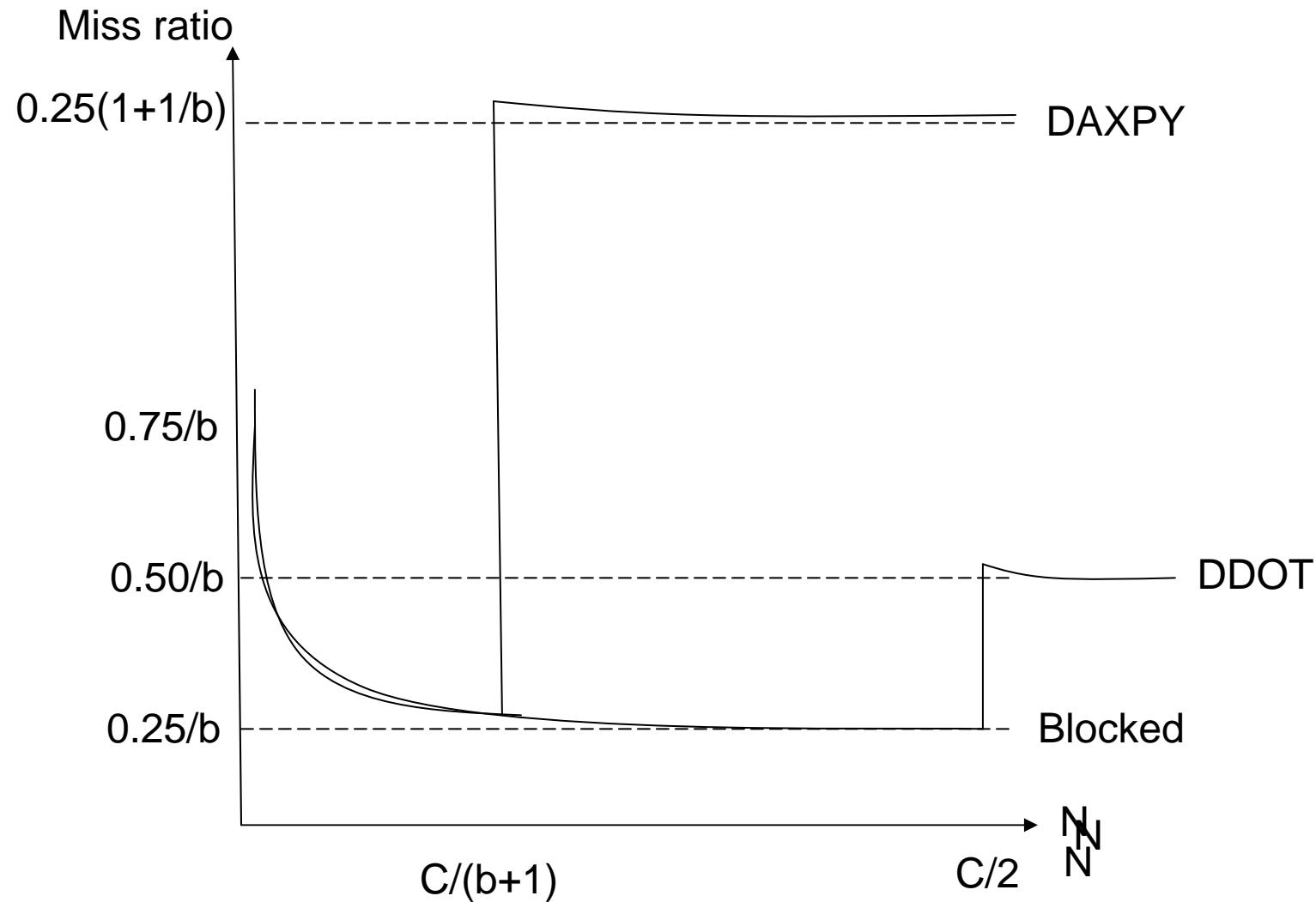
Better code: interchange the two outermost loops and fuse bi and i loops

```
for bj = 1,N,B  
  for i = 1,N  
    for j = bi,min(bi+B-1,N)  
      y(i)=y(i)+A(I,j)*x(j)
```



This has the same memory behavior as doubly-blocked loop but less loop overhead.

# Miss ratios



# Key transformations

- Loop permutation

for i = 1,N	for j = 1,N
for j = 1,N	for i = 1,N
S	S



- Strip-mining

for i = 1,N	for bi = 1,N,B
S	for i = bi, min(bi+B-1,N)
	S



- Loop tiling = strip-mine and interchange

for i = 1,N	for bi = 1,N,B
for j = 1,N	for j = 1,N
S	for i = bj,min(bj+B-1,N)
	S

# Notes

- Strip-mining does not change the order in which loop body instances are executed
  - so it is always legal
- Loop permutation and tiling do change the order in which loop body instances are executed
  - so they are not always legal
- For MVM and MMM, they are legal, so there are many variations of these kernels that can be generated by using these transformations
  - different versions have different memory behavior as we have seen

# Matrix multiplication

- We have studied MVM in detail.
- In dense linear algebra, matrix-matrix multiplication is more important.
- Everything we have learnt about MVM carries over to MMM fortunately, but there are more variations to consider since there are three matrices and three loops.

# MMM

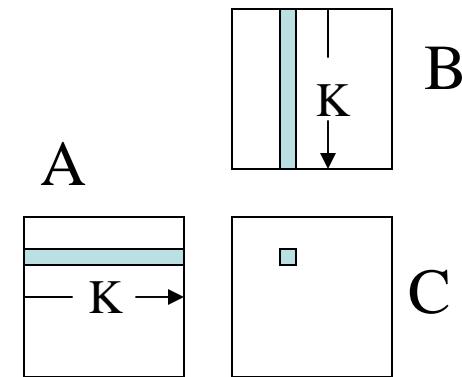
DO I = 1, N//row-major storage

DO J = 1, N

DO K = 1, N

$C(I,J) = C(I,J) + A(I,K)*B(K,J)$

IJK version of matrix multiplication



- Three loops: I,J,K
- You can show that all six permutations of these three loops compute the same values.
- As in MVM, the cache behavior of the six versions is different

# MMM

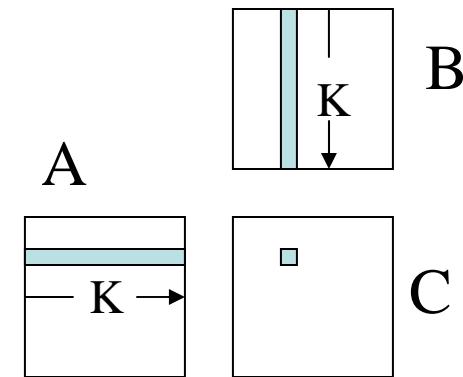
DO I = 1, N//row-major storage

  DO J = 1, N

    DO K = 1, N

      C(I,J) = C(I,J) + A(I,K)\*B(K,J)

IJK version of matrix multiplication

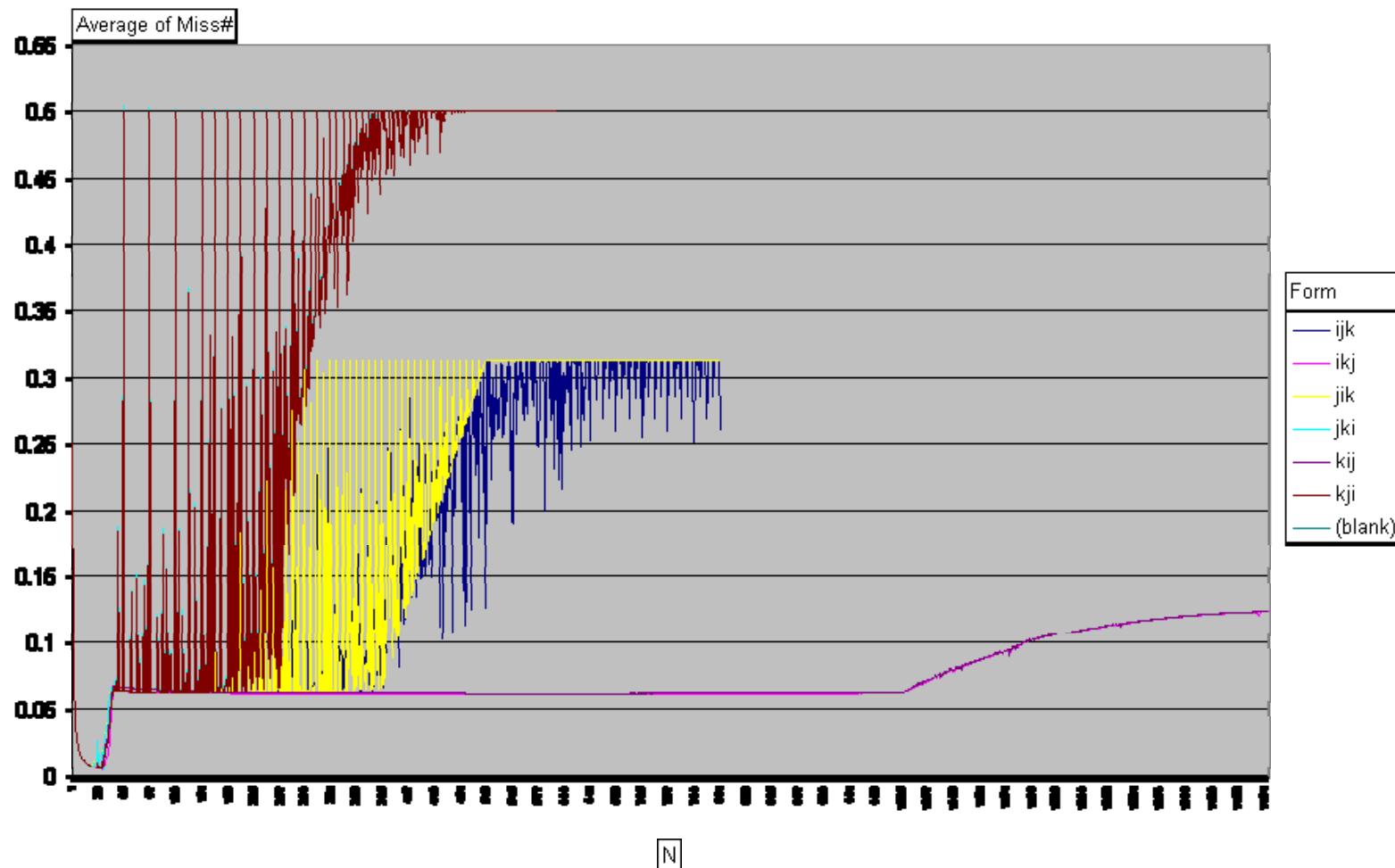


- **K loop innermost**
  - A: good spatial locality
  - C: good temporal locality
- **I loop innermost**
  - B: good temporal locality
- **J loop innermost**
  - B,C: good spatial locality
  - A: good temporal locality
- **So we would expect IKJ/KIJ versions to perform best, followed by IJK/JIK, followed by JKI/KJI**

# MMM miss ratios (simulated)

## L1 Cache Miss Ratio for Intel Pentium III

- MMM with  $N = 1 \dots 1300$
- 16KB 32B/Block 4-way 8-byte elements



# Observations

- Miss ratios depend on which loop is in innermost position
  - so there are three distinct miss ratio graphs
- Large cache behavior can be seen very clearly and all six version perform similarly in that region
- Big spikes are due to conflict misses for particular matrix sizes
  - notice that versions with J loop innermost have few conflict misses (why?)

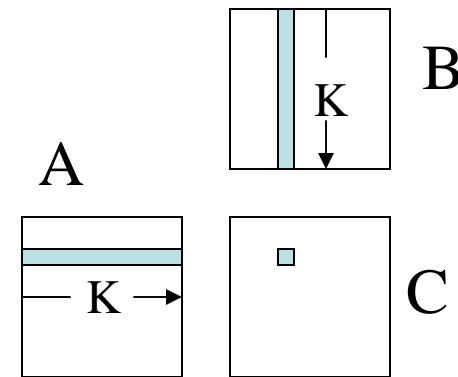
# IJK version

DO I = 1, N//row-major storage

DO J = 1, N

DO K = 1, N

$C(I,J) = C(I,J) + A(I,K)*B(K,J)$



- Large cache scenario:
  - Matrices are small enough to fit into cache
  - Only cold misses, no capacity misses
  - Miss ratio:
    - Data size =  $3 N^2$
    - Each miss brings in b floating-point numbers
    - Miss ratio =  $3 N^2 / b * 4N^3 = 0.75/bN$  (eg) 0.019 ( $b = 4, N=10$ )

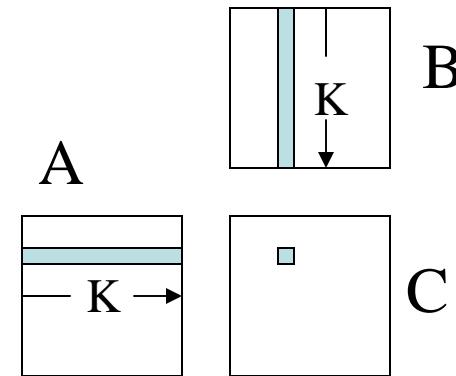
# IJK version (large cache)

DO I = 1, N//row-major storage

DO J = 1, N

DO K = 1, N

$$C(I,J) = C(I,J) + A(I,K)*B(K,J)$$

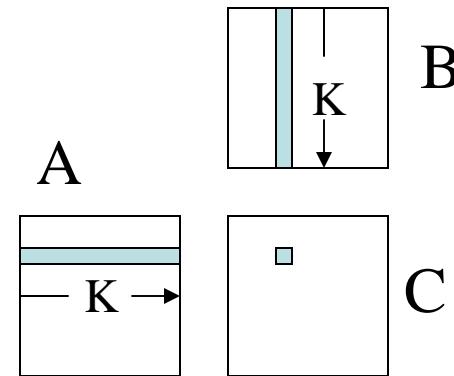


- Large cache scenario:
  - Matrices are small enough to fit into cache
  - Only cold misses, no capacity misses
  - Miss ratio:
    - Data size =  $3 N^2$
    - Each miss brings in b floating-point numbers
    - Miss ratio =  $3 N^2 / b * 4N^3 = 0.75/bN = 0.019$  ( $b = 4, N=10$ )

# IJK version (small cache)

```
DO I = 1, N  
  DO J = 1, N  
    DO K = 1, N  
      C(I,J) = C(I,J) + A(I,K)*B(K,J)
```

- Small cache scenario:
  - Matrices are large compared to cache
    - stack distance is not  $O(1) \Rightarrow$  miss
  - Cold and capacity misses
  - Miss ratio:
    - C:  $N^2/b$  misses (good temporal locality)
    - A:  $N^3/b$  misses (good spatial locality)
    - B:  $N^3$  misses (poor temporal and spatial locality)
    - Miss ratio  $\rightarrow 0.25(b+1)/b = 0.3125$  (for  $b = 4$ )



# Miss ratios for other versions

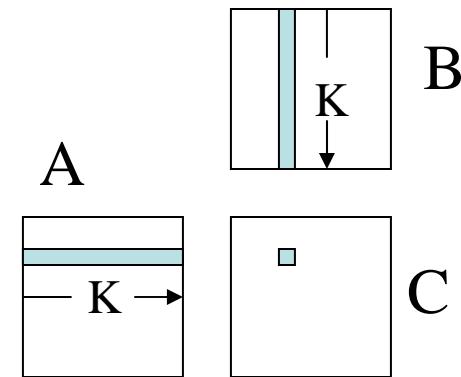
DO I = 1, N//row-major storage

DO J = 1, N

DO K = 1, N

$$C(I,J) = C(I,J) + A(I,K)*B(K,J)$$

IJK version of matrix multiplication



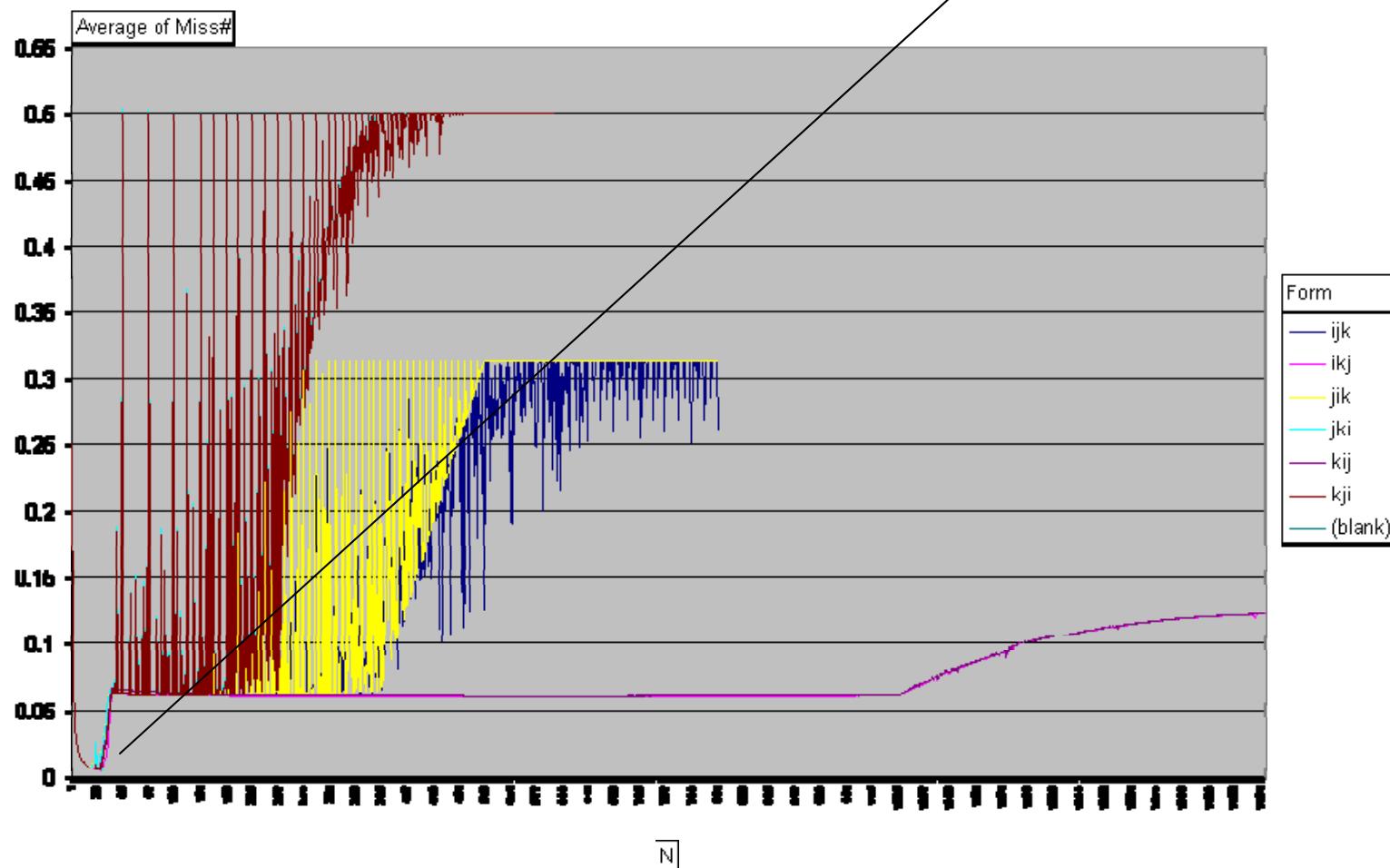
- **K loop innermost**
  - A: good spatial locality
  - C: good temporal locality
$$0.25(b+1)/b$$
- **I loop innermost**
  - B: good temporal locality
$$(N^2/b + N^3 + N^3)/4N^3 \rightarrow 0.5$$
- **J loop innermost**
  - B,C: good spatial locality
  - A: good temporal locality
$$(N^3/b + N^3/b + N^2/b)/4N^3 \rightarrow 0.5/b$$
- So we would expect IKJ/KIJ versions to perform best, followed by IJK/JIK, followed by JKI/KJI

# MMM experiments

L1 Cache Miss Ratio for Intel Pentium III

- MMM with  $N = 1 \dots 1300$
- 16KB 32B/Block 4-way 8-byte elements

Can we predict this?



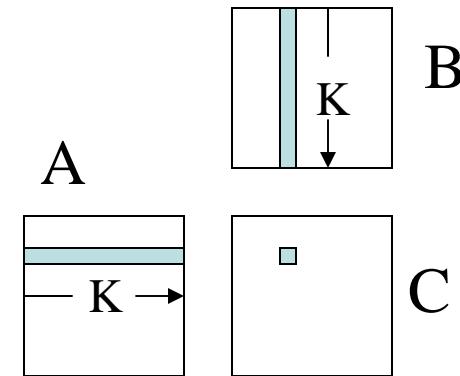
# Transition out of large cache

DO I = 1, N//row-major storage

DO J = 1, N

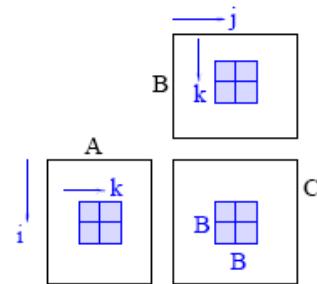
DO K = 1, N

$C(I,J) = C(I,J) + A(I,K)*B(K,J)$



- Find the data element(s) that are reused with the largest stack distance
- Determine the condition on N for that to be less than C
- For our problem:
  - $N^2 + N + b < C$  (with optimal replacement)
  - $N^2 + 2N < C$  (with LRU replacement)
  - In either case, we get  $N \sim \sqrt{C}$
  - For our cache, we get  $N \sim 45$  which agrees quite well with data

# Blocked code



```
for bi = 1,N,B  
  for bj = 1,N,B  
    for bk = 1,N,B  
      for i = bi, min(bi+B-1,N)  
        for j = bj, min(bj+B-1,N)  
          for k = bk, min(bk+B-1,N)  
            y(i) = y(i) + A(i,j)*x(j)
```

As in blocked MVM, we actually need to stripmine only two loops

# Notes

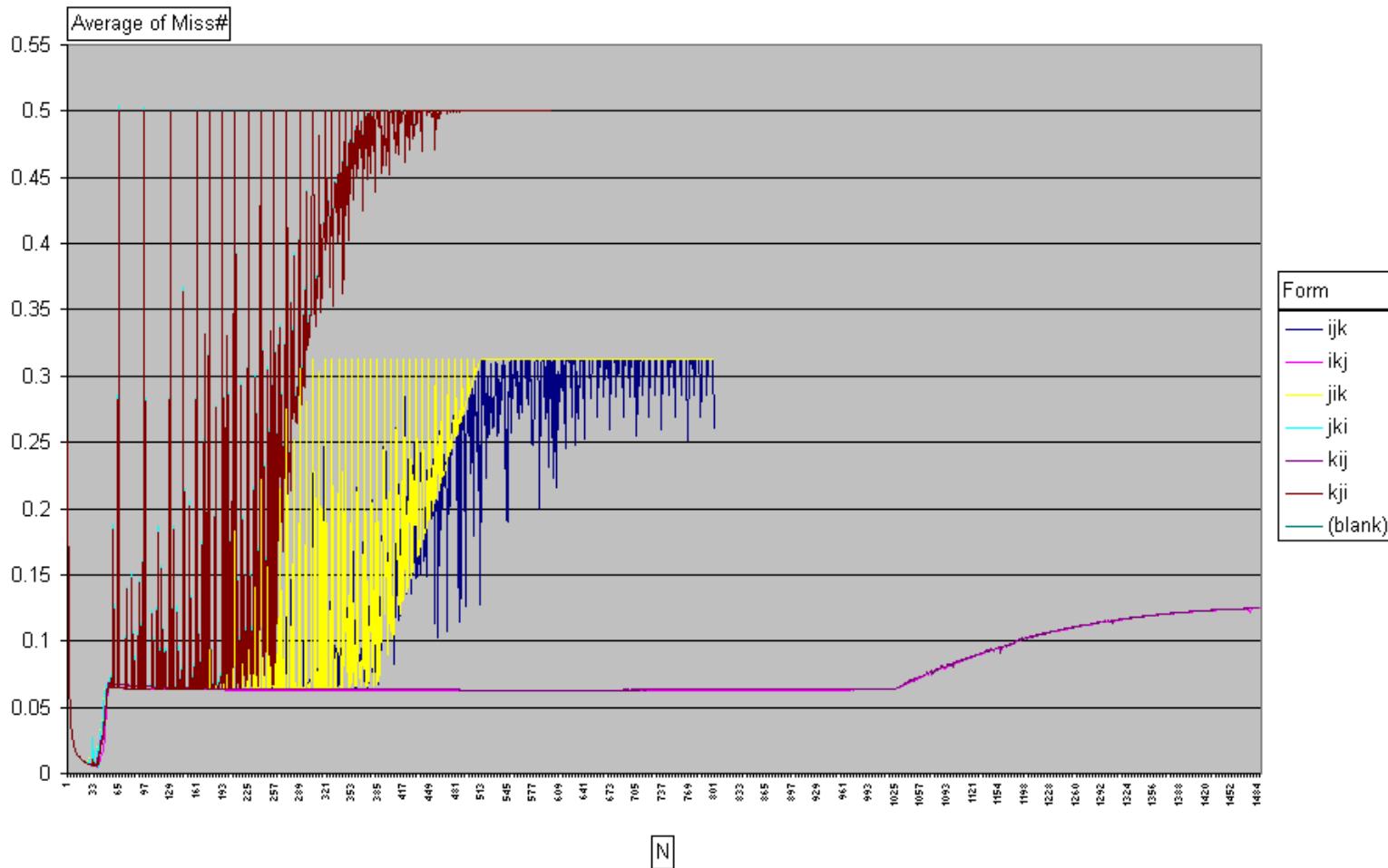
- So far, we have considered a two-level memory hierarchy
- Real machines have multiple level memory hierarchies
- In principle, we need to block for all levels of the memory hierarchy
- In practice, matrix multiplication with really large matrices is **very rare**
  - MMM shows up mainly in blocked matrix factorizations
  - therefore, it is enough to block for registers, and L1/L2 cache levels
- **How do we organize such a code?**
  - We will study the code produced by ATLAS.
  - ATLAS also introduces us to self-optimizing programs.

# Optimizing MMM & ATLAS Library Generator

# Recall: MMM miss ratios

## L1 Cache Miss Ratio for Intel Pentium III

- MMM with  $N = 1 \dots 1300$
- 16KB 32B/Block 4-way 8-byte elements



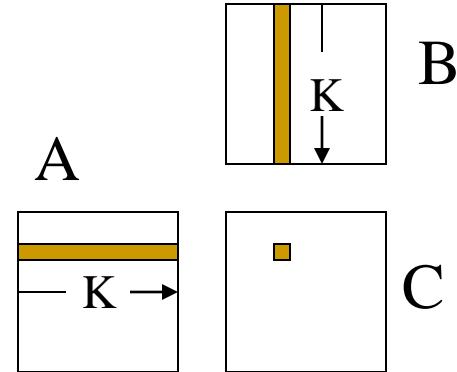
# IJK version (large cache)

DO I = 1, N//row-major storage

DO J = 1, N

DO K = 1, N

$$C(I,J) = C(I,J) + A(I,K) * B(K,J)$$

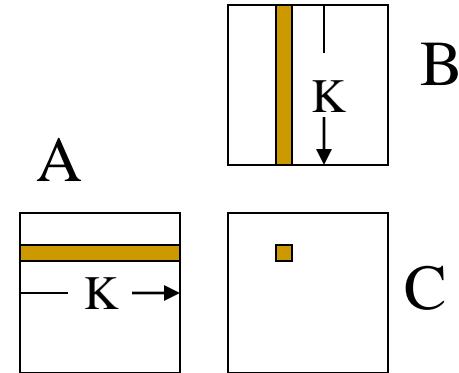


## ■ Large cache scenario:

- Matrices are small enough to fit into cache
- Only cold misses, no capacity misses
- Miss ratio:
  - Data size =  $3 N^2$
  - Each miss brings in b floating-point numbers
  - Miss ratio =  $3 N^2 / b * 4N^3 = 0.75/bN = 0.019$  ( $b = 4, N=10$ )

# IJK version (small cache)

```
DO I = 1, N  
  DO J = 1, N  
    DO K = 1, N  
      C(I,J) = C(I,J) + A(I,K)*B(K,J)
```



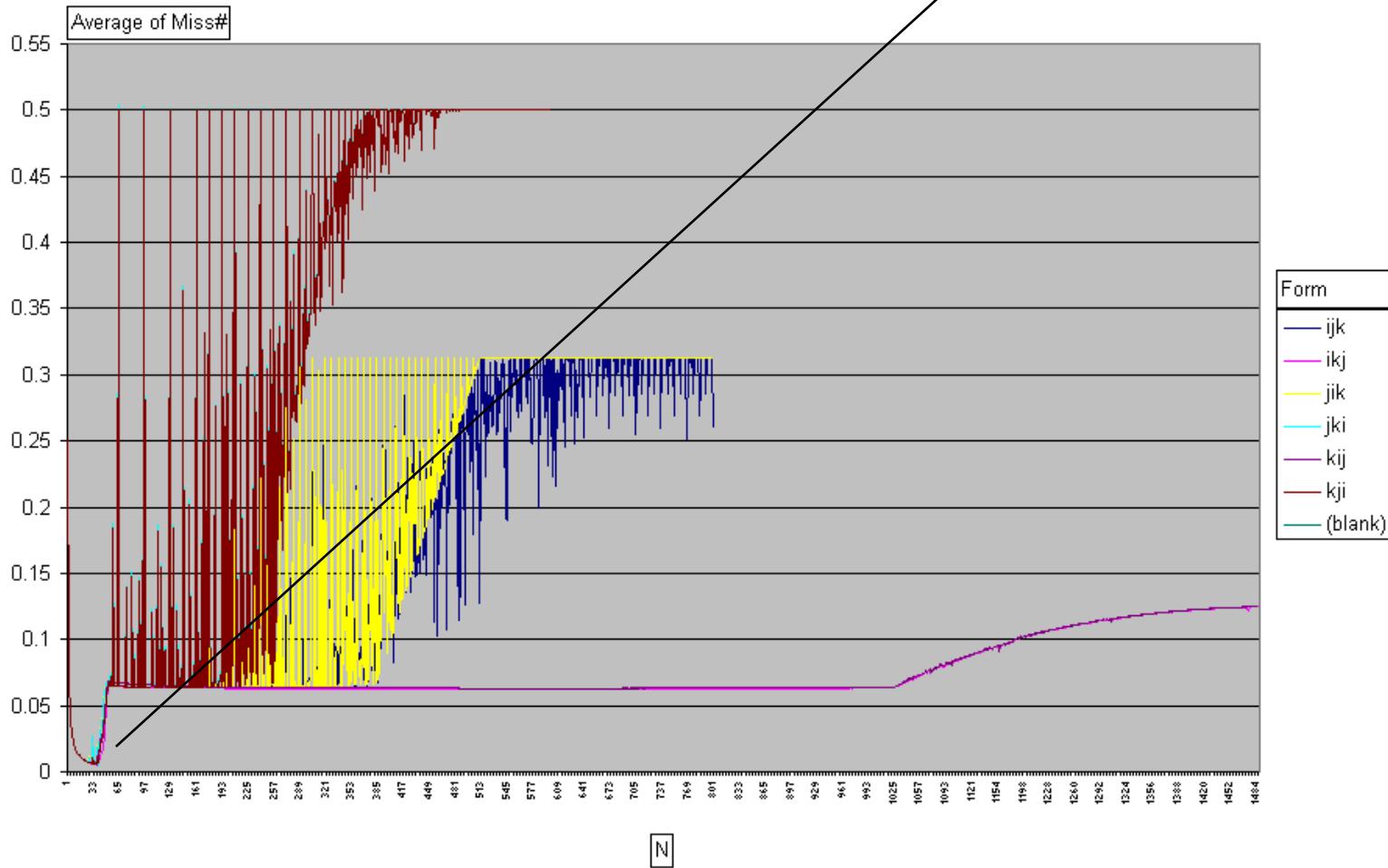
- Small cache scenario:
  - Matrices are large compared to cache
    - reuse distance is not  $O(1)$  => miss
  - Cold and capacity misses
  - Miss ratio:
    - C:  $N^2/b$  misses (good temporal locality)
    - A:  $N^3/b$  misses (good spatial locality)
    - B:  $N^3$  misses (poor temporal and spatial locality)
    - Miss ratio  $\rightarrow 0.25(b+1)/b = 0.3125$  (for  $b = 4$ )

# MMM experiments

L1 Cache Miss Ratio for Intel Pentium III

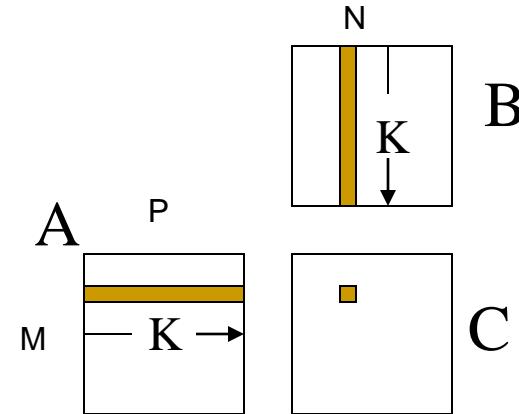
- MMM with  $N = 1 \dots 1300$
- 16KB 32B/Block 4-way 8-byte elements

Can we predict this?



# How large can matrices be and still not suffer capacity misses?

```
DO I = 1, M  
DO J = 1, N  
  DO K = 1, P  
    C(I,J) = C(I,J) + A(I,K)*B(K,J)
```



- How large can these matrices be without suffering capacity misses?
  - Each iteration of outermost loop walks over entire B matrix, so all of B must be in cache
  - We walk over rows of A and successive iterations of middle loop touch same row of A, so one row of A must be in cache
  - We walk over elements of C one at a time.
  - So inequality is  $NP + P + 1 \leq C$

# Check with experiment

- For our machine, capacity of L1 cache is  
 $16\text{KB}/8 \text{ doubles} = 2^{11} \text{ doubles}$
- If matrices are square, we must solve
$$N^2 + N + 1 = 2^{11}$$
which gives us  $N = 45$
- This agrees well with experiment.

# High-level picture of high-performance MMM code

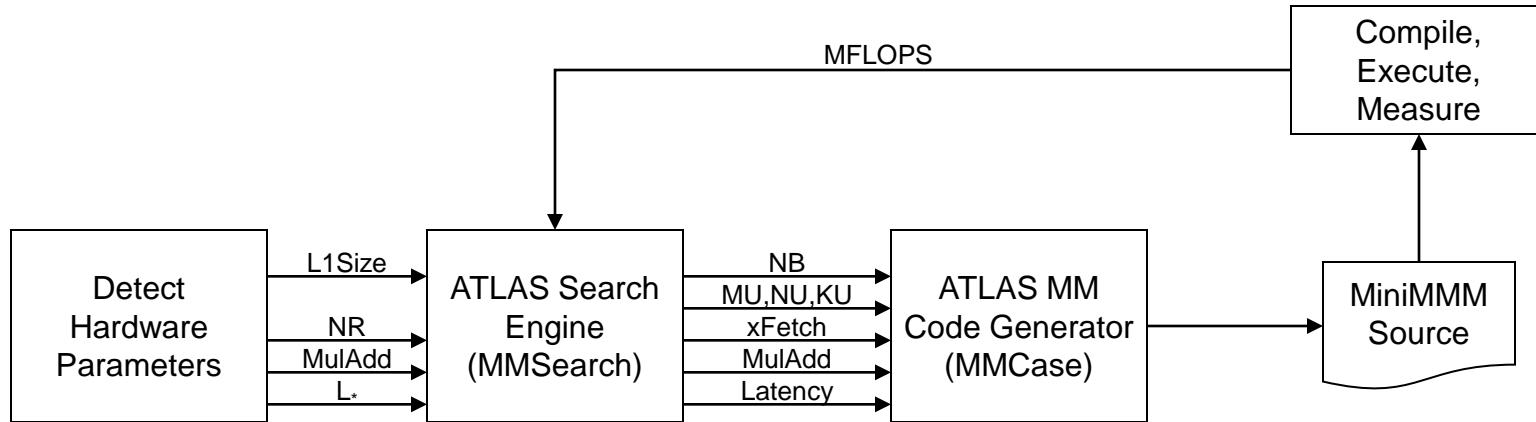
- Block the code for each level of memory hierarchy
  - Registers
  - L1 cache
  - .....
- Choose block sizes at each level using the theory described previously
  - Useful optimization: choose block size at level  $L+1$  to be multiple of the block size at level  $L$

# ATLAS

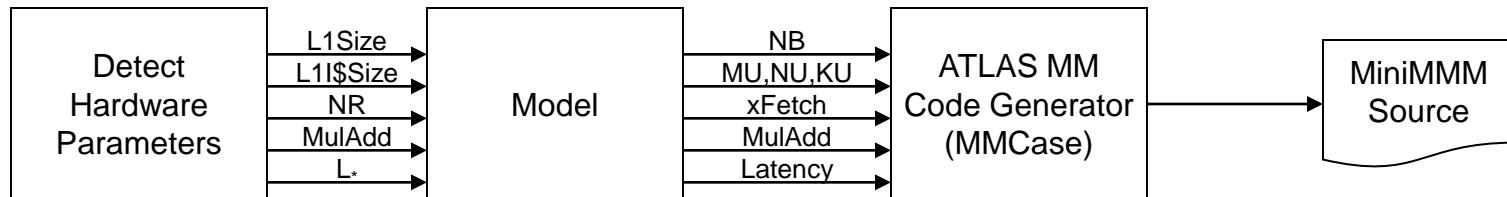
- Library generator for MMM and other BLAS
- Blocks only for registers and L1 cache
- Uses search to determine block sizes, rather than the analytical formulas we used
  - Search takes more time, but we do it once when library is produced
- Let us study structure of ATLAS in little more detail

# Our approach

## ■ Original ATLAS Infrastructure



## ■ Model-Based ATLAS Infrastructure



# BLAS

## ■ Let us focus on MMM:

```
for (int i = 0; i < M; i++)  
    for (int j = 0; j < N; j++)  
        for (int k = 0; k < K; k++)  
            C[i][j] += A[i][k]*B[k][j]
```

## ■ Properties

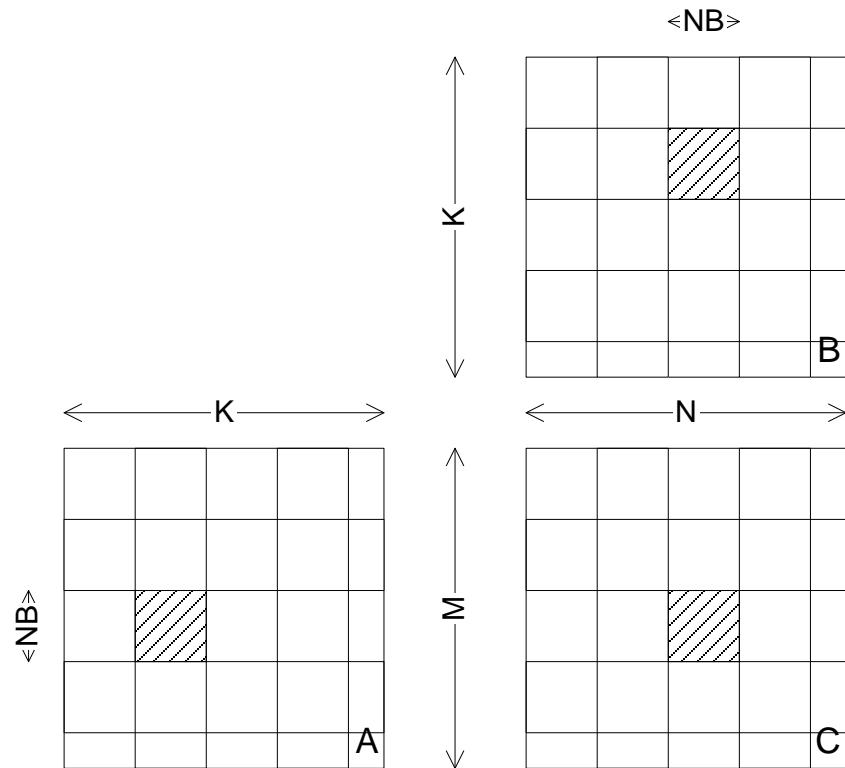
- Very good reuse:  $O(N^2)$  data,  $O(N^3)$  computation
- Many optimization opportunities
  - Few “real” dependencies
- Will run poorly on modern machines
  - Poor use of cache and registers
  - Poor use of processor pipelines

# Optimizations

- Cache-level blocking (tiling)
  - Atlas blocks only for L1 cache
  - NB: L1 cache time size
- Register-level blocking
  - Important to hold array values in registers
  - MU,NU: register tile size
- Filling the processor pipeline
  - Unroll and schedule operations
  - Latency, xFetch: scheduling parameters
- Versioning
  - Dynamically decide which way to compute
- Back-end compiler optimizations
  - Scalar Optimizations
  - Instruction Scheduling

# Cache-level blocking (tiling)

- Tiling in ATLAS
    - Only square tiles ( $NB \times NB \times NB$ )
    - Working set of tile fits in L1
    - Tiles are usually copied to continuous storage
    - Special “clean-up” code generated for boundaries
  - Mini-MMM
  - **NB:** Optimization parameter
- ```
for (int j = 0; j < NB; j++)
    for (int i = 0; i < NB; i++)
        for (int k = 0; k < NB; k++)
            C[i][j] += A[i][k] * B[k][j]
```



# Register-level blocking

## ■ Micro-MMM

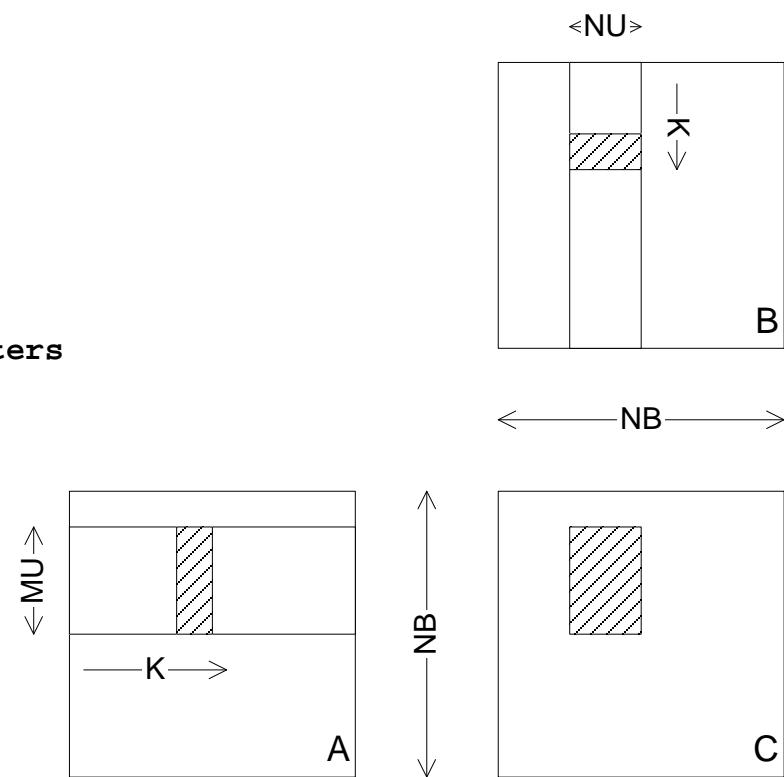
- A: MUx1
- B: 1xNU
- C: MUxNU
- MUxNU+MU+NU registers

## ■ Unroll loops by MU, NU, and KU

## ■ Mini-MMM with Micro-MMM inside

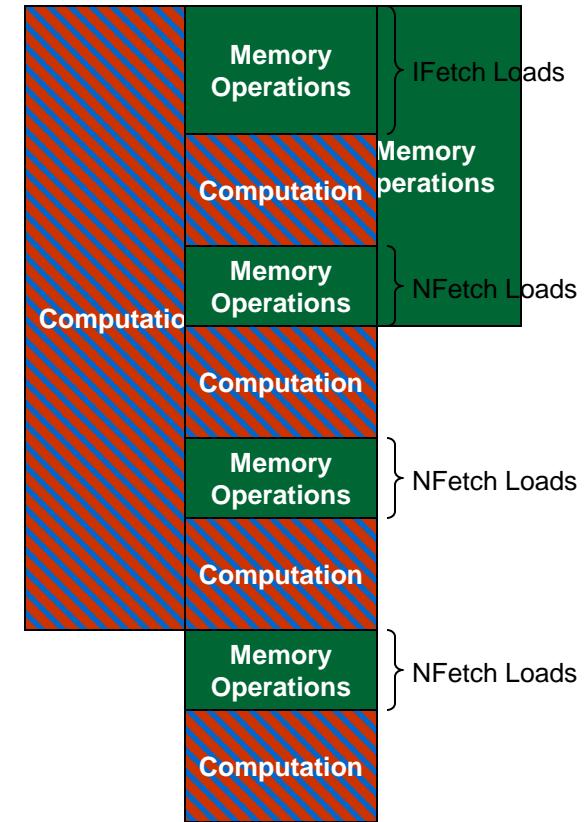
```
for (int j = 0; j < NB; j += NU)
    for (int i = 0; i < NB; i += MU)
        load C[i..i+MU-1, j..j+NU-1] into registers
        for (int k = 0; k < NB; k++)
            KU times { load A[i..i+MU-1,k] into registers
                        load B[k,j..j+NU-1] into registers
                        multiply A's and B's and add to C's
                        store C[i..i+MU-1, j..j+NU-1]
```

- Special clean-up code required if NB is not a multiple of MU,NU,KU
- **MU, NU, KU:** optimization parameters



# Scheduling

- FMA Present?
- Schedule Computation
  - Using **Latency**
- Schedule Memory Operations
  - Using **IFetch, NFetch,FFetch**
- **Latency, xFetch:** optimization parameters



# Search Strategy

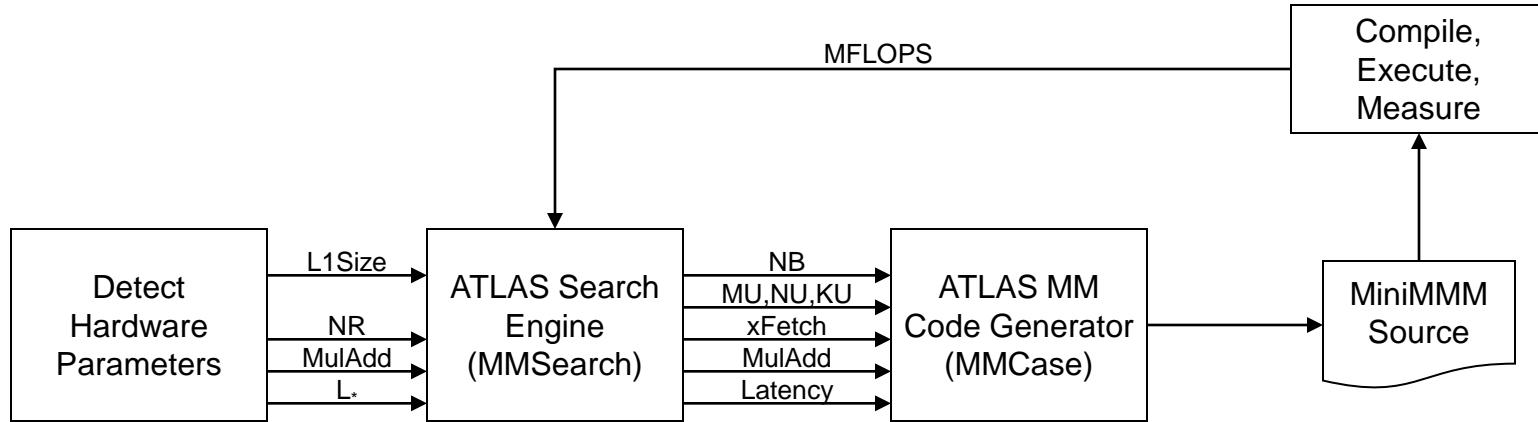
- Multi-dimensional optimization problem:
  - Independent parameters: NB,MU,NU,KU,...
  - Dependent variable: MFlops
  - Function from parameters to variables is given implicitly; can be evaluated repeatedly
- One optimization strategy: orthogonal line search
  - Optimize along one dimension at a time, using reference values for parameters not yet optimized
  - Not guaranteed to find optimal point, but might come close

# Find Best NB

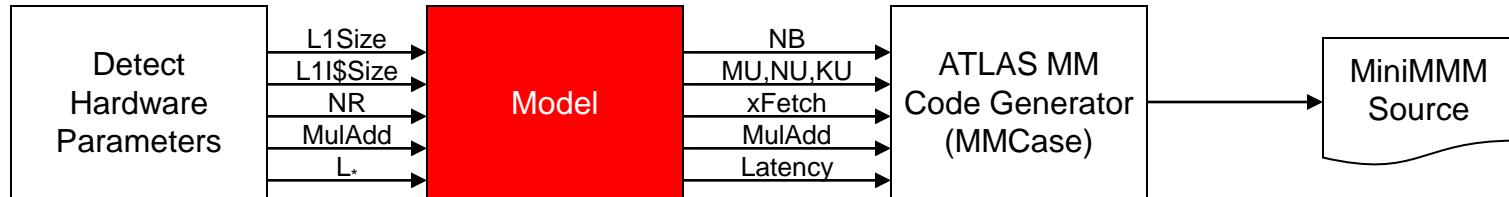
- Search in following range
  - $16 \leq NB \leq 80$
  - $NB^2 \leq L1Size$
- In this search, use simple estimates for other parameters
  - (eg) KU: Test each candidate for
    - Full K unrolling ( $KU = NB$ )
    - No K unrolling ( $KU = 1$ )

# Model-based optimization

## ■ Original ATLAS Infrastructure



## ■ Model-Based ATLAS Infrastructure



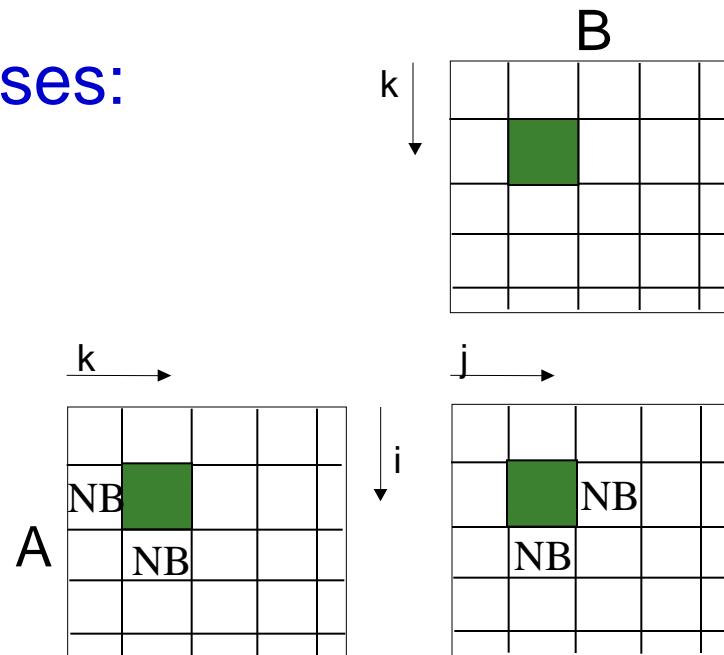
# Modeling for Optimization Parameters

## ■ Optimization parameters

- NB
  - Hierarchy of Models (later)
- MU, NU
  - $MU * NU + MU + NU + Latency \leq NR$
- KU
  - maximize subject to L1 Instruction Cache
- Latency
  - $\lceil (L_* + 1)/2 \rceil$
- MulAdd
  - hardware parameter
- xFetch
  - set to 2

# Largest NB for no capacity/conflict misses

- If tiles are copied into contiguous memory,  
condition for only cold misses:
    - $3 * NB^2 \leq L1Size$



# Largest NB for no capacity misses

- MMM:

```
for (int j = 0; i < N; i++)  
    for (int i = 0; j < N; j++)  
        for (int k = 0; k < N; k++)  
            c[i][j] += a[i][k] * b[k][j]
```

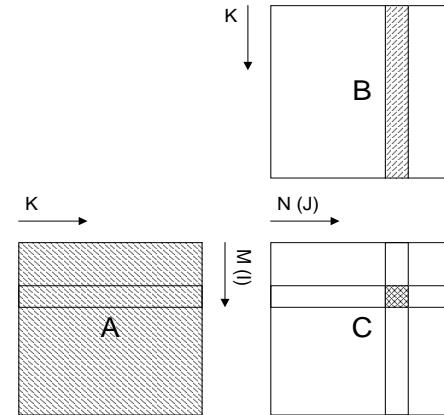
- Cache model:

- Fully associative
- Line size 1 Word
- Optimal Replacement

- Bottom line:

$$NB^2 + NB + 1 < C$$

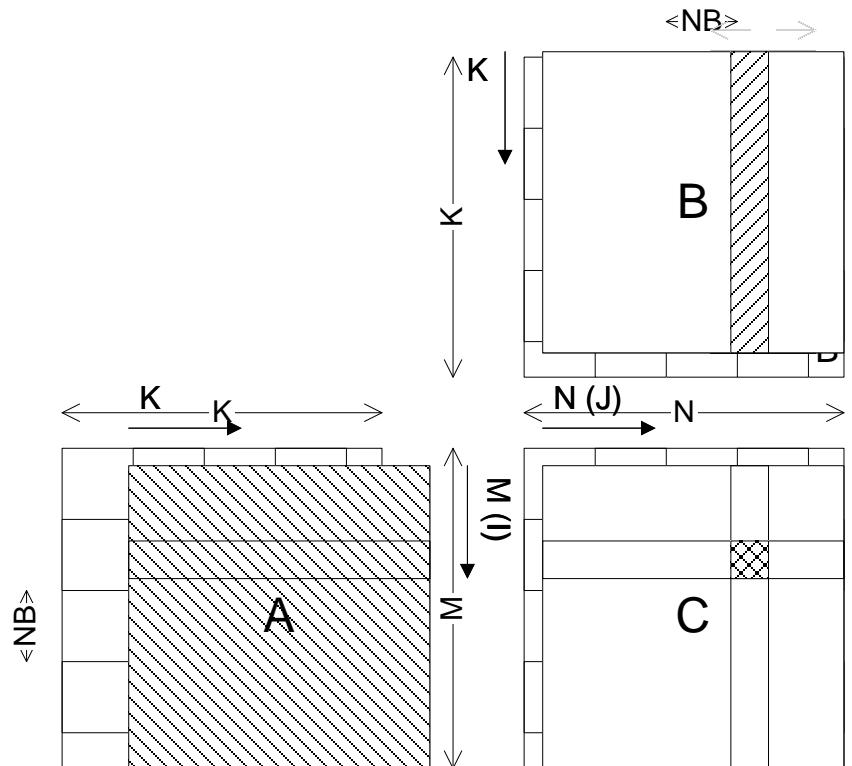
- One full matrix
- One row / column
- One element



# Summary: Modeling for Tile Size (NB)

- Models of increasing complexity

- $3 \cdot NB^2 \leq C$ 
  - Whole work-set fits in L1
- $NB^2 + NB + 1 \leq C$ 
  - Fully Associative
  - Optimal Replacement
  - Line Size: 1 word
- $\left\lceil \frac{NB^2}{B} \right\rceil + \left\lceil \frac{NB}{B} \right\rceil + 1 \leq \frac{C}{B}$  or  $\left\lceil \frac{NB^2}{B} \right\rceil + NB + 1 \leq \frac{C}{B}$ 
  - Line Size > 1 word
- $\left\lceil \frac{NB^2}{B} \right\rceil + 2 \left\lceil \frac{NB}{B} \right\rceil + \left( \left\lceil \frac{NB}{B} \right\rceil + 1 \right) \leq \frac{C}{B}$  or  
 $\left\lceil \frac{NB^2}{B} \right\rceil + 3NB + 1 \leq \frac{C}{B}$ 
  - LRU Replacement



# Summary of model

- **Estimating  $FMA$ :**

Use the machine parameter  $FMA$

- **Estimating  $L_s$ :**

$$L_s = \left\lceil \frac{L_* \times |ALU_{FP}| + 1}{2} \right\rceil$$

- **Estimating  $M_U$  and  $N_U$ :**

$$M_U \times N_U + N_U + M_U + L_s \leq N_R$$

- 1)  $M_U, N_U \leftarrow u$ .
- 2) Solve constraint for  $u$ .
- 3)  $M_U \leftarrow \max(u, 1)$ .
- 4) Solve constraint for  $N_U$ .
- 5)  $N_U \leftarrow \max(N_U, 1)$ .
- 6) If  $M_U < N_U$  then swap  $M_U$  and  $N_U$ .

- **Estimating  $N_B$ :**

$$\left\lceil \frac{N_B^2}{B_1} \right\rceil + 3 \left\lceil \frac{N_B \times N_U}{B_1} \right\rceil + \left\lceil \frac{M_U}{B_1} \right\rceil \times N_U \leq \frac{C_1}{B_1}$$

Trim  $N_B$ , to make it a multiple of  $M_U$ ,  $N_U$ , and 2.

- **Estimating  $K_U$ :**

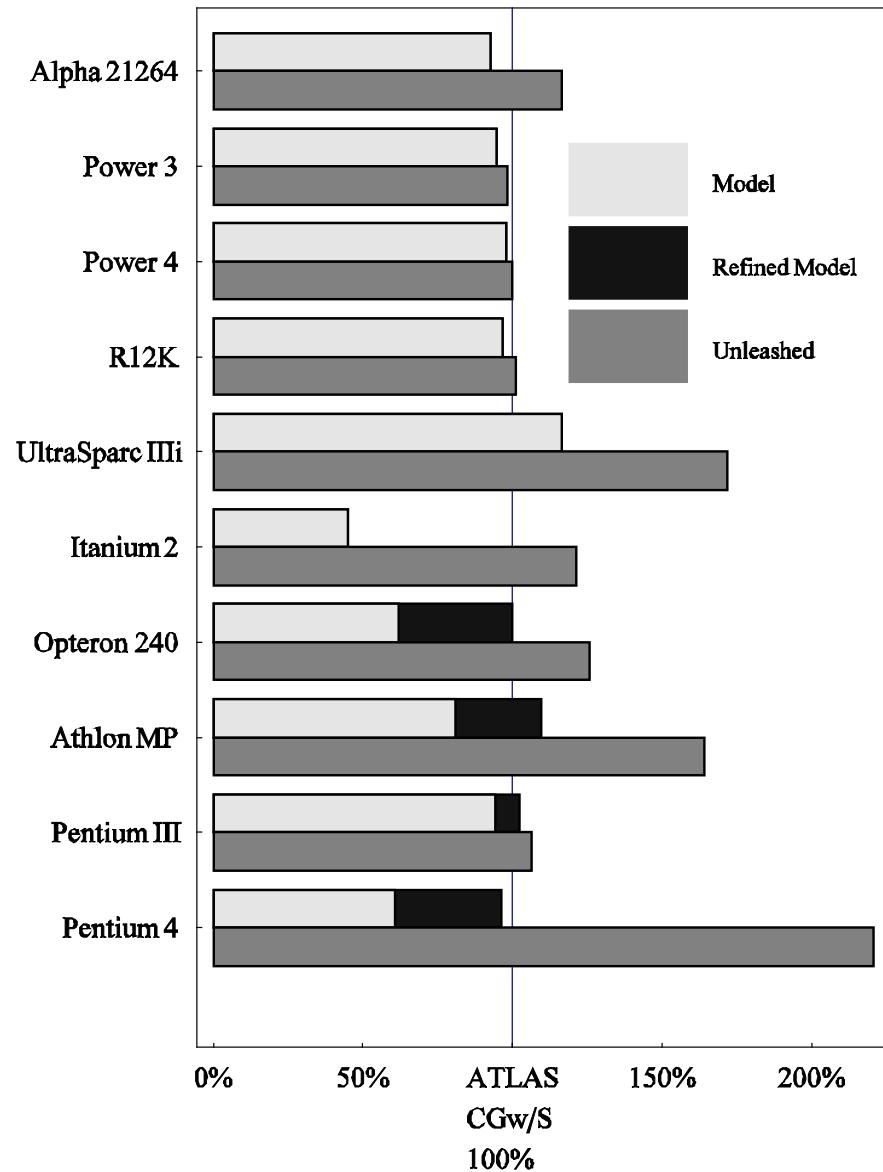
Choose  $K_U$  as the maximum value for which mini-MMM fits in the L1 instruction cache. Trim  $K_U$  to make it divide  $N_B$  evenly.

- **Estimating  $F_F$ ,  $I_F$ , and  $N_F$ :**

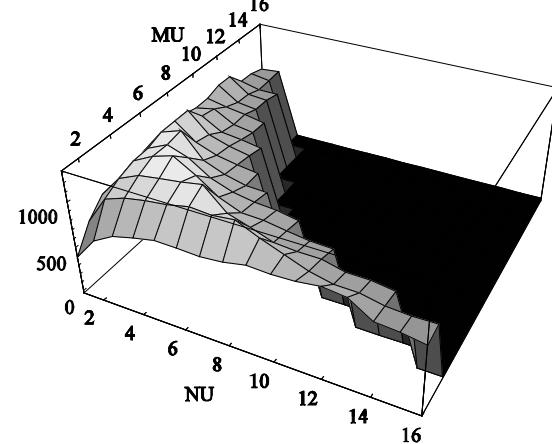
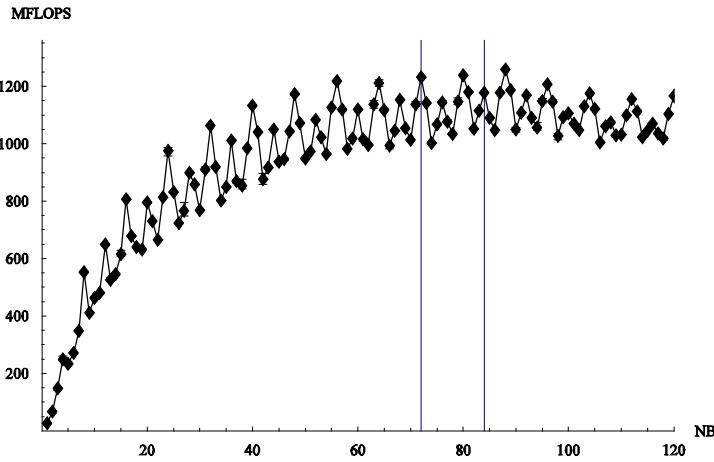
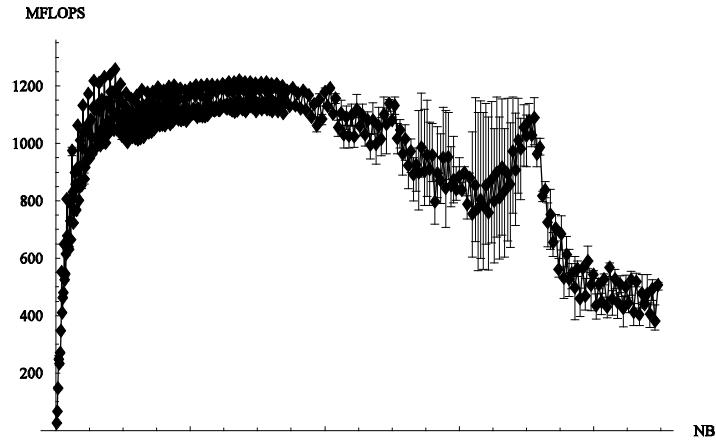
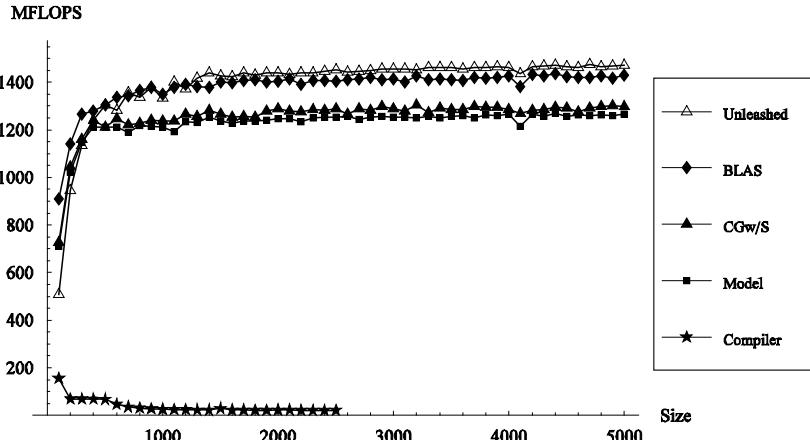
$$F_F = 0, I_F = 2, N_F = 2$$

# Experiments

- Ten modern architectures
- Model did well on
  - RISC architectures
  - UltraSparc: did better
- Model did not do as well on
  - Itanium
  - CISC architectures
- Substantial gap between ATLAS CGw/S and ATLAS Unleashed on some architectures



# Some sensitivity graphs for Alpha 21264



# Eliminating performance gaps

- Think globally, search locally
- Gap between model-based optimization and empirical optimization can be eliminated by
  - Local search:
    - for small performance gaps
    - in neighborhood of model-predicted values
  - Model refinement:
    - for large performance gaps
    - must be done manually
    - (future) machine learning: learn new models automatically
- Model-based optimization and empirical optimization are not in conflict

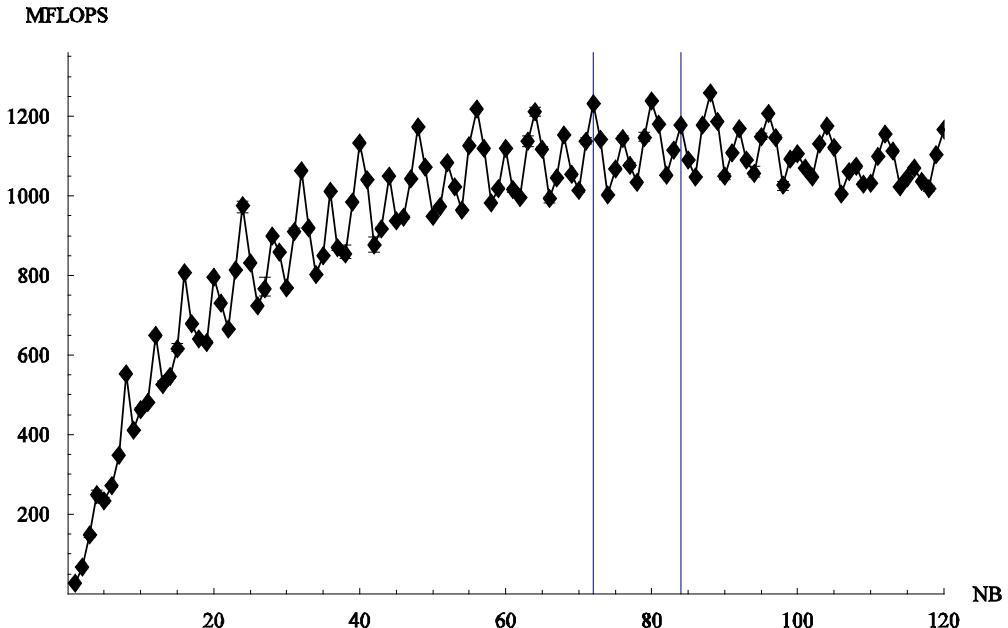
# Small performance gap: Alpha 21264

ATLAS CGw/S:

mini-MMM: 1300 MFlops  
NB = 72  
 $(MU,NU) = (4,4)$

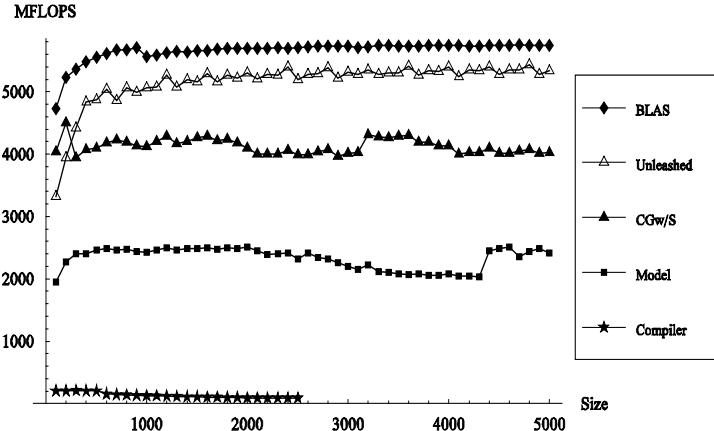
ATLAS Model

mini-MMM: 1200 MFlops  
NB = 84  
 $(MU,NU) = (4,4)$



- Local search
  - Around model-predicted NB
  - Hill-climbing not useful
  - Search interval:  $[NB - \text{lcm}(MU,NU), NB + \text{lcm}(MU,NU)]$
- Local search for MU,NU
  - Hill-climbing OK

# Large performance gap: Itanium



## MMM Performance

### Performance of mini-MMM

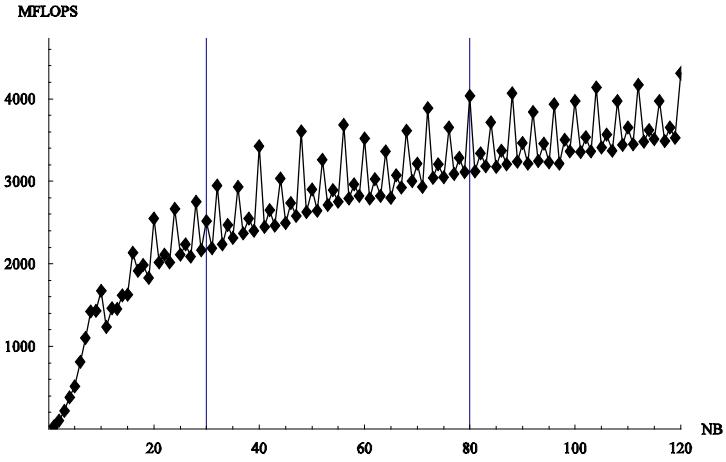
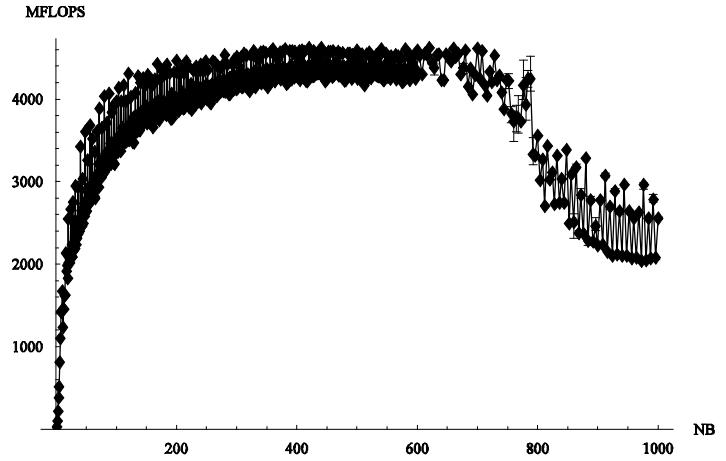
- ATLAS CGw/S: 4000 MFlops
- ATLAS Model: 1800 MFlops

### Problem with NB value

ATLAS Model: 30

ATLAS CGw/S: 80 (search space max)

Local search will not solve problem.

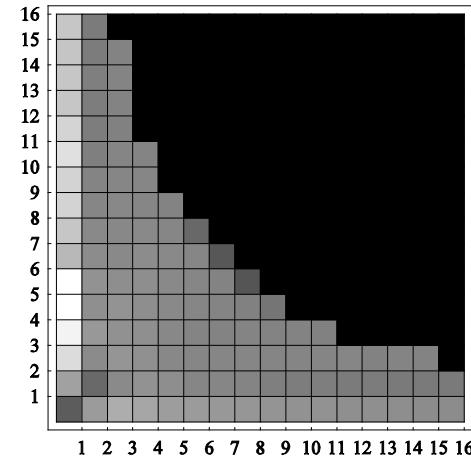
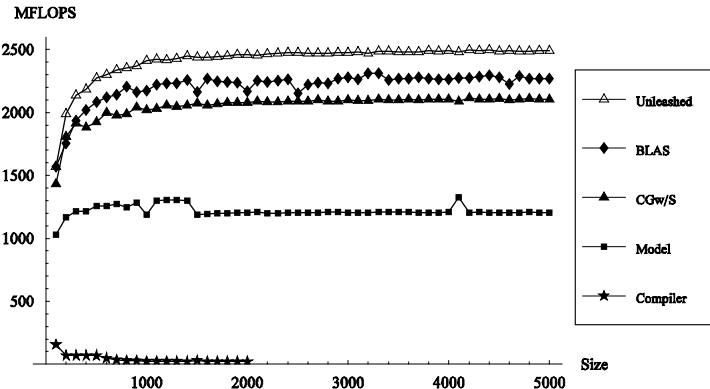


## NB Sensitivity

# Itanium diagnosis and solution

- Memory hierarchy
  - L1 data cache: 16 KB
  - L2 cache: 256 KB
  - L3 cache: 3 MB
- Diagnosis:
  - Model tiles for L1 cache
  - On Itanium, FP values not cached in L1 cache!
  - Performance gap goes away if we model for L2 cache (NB = 105)
  - Obtain even better performance if we model for L3 cache (NB = 360, 4.6 GFlops)
- Problem:
  - Tiling for L2 or L3 may be better than tiling for L1
  - How do we determine which cache level to tile for??
- Our solution: model refinement + a little search
  - Determine tile sizes for all cache levels
  - Choose between them empirically

# Large performance gap: Opteron



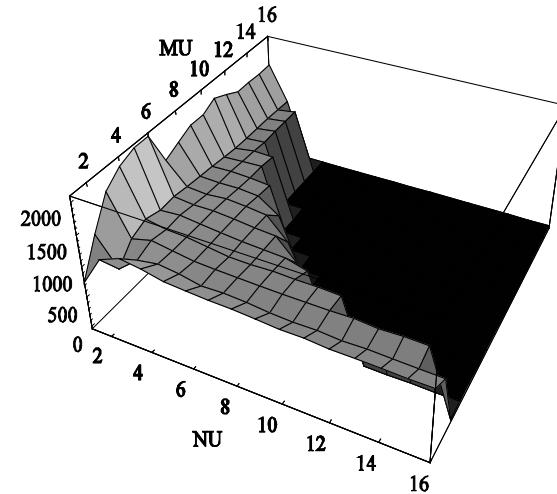
## MMM Performance

### Performance of mini-MMM

- ATLAS CGw/S: 2072 MFlops
- ATLAS Model: 1282 MFlops

### Key differences in parameter values: MU/NU

- ATLAS CGw/S: (6,1)
- ATLAS Model: (2,1)



## MU,NU Sensitivity

# Opteron diagnosis and solution

- Opteron characteristics
  - Small number of logical registers
  - Out-of-order issue
  - Register renaming
- For such processors, it is better to
  - let hardware take care of scheduling dependent instructions,
  - use logical registers to implement a bigger register tile.
- x86 has 8 logical registers
  - → register tiles must be of the form (x,1) or (1,x)

# Refined model

- **Estimating  $FMA$ :**  
Use the machine parameter  $FMA$
- **Estimating  $L_s$ :**

$$L_s = \left\lceil \frac{L_* \times |ALUFP| + 1}{2} \right\rceil$$

- **Estimating  $M_U$  and  $N_U$ :**

$$M_U \times N_U + N_U + M_U + L_s \leq N_R$$

- 1)  $M_U, N_U \leftarrow u.$
  - 2) Solve constraint for  $u$ .
  - 3)  $M_U \leftarrow \max(u, 1).$
  - 4) Solve constraint for  $N_U$ .
  - 5)  $N_U \leftarrow \max(N_U, 1).$
  - 6) If  $M_U < N_U$  then swap  $M_U$  and  $N_U$ .
  - 7) **Refined Model:** If  $N_U = 1$  then
    - $M_U \leftarrow N_R - 2$
    - $N_U \leftarrow 1$
    - $FMA \leftarrow 1$
- **Estimating  $N_B$ :**

$$\left\lceil \frac{N_B^2}{B_1} \right\rceil + 3 \left\lceil \frac{N_B \times N_U}{B_1} \right\rceil + \left\lceil \frac{M_U}{B_1} \right\rceil \times N_U \leq \frac{C_1}{B_1}$$

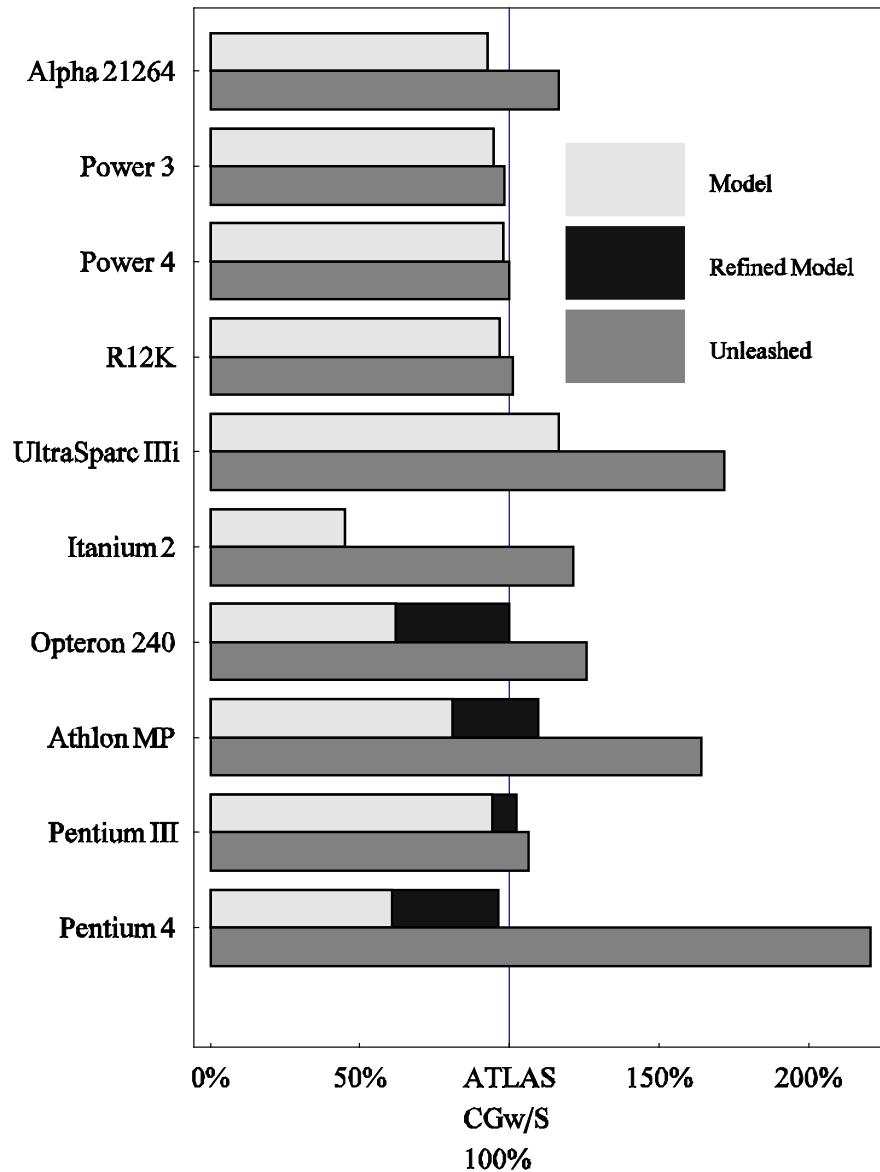
Trim  $N_B$ , to make it a multiple of  $M_U$ ,  $N_U$ , and 2.

- **Estimating  $K_U$ :**  
Choose  $K_U$  as the maximum value for which mini-MMM fits in the L1 instruction cache. Trim  $K_U$  to make it divide  $N_B$  evenly.
- **Estimating  $F_F$ ,  $I_F$ , and  $N_F$ :**

$$F_F = 0, I_F = 2, N_F = 2$$

# Bottom line

- Refined model is not complex.
- Refined model by itself eliminates most performance gaps.
- Local search eliminates all performance gaps.



# Future Directions

- Repeat study with FFTW/SPIRAL
  - Uses search to choose between algorithms
- Feed insights back into compilers
  - Build a linear algebra compiler for generating high-performance code for dense linear algebra codes
    - Start from high-level algorithmic descriptions
    - Use restructuring compiler technology
    - Part IBM PERCS Project
  - Generalize to other problem domains

# Program Optimization Through Loop Vectorization

María Garzarán, Saeed Maleki

William Gropp and David Padua

*Department of Computer Science  
University of Illinois at Urbana-Champaign*



# Program Optimization Through Loop Vectorization

Materials for this tutorial can be found:

<http://polaris.cs.uiuc.edu/~garzaran/pldi-polv.zip>

Questions?

Send an email to [garzaran@uiuc.edu](mailto:garzaran@uiuc.edu)



# Topics covered in this tutorial

- What are the microprocessor vector extensions or SIMD (Single Instruction Multiple Data Units)
- How to use them
  - Through the compiler via automatic vectorization
    - Manual transformations that enable vectorization
    - Directives to guide the compiler
  - Through intrinsics
- Main focus on vectorizing through the compiler.
  - Code more readable
  - Code portable



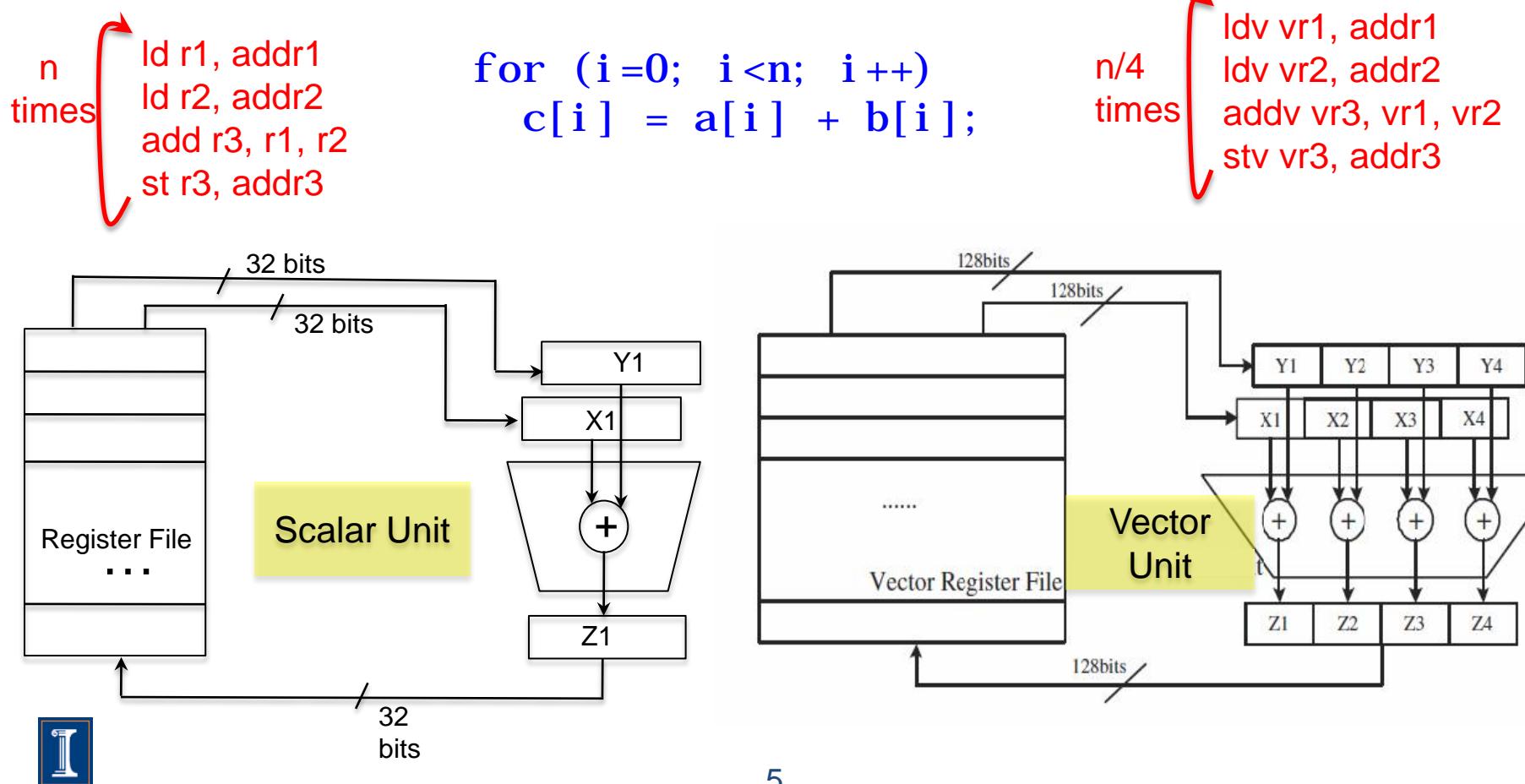
# Outline

1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
  - Data Dependences
  - Data Alignment
  - Aliasing
  - Non-unit strides
  - Conditional Statements
4. Vectorization with intrinsics



# Simple Example

- Loop vectorization transforms a program so that the same operation is performed at the same time on several vector elements



# SIMD Vectorization

- The use of SIMD units can speed up the program.
- Intel SSE and IBM Altivec have 128-bit vector registers and functional units
  - 4 32-bit single precision floating point numbers
  - 2 64-bit double precision floating point numbers
  - 4 32-bit integer numbers
  - 2 64 bit integer
  - 8 16-bit integer or shorts
  - 16 8-bit bytes or chars
- Assuming a single ALU, these SIMD units can execute 4 single precision floating point number or 2 double precision operations in the time it takes to do only one of these operations by a scalar unit.



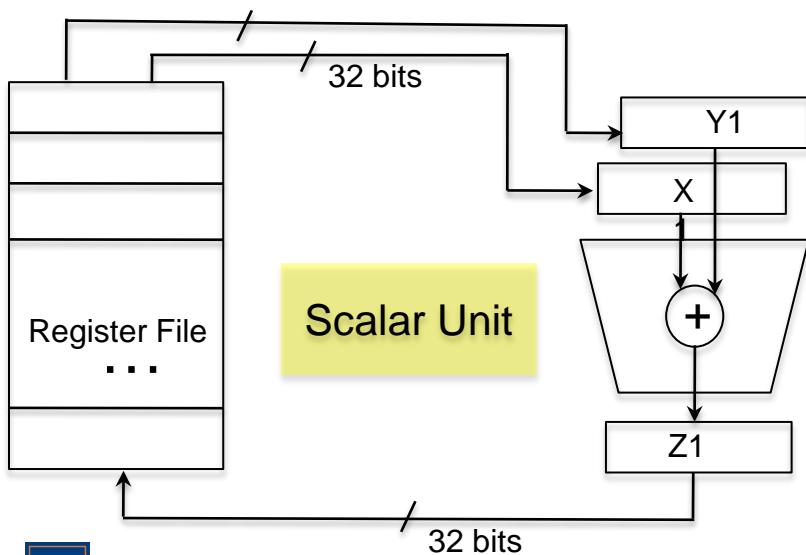
# Executing Our Simple Example

S000

```
for (i=0; i<n; i++)  
    c[i] = a[i] + b[i];
```

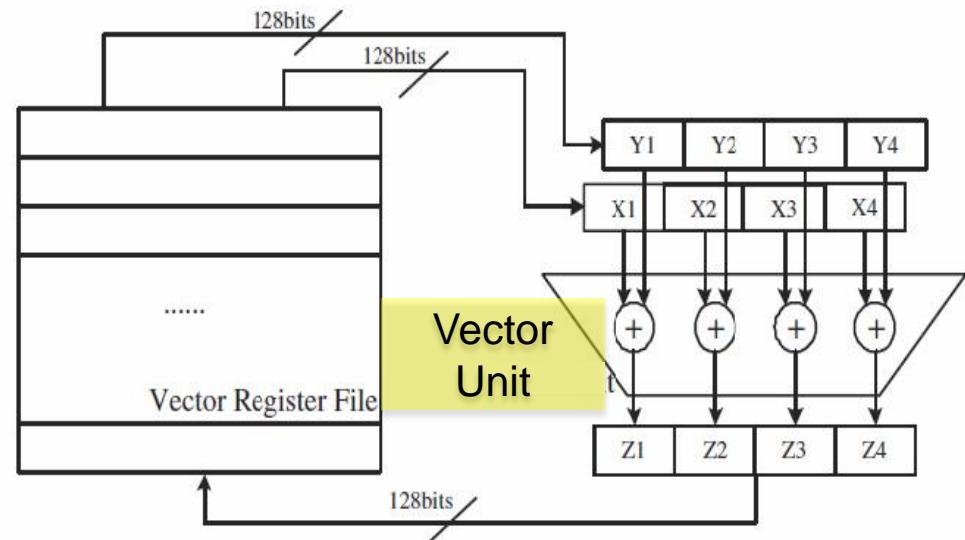
## Intel Nehalem

Exec. Time scalar code: 6.1  
Exec. Time vector code: 3.2  
Speedup: 1.8



## IBM Power 7

Exec. Time scalar code: 2.1  
Exec. Time vector code: 1.0  
Speedup: 2.1



# How do we access the SIMD units?

- Three choices
  1. C code and a vectorizing compiler

```
for (i=0; i<LEN; i++)  
    c[i] = a[i] + b[i];
```

1. Macros or Vector Intrinsics

```
void example(){  
    __m128 rA, rB, rC;  
    for (int i = 0; i < LEN; i+=4){  
        rA = _mm_load_ps(&a[i]);  
        rB = _mm_load_ps(&b[i]);  
        rC = _mm_add_ps(rA, rB);  
        _mm_store_ps(&c[i], rC);  
    } }
```

1. Assembly Language

```
... B8. 5  
movaps    a(, %rdx, 4), %xmm0  
addps    b(, %rdx, 4), %xmm0  
movaps    %xmm0, c(, %rdx, 4)  
addq     $4, %rdx  
cmpq     $rdi, %rdx  
jl      ... B8. 5
```





# Why should the compiler vectorize?

1. Easier
2. Portable across vendors and machines
  - Although compiler directives differ across compilers
3. Better performance of the compiler generated code
  - Compiler applies other transformations

Compilers make your codes (almost) machine independent

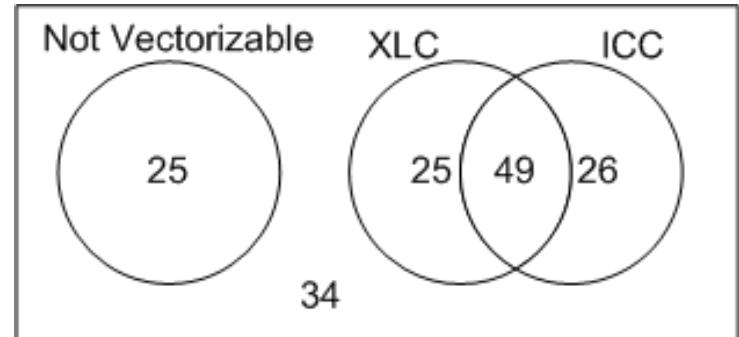
But, compilers fail:

- Programmers need to provide the necessary information
- Programmers need to transform the code



# How well do compilers vectorize?

| Loops \ Compiler | XLC  | ICC  | GCC  |
|------------------|------|------|------|
| Total            |      | 159  |      |
| Vectorized       | 74   | 75   | 32   |
| Not vectorized   | 85   | 84   | 127  |
| Average Speed Up | 1.73 | 1.85 | 1.30 |



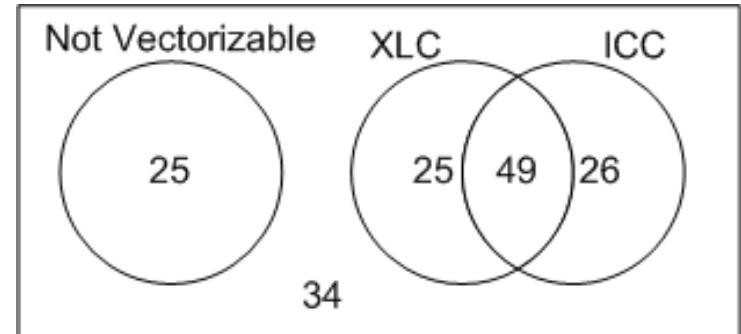
| Loops \ Compiler | XLC but not ICC | ICC but not XLC |
|------------------|-----------------|-----------------|
| Vectorized       | 25              | 26              |



# How well do compilers vectorize?

| Loops \ Compiler | XLC  | ICC  | GCC  |
|------------------|------|------|------|
| Total            | 159  |      |      |
| Vectorized       | 74   | 75   | 32   |
| Not vectorized   | 85   | 84   | 127  |
| Average Speed Up | 1.73 | 1.85 | 1.30 |

| Loops \ Compiler | XLC but<br>not ICC | ICC but<br>not XLC |
|------------------|--------------------|--------------------|
| Vectorized       | 25                 | 26                 |



By adding manual vectorization the average speedup was 3.78 (versus 1.73 obtained by the XLC compiler)

# How much programmer intervention?

- Next, three examples to illustrate what the programmer may need to do:
  - Add compiler directives
  - Transform the code
  - Program using vector intrinsics



# Experimental results

- The tutorial shows results for two different platforms with their compilers:
  - Report generated by the compiler
  - Execution Time for each platform

Platform 1: Intel Nehalem  
Intel Core i7 CPU 920@2.67GHz  
Intel ICC compiler, version 11.1  
OS Ubuntu Linux 9.04

Platform 2: IBM Power 7  
IBM Power 7, 3.55 GHz  
IBM xlc compiler, version 11.0  
OS Red Hat Linux Enterprise 5.4

The examples use single precision floating point numbers



# Compiler directives

```
void test(float* A, float* B, float* C, float* D, float* E)
{
    for (int i = 0; i < LEN; i++) {
        A[i] = B[i] + C[i] + D[i] + E[i];
    }
}
```



# Compiler directives

S1111

```
void test(float* A, float* B, float*  
C, float* D, float* E)  
{  
    for (int i = 0; i < LEN; i++) {  
        A[i] = B[i] + C[i] + D[i] + E[i];  
    }  
}
```

S1111

S1111

```
void test(float* __restrict__ A,  
float* __restrict__ B,  
float* __restrict__ C,  
float* __restrict__ D,  
float* __restrict__ E)  
{  
    for (int i = 0; i < LEN; i++) {  
        A[i] = B[i] + C[i] + D[i] + E[i];  
    }  
}
```

S1111

## Intel Nehalem

**Compiler report:** Loop was not vectorized.

**Exec. Time scalar code:** 5.6

**Exec. Time vector code:** --

**Speedup:** --



## Intel Nehalem

**Compiler report:** Loop was vectorized.

**Exec. Time scalar code:** 5.6

**Exec. Time vector code:** 2.2

**Speedup:** 2.5

# Compiler directives

S1111

```
void test(float* A, float* B, float*  
C, float* D, float* E)  
{  
    for (int i = 0; i < LEN; i++) {  
        A[i] = B[i] + C[i] + D[i] + E[i];  
    }  
}
```

S1111

S1111

```
void test(float* __restrict__ A,  
float* __restrict__ B,  
float* __restrict__ C,  
float* __restrict__ D,  
float* __restrict__ E)  
{  
    for (int i = 0; i < LEN; i++) {  
        A[i] = B[i] + C[i] + D[i] + E[i];  
    }  
}
```

S1111

## Power 7

**Compiler report:** Loop was not vectorized.

**Exec. Time scalar code:** 2.3

**Exec. Time vector code:** --

**Speedup:** --



## Power 7

**Compiler report:** Loop was vectorized.

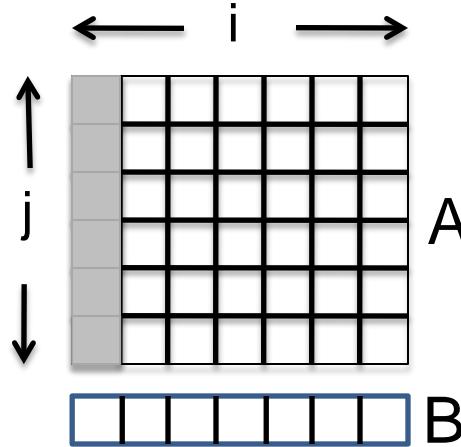
**Exec. Time scalar code:** 1.6

**Exec. Time vector code:** 0.6

**Speedup:** 2.7

# Loop Transformations

```
for (int i=0; i<LEN; i++) {  
    sum = (float) 0.0;  
    for (int j=0; j<LEN; j++) {  
        sum += A[j][i];  
    }  
    B[i] = sum;  
}
```



```
for (int i=0; i<size; i++) {  
    sum[i] = 0;  
    for (int j=0; j<size; j++) {  
        sum[i] += A[j][i];  
    }  
    B[i] = sum[i];  
}
```

# Loop Transformations

S136

```
for (int i=0; i<LEN; i++) {  
    sum = (float) 0. 0;  
    for (int j=0; j<LEN; j++) {  
        sum += A[j][i];  
    }  
    B[i] = sum;  
}
```

S136\_1

```
for (int i=0; i<LEN; i++)  
    sum[i] = (float) 0. 0;  
    for (int j=0; j<LEN; j++) {  
        sum[i] += A[j][i];  
    }  
    B[i]=sum[i];  
}
```

S136\_2

```
for (int i=0; i<LEN; i++)  
    B[i] = (float) 0. 0;  
    for (int j=0; j<LEN; j++) {  
        B[i] += A[j][i];  
    }  
}
```

S136

**Intel Nehalem**

**Compiler report:** Loop was not vectorized. Vectorization possible but seems inefficient

**Exec. Time scalar code:** 3.7

**Exec. Time vector code:** --

**Speedup:** --

S136\_1

**Intel Nehalem**  
**report:** Permuted loop was vectorized.

**scalar code:** 1.6

**vector code:** 0.6

**Speedup:** 2.6

S136\_2

**Intel Nehalem**  
**report:** Permuted loop was vectorized.

**scalar code:** 1.6

**vector code:** 0.6

**Speedup:** 2.6



# Loop Transformations

S136

```
for (int i=0; i<LEN; i++) {  
    sum = (float) 0. 0;  
    for (int j=0; j<LEN; j++) {  
        sum += A[j][i];  
    }  
    B[i] = sum;  
}
```

S136\_1

```
for (int i=0; i<LEN; i++)  
    sum[i] = (float) 0. 0;  
    for (int j=0; j<LEN; j++) {  
        sum[i] += A[j][i];  
    }  
    B[i]=sum[i];  
}
```

S136\_2

```
for (int i=0; i<LEN; i++)  
    B[i] = (float) 0. 0;  
    for (int j=0; j<LEN; j++) {  
        B[i] += A[j][i];  
    }  
}
```

S136

**IBM Power 7**

**Compiler report:** Loop was not SIMD vectorized

**Exec. Time scalar code:** 2.0

**Exec. Time vector code:** --

**Speedup:** --

S136\_1

**IBM Power 7**

**report:** Loop interchanging applied.

Loop was SIMD vectorized

**scalar code:** 0.4

**vector code:** 0.2

**Speedup:** 2.0

S136\_2

**IBM Power 7**

**report:** Loop interchanging applied.

Loop was SIMD

**scalar code:** 0.4

**vector code:** 0.16

**Speedup:** 2.7



# Intrinsics (SSE)

```
#define n 1024
__attribute__((aligned(16))) float a[n], b[n], c[n];
int main() {
    for (i = 0; i < n; i++) {
        c[i]=a[i]*b[i];
    }
}
```



```
#include <xmmi ntrin.h>
#define n 1024
__attribute__((aligned(16))) float a[n], b[n], c[n];

int main() {
    __m128 rA, rB, rC;
    for (i = 0; i < n; i+=4) {
        rA = _mm_load_ps(&a[i]);
        rB = _mm_load_ps(&b[i]);
        rC= _mm_mul_ps(rA, rB);
        _mm_store_ps(&c[i], rC);
    }
}
```



# Intrinsics (Altivec)

```
#define n 1024
__attribute__((aligned(16))) float a[n], b[n], c[n];
...
for (int i=0; i<LEN; i++)
    c[i]=a[i]*b[i];
```



```
vector float rA, rB, rC, r0;          // Declares vector registers
r0 = vec_xor(r0, r0);                  // Sets r0 to {0,0,0,0}
for (int i=0; i<LEN; i+=4){           // Loop stride is 4
    rA = vec_ld(0, &a[i]);            // Load values to rA
    rB = vec_ld(0, &b[i]);            // Load values to rB
    rC = vec_madd(rA, rB, r0);       // rA and rB are multiplied
    vec_st(rC, 0, &c[i]);           // rC is stored to the c[i:i+3]
}
```



# Outline

1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
  - Data Dependences
  - Data Alignment
  - Aliasing
  - Non-unit strides
  - Conditional Statements
4. Vectorization with intrinsics



# Data dependences

- The notion of dependence is the foundation of the process of vectorization.
- It is used to build a calculus of program transformations that can be applied manually by the programmer or automatically by a compiler.



# Definition of Dependence

- A statement S is said to be data dependent on statement T if
  - T executes before S in the original sequential/scalar program
  - S and T access the same data item
  - At least one of the accesses is a write.

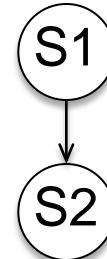


# Data Dependence

Flow dependence (True dependence)

S1:  $X = A + B$

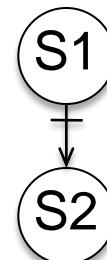
S2:  $C = X + A$



Anti dependence

S1:  $A = X + B$

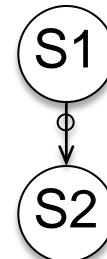
S2:  $X = C + D$



Output dependence

S1:  $X = A + B$

S2:  $X = C + D$



# Data Dependence

- Dependences indicate an execution order that must be honored.
- Executing statements in the order of the dependences guarantee correct results.
- Statements not dependent on each other can be reordered, executed in parallel, or coalesced into a vector operation.





# Dependences in Loops (I)

- Dependences in loops are easy to understand if the loops are unrolled. Now the dependences are between statement “executions”.

```
for (i=0; i<n; i++) {  
    S1  a[i] = b[i] + 1;  
    S2  c[i] = a[i] + 2;  
}
```



# Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

```
for (i=0; i<n; i++) {  
    S1 a[i] = b[i] + 1;  
    S2 c[i] = a[i] + 2;  
}  
i=0           i=1           i=2
```

S1:  $a[0] = b[0] + 1$   
S2:  $c[0] = a[0] + 2$

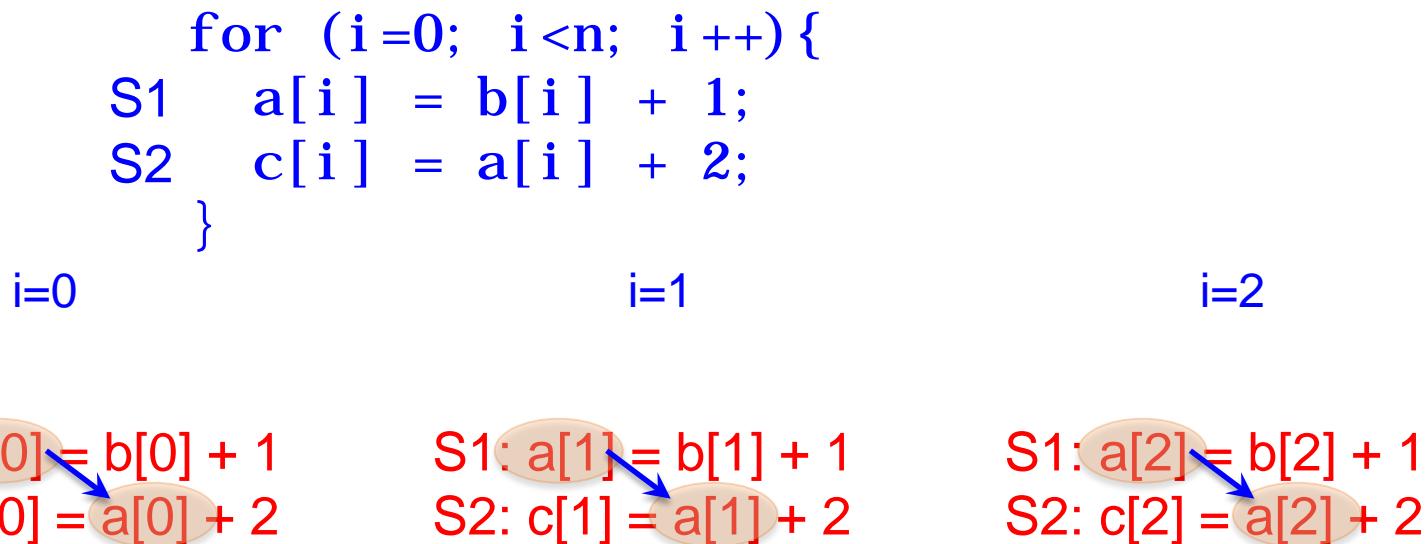
S1:  $a[1] = b[1] + 1$   
S2:  $c[1] = a[1] + 2$

S1:  $a[2] = b[2] + 1$   
S2:  $c[2] = a[2] + 2$



# Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

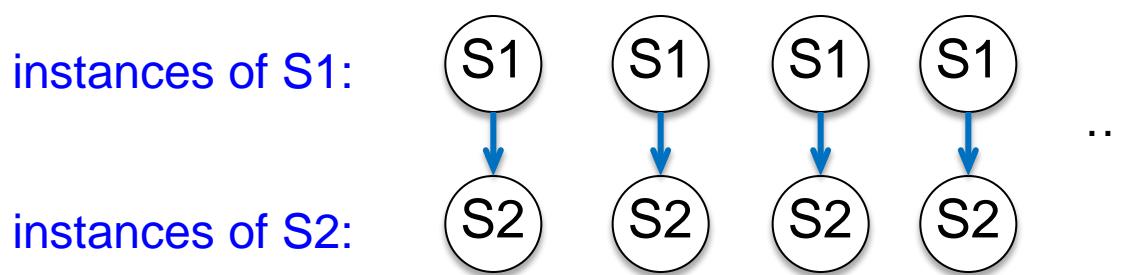


# Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

```
for (i=0; i<n; i++) {  
    S1 a[i] = b[i] + 1;  
    S2 c[i] = a[i] + 2;  
}
```

iteration:      0      1      2      3      ...



# Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

```
for (i=0; i<n; i++) {  
    S1 a[i] = b[i] + 1;  
    S2 c[i] = a[i] + 2;  
}
```

iteration:      0      1      2      3      ...

instances of S1:      S1      S1      S1      S1      ...

instances of S2:      S2      S2      S2      S2

→ Loop independent dependence

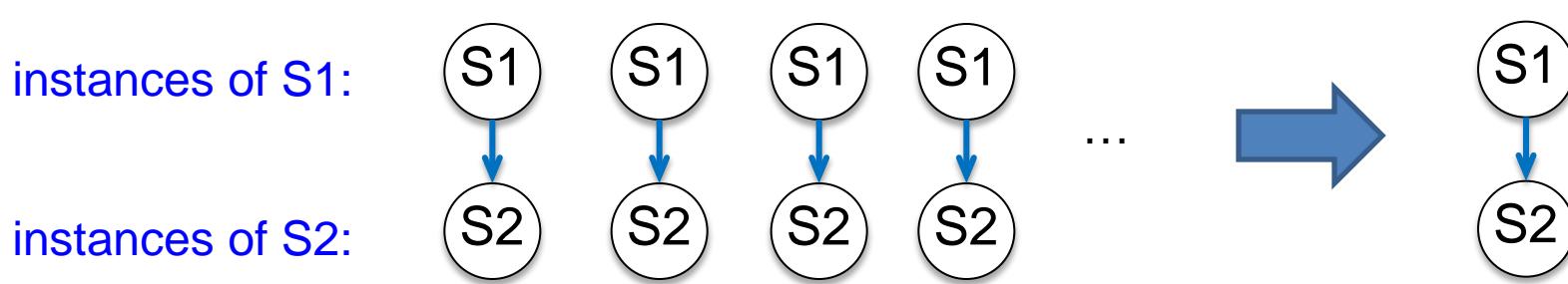


# Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

```
for (i=0; i<n; i++) {  
    S1 a[i] = b[i] + 1;  
    S2 c[i] = a[i] + 2;  
}
```

iteration:      0      1      2      3      ...



# Dependences in Loops (I)

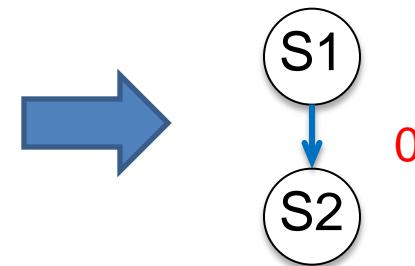
- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

```
for (i=0; i<n; i++) {  
    S1 a[i] = b[i] + 1;  
    S2 c[i] = a[i] + 2;  
}
```

iteration:      0      1      2      3      ...

instances of S1:      S1      S1      S1      S1      ...

instances of S2:      S2      S2      S2      S2



For the whole loop



# Dependences in Loops (I)

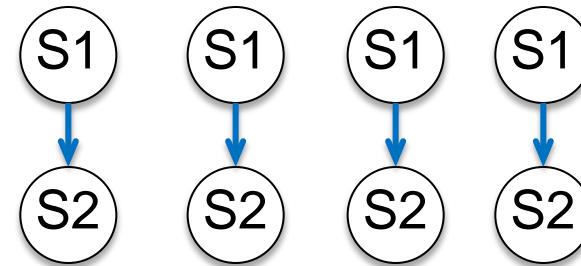
- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

```
for (i=0; i<n; i++) {  
    S1 a[i] = b[i] + 1;  
    S2 c[i] = a[i] + 2;  
}
```

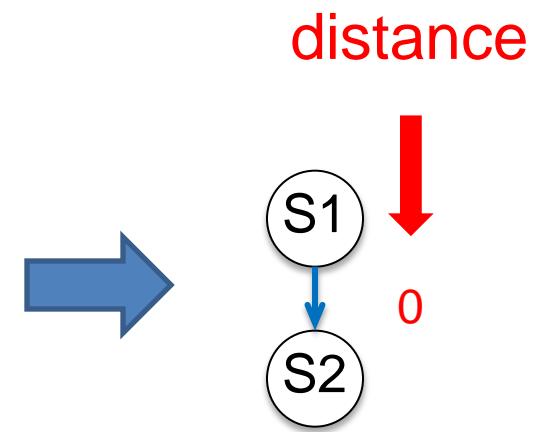
iteration:

0      1      2      3      ...

instances of S1:



instances of S2:



For the whole loop



# Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

```
for (i=0; i<n; i++) {  
    S1  a[i] = b[i] + 1;  
    S2  c[i] = a[i] + 2;  
}
```

For the dependences shown here, we assume that arrays do not overlap in memory (no aliasing). Compilers must know that there is no aliasing in order to vectorize.



# Dependences in Loops (II)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

```
for (i=1; i<n; i++) {  
    S1 a[i] = b[i] + 1;  
    S2 c[i] = a[i-1] + 2;  
}
```



# Dependences in Loops (II)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

```
for (i=1; i<n; i++) {  
    S1 a[i] = b[i] + 1;  
    S2 c[i] = a[i-1] + 2;  
}
```

i=1

i=2

i=3

S1: a[1] = b[1] + 1  
S2: c[1] = a[0] + 2

S1: a[2] = b[2] + 1  
S2: c[2] → a[1] + 2

S1: a[3] = b[3] + 1  
S2: c[3] → a[2] + 2

# Dependences in Loops (II)

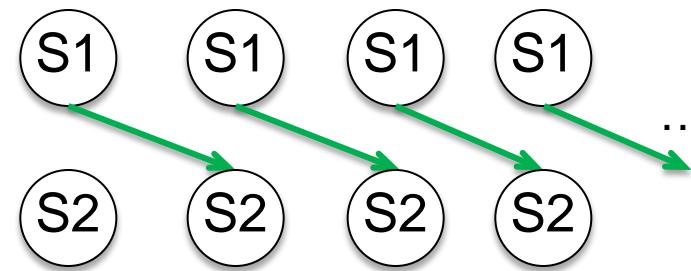
- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

```
for (i=1; i<n; i++) {  
    S1  a[i] = b[i] + 1;  
        c[i] = a[i-1] + 2;  
}
```

iteration:

1      2      3      4      ...

instances of S1:



instances of S2:

# Dependences in Loops (II)

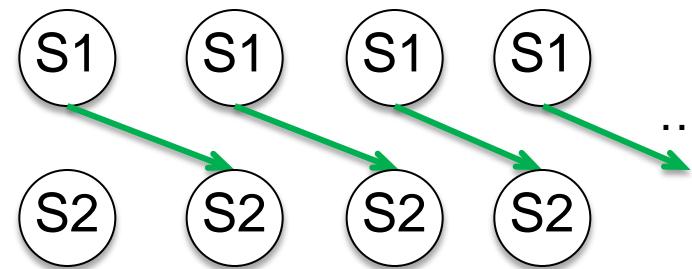
- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

```
for (i=1; i<n; i++) {  
    S1  a[i] = b[i] + 1;  
        c[i] = a[i-1] + 2;  
}
```

iteration:

1      2      3      4      ...

instances of S1:



instances of S2:

→ Loop carried dependence

# Dependences in Loops (II)

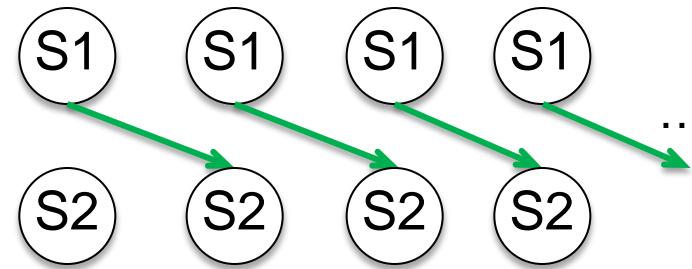
- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

```
for (i=1; i<n; i++) {  
    S1  a[i] = b[i] + 1;  
    c[i] = a[i-1] + 2;  
}
```

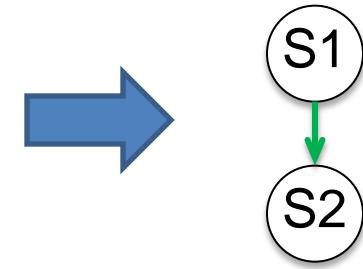
iteration:

1      2      3      4      ...

instances of S1:



instances of S2:



For the whole loop



# Dependences in Loops (II)

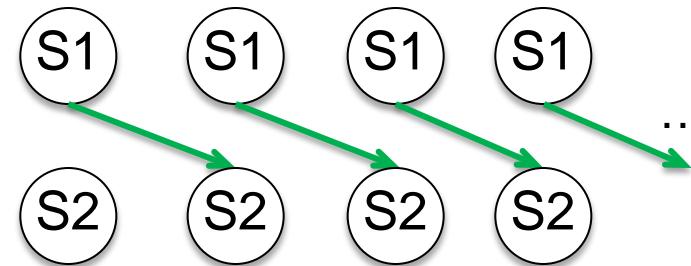
- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

```
for (i=1; i<n; i++) {  
    S1  a[i] = b[i] + 1;  
    c[i] = a[i-1] + 2;  
}
```

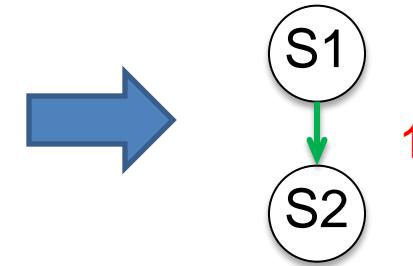
iteration:

1      2      3      4      ...

instances of S1:



instances of S2:

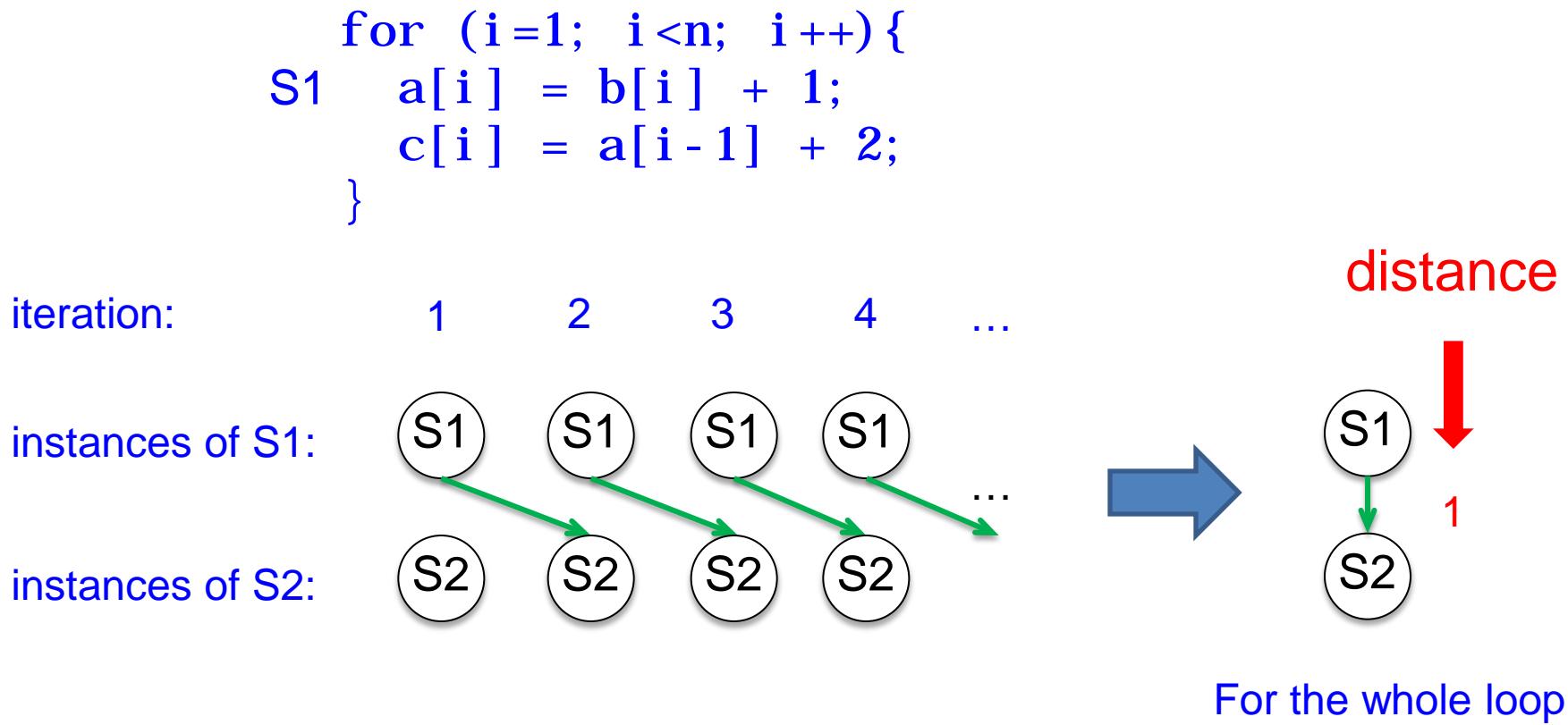


For the whole loop



# Dependences in Loops (II)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"



# Dependences in Loops (III)

- Dependences in loops are easy to understand if loops are unrolled.  
Now the dependences are between statement “executions”

```
for (i=0; i<n; i++) {  
    S1    a = b[i] + 1;  
    S2    c[i] = a + 2;  
}
```



# Dependences in Loops (III)

```
for (i=0; i<n; i++) {  
S1    a = b[i] + 1;  
S2    c[i] = a + 2;  
}
```

i=0

i=1

i=2

S1: a= b[0] + 1  
S2: c[0] = a + 2

S1: a = b[1] + 1  
S2: c[1] = a + 2

S1: a = b[2] + 1  
S2: c[2] = a+ 2



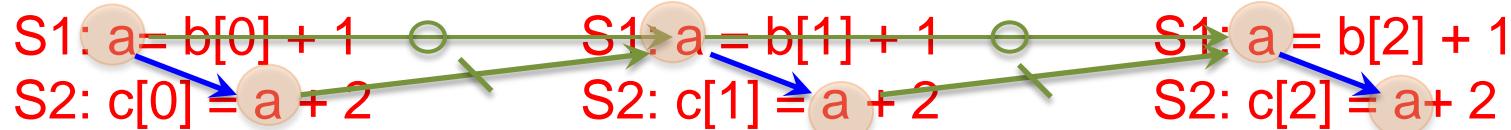
# Dependences in Loops (III)

```
for (i=0; i<n; i++) {  
S1    a = b[i] + 1;  
S2    c[i] = a + 2;  
}
```

i=0

i=1

i=2



- Loop independent dependence
- Loop carried dependence



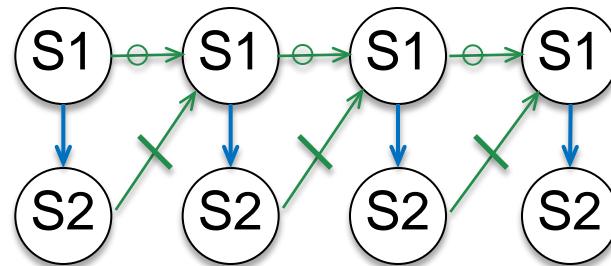
# Dependences in Loops (III)

```
for (i=0; i<n; i++) {  
S1    a = b[i] + 1;  
S2    c[i] = a + 2;  
}
```

iteration:

0      1      2      3      ...

instances of S1:



instances of S2:

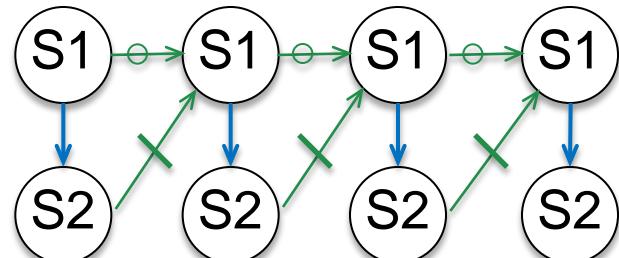
# Dependences in Loops (III)

```
for (i=0; i<n; i++) {  
S1    a = b[i] + 1;  
S2    c[i] = a + 2;  
}
```

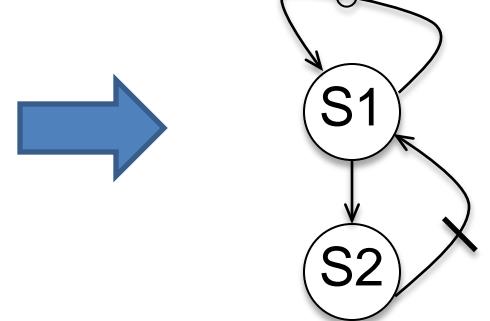
iteration:

0      1      2      3      ...

instances of S1:



instances of S2:





# Dependences in Loops (IV)

- Doubly nested loops

```
for (i=1; i<n; i++) {  
    for (j=1; j<n; j++) {  
S1    a[i][j]=a[i][j - 1]+a[i - 1][j];  
    }  
}
```



# Dependences in Loops (IV)

```
for (i=1; i<n; i++) {  
    for (j=1; j<n; j++) {  
S1    a[i][j]=a[i][j - 1]+a[i - 1][j];  
    }  
}
```

i=1

$$\begin{array}{ll} j=1 & a[1][1] = a[1][0] + a[0][1] \\ j=2 & a[1][2] = a[1][1] + a[0][2] \\ j=3 & a[1][3] = a[1][2] + a[0][3] \\ j=4 & a[1][4] = a[1][3] + a[0][4] \end{array}$$

i=2

$$\begin{array}{ll} & a[2][1] = a[2][0] + a[1][1] \\ & a[2][2] = a[2][1] + a[1][2] \\ & a[2][3] = a[2][2] + a[1][3] \\ & a[2][4] = a[2][3] + a[1][4] \end{array}$$

Loop carried dependences

# Dependences in Loops (IV)

```
for (i=1; i<n; i++) {  
    for (j=1; j<n; j++) {  
S1    a[i][j]=a[i][j - 1]+a[i - 1][j];  
    } }
```

i=1

i=2

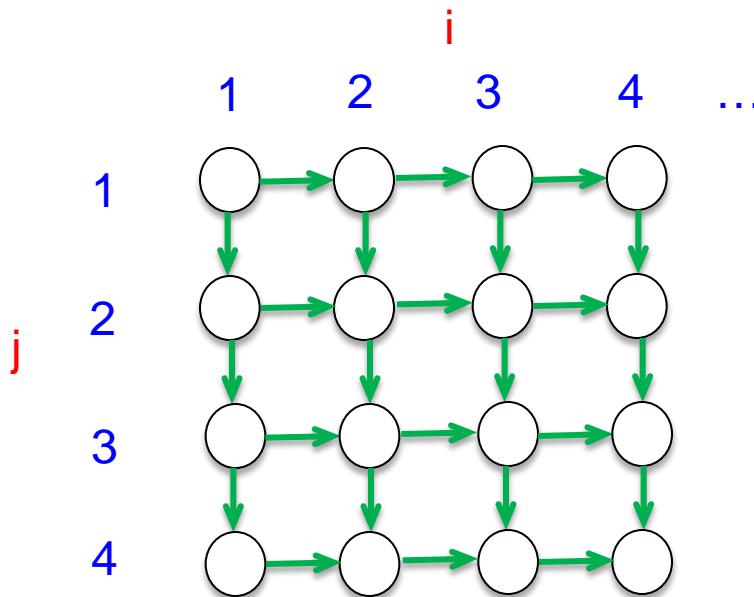
|     |                               |                                         |
|-----|-------------------------------|-----------------------------------------|
| j=1 | $a[1][1] = a[1][0] + a[0][1]$ | $a[2][1] = a[2][0] \rightarrow a[1][1]$ |
| j=2 | $a[1][2] = a[1][1] + a[0][2]$ | $a[2][2] = a[2][1] \rightarrow a[1][2]$ |
| j=3 | $a[1][3] = a[1][2] + a[0][3]$ | $a[2][3] = a[2][2] \rightarrow a[1][3]$ |
| j=4 | $a[1][4] = a[1][3] + a[0][4]$ | $a[2][4] = a[2][3] \rightarrow a[1][4]$ |

Loop carried dependences



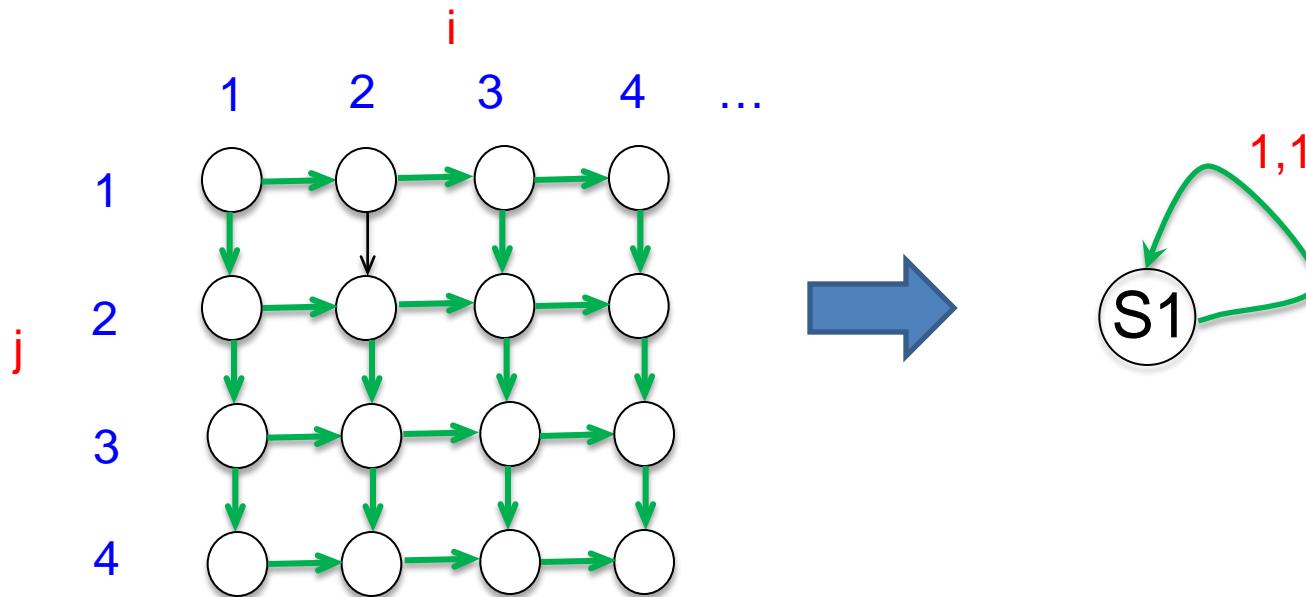
# Dependences in Loops (IV)

```
for (i=1; i<n; i++) {  
    for (j=1; j<n; j++) {  
S1    a[i][j]=a[i][j - 1]+a[i - 1][j];  
    }  
}
```



# Dependences in Loops (IV)

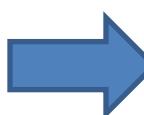
```
for (i=1; i<n; i++) {  
    for (j=1; j<n; j++) {  
S1    a[i][j]=a[i][j - 1]+a[i - 1][j];  
    }  
}
```



# Data dependences and vectorization

- Loop dependences guide vectorization
- Main idea: A statement inside a loop which is not in a cycle of the dependence graph can be vectorized.

```
for (i=0; i<n; i++) {  
S1  a[i] = b[i] + 1;  
}
```



```
a[0:n-1] = b[0:n-1] + 1;
```

S1



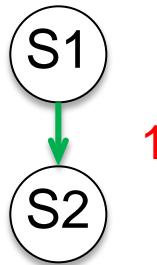
# Data dependences and vectorization

- Main idea: A statement inside a loop which is not in a cycle of the dependence graph can be vectorized.

```
for (i=1; i<n; i++) {  
S1  a[i] = b[i] + 1;  
S2  c[i] = a[i - 1] + 2;  
}
```



```
a[1:n] = b[1:n] + 1;  
c[1:n] = a[0:n-1] + 2;
```



# Data dependences and transformations

- When cycles are present, vectorization can be achieved by:
  - Separating (distributing) the statements not in a cycle
  - Removing dependences
  - Freezing loops
  - Changing the algorithm

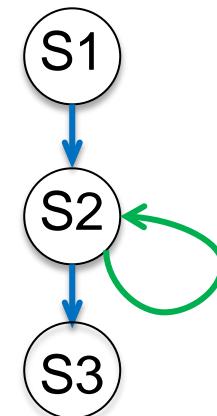


# Distributing

```
for (i=1; i<n; i++) {  
S1 b[i] = b[i] + c[i];  
S2 a[i] = a[i-1]*a[i-2]+b[i];  
S3 c[i] = a[i] + 1;  
}
```



```
b[1:n-1] = b[1:n-1] + c[1:n-1];  
for (i=1; i<n; i++) {  
    a[i] = a[i-1]*a[i-2]+b[i];  
}  
c[1:n-1] = a[1:n-1] + 1;
```



# Removing dependences

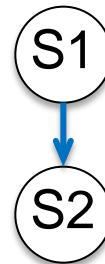
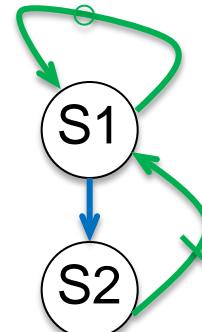
```
for (i=0; i<n; i++) {  
S1    a = b[i] + 1;  
S2    c[i] = a + 2;  
}
```



```
for (i=0; i<n; i++) {  
S1    a' [i] = b[i] + 1;  
S2    c[i] = a' [i] + 2;  
}  
a=a' [n- 1]
```

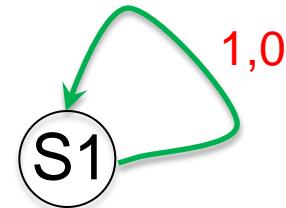


```
S1    a' [0: n- 1] = b[0: n- 1] + 1;  
S2    c[0: n- 1] = a' [0: n- 1] + 2;  
a=a' [n- 1]
```



# Freezing Loops

```
for (i=1; i<n; i++) {  
    for (j=1; j<n; j++) {  
        a[i][j]=a[i][j]+a[i-1][j];  
    }  
}
```



Ignoring (freezing) the outer loop:

```
for (j=1; j<n; j++) {  
    a[i][j]=a[i][j]+a[i-1][j];  
}
```



```
for (i=1; i<n; i++) {  
    a[i][1:n-1]=a[i][1:n-1]+a[i-1][1:n-1];  
}
```



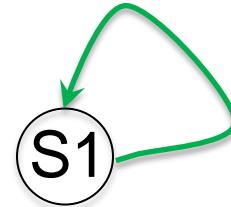
# Changing the algorithm

- When there is a recurrence, it is necessary to change the algorithm in order to vectorize.
- Compiler use pattern matching to identify the recurrence and then replace it with a parallel version.
- Examples or recurrences include:
  - Reductions ( $S+=A[i]$ )
  - Linear recurrences ( $A[i]=B[i]*A[i-1]+C[i]$ )
  - Boolean recurrences ( $i\ f\ (A[i]>\max)\ \max = A[i]$ )



# Changing the algorithm (cont.)

```
S1  a[0]=b[0];  
    for (i=1; i<n; i++)  
S2      a[i]=a[i-1]+b[i];
```



```
a[0:n-1]=b[0:n-1];  
for (i=0;i<k;i++) /* n = 2k */  
    a[2**i:n-1]=a[2**i:n-1]+b[0:n-2**i];
```

# Stripmining

- Stripmining is a simple transformation.

```
for (i=1; i<n; i++) {           /* n is a multiple of q */  
    ...  
}   ...  
      for (k=1; k<n; k+=q) {  
          for (i=k; i<k+q-1; i++) {  
              ...  
          }  
      }  
  }
```

- It is typically used to improve locality.



# Stripmining (cont.)

- Stripmining is often used when vectorizing

```
for (i=1; i<n; i++) {  
    a[i] = b[i] + 1;  
    c[i] = a[i] + 2;  
}
```



stripmine

```
for (k=1; k<n; k+=q) {  
    /* q could be size of vector register */  
    for (i=k; i < k+q; i++) {  
        a[i] = b[i] + 1;  
        c[i] = a[i-1] + 2;  
    }  
}
```



vectorize

```
for (i=1; i<n; i+=q) {  
    a[i:i+q-1] = b[i:i+q-1] + 1;  
    c[i:i+q-1] = a[i:i+q] + 2;  
}
```



# Outline

1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
  - Data Dependences
  - Data Alignment
  - Aliasing
  - Non-unit strides
  - Conditional Statements
4. Vectorization with intrinsics



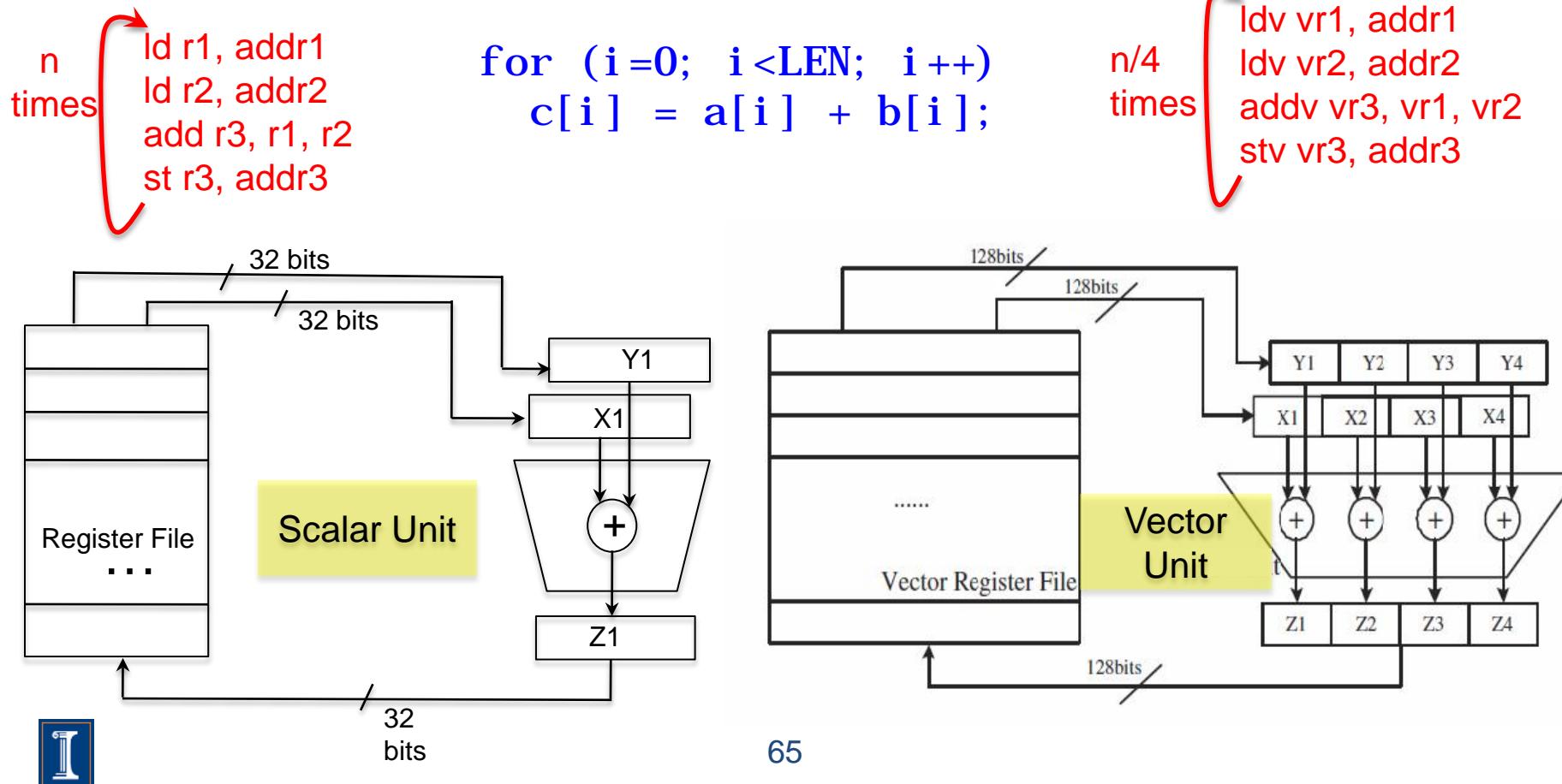
# Loop Vectorization

- Loop Vectorization is not always a legal and profitable transformation.
- Compiler needs:
  - Compute the dependences
    - The compiler figures out dependences by
      - Solving a system of (integer) equations (with constraints)
      - Demonstrating that there is no solution to the system of equations
  - Remove cycles in the dependence graph
  - Determine data alignment
  - Vectorization is profitable



# Simple Example

- Loop vectorization transforms a program so that the same operation is performed at the same time on several of the elements of the vectors



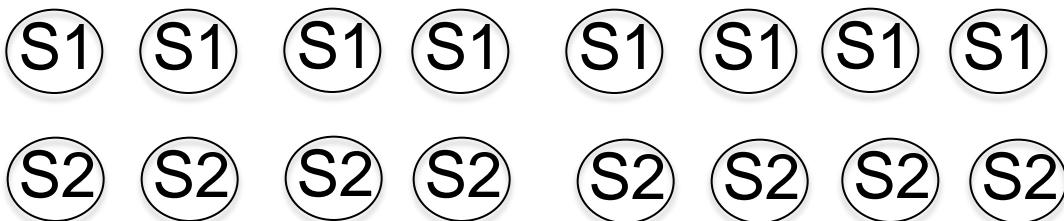
# Loop Vectorization

- When vectorizing a loop with several statements the compiler need to strip-mine the loop and then apply loop distribution

```
for (i=0; i<LEN; i++) {  
S1 a[i]=b[i]+(float)1.0;  
S2 c[i]=b[i]+(float)2.0;  
}
```

```
for (i=0; i<LEN; i+=strip_size) {  
    for (j=i; j<i+strip_size; j++)  
        a[j]=b[j]+(float)1.0;  
    for (j=i; j<i+strip_size; j++)  
        c[j]=b[j]+(float)2.0;  
}
```

i=0    i=1    i=2    i=3    i=4    i=5    i=6    i=7



# Loop Vectorization

- When vectorizing a loop with several statements the compiler need to strip-mine the loop and then apply loop distribution

```
for (i=0; i<LEN; i++) {  
S1 a[i]=b[i]+(float)1.0;  
S2 c[i]=b[i]+(float)2.0;  
}
```

```
for (i=0; i<LEN; i+=strip_size) {  
    for (j=i; j<i+strip_size; j++)  
        a[j]=b[j]+(float)1.0;  
    for (j=i; j<i+strip_size; j++)  
        c[j]=b[j]+(float)2.0;  
}
```

i=0    i=1    i=2    i=3    i=4    i=5    i=6    i=7



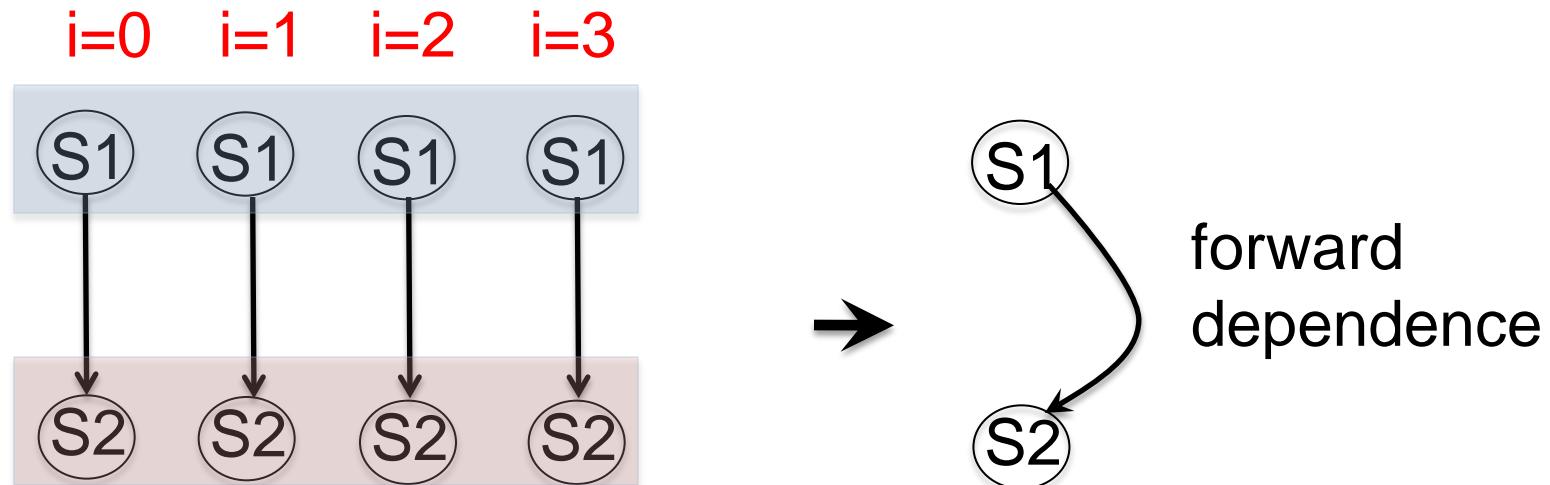
# Dependence Graphs and Compiler Vectorization

- No dependences: previous two slides
- Acyclic graphs:
  - All dependences are forward:
    - Vectorized by the compiler
  - Some backward dependences:
    - Sometimes vectorized by the compiler
- Cycles in the dependence graph
  - Self-antidependence:
    - Vectorized by the compiler
  - Recurrence:
    - Usually not vectorized by the compiler
  - Other examples



# Acyclic Dependence Graphs: Forward Dependences

```
for (i=0; i<LEN; i++) {  
S1 a[i] = b[i] + c[i]  
S2 d[i] = a[i] + (float) 1.0;  
}
```



# Acyclic Dependence Graphs: Forward Dependencies

S113

```
for (i=0; i<LEN; i++) {  
    a[i] = b[i] + c[i]  
    d[i] = a[i] + (float) 1.0;  
}
```

## Intel Nehalem

**Compiler report:** Loop was  
vectorized

**Exec. Time scalar code:** 10.2

**Exec. Time vector code:** 6.3

**Speedup:** 1.6

## IBM Power 7

**Compiler report:** Loop was SIMD  
vectorized

**Exec. Time scalar code:** 3.1

**Exec. Time vector code:** 1.5

**Speedup:** 2.0



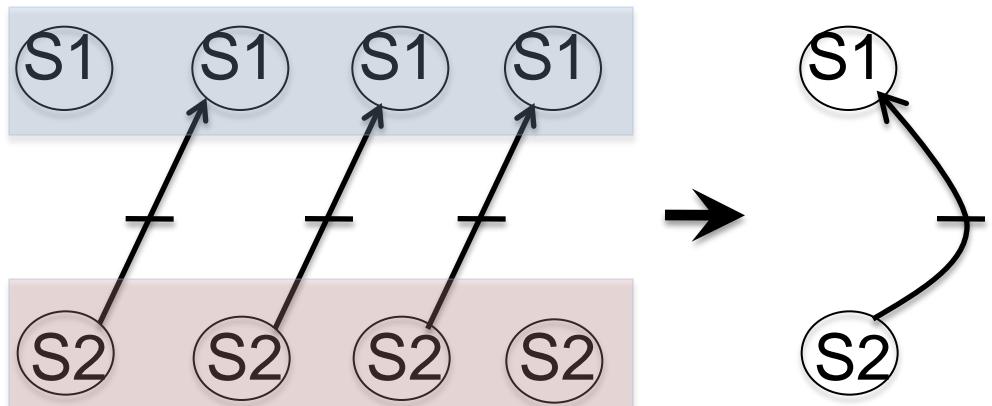
# Acyclic Dependenden Graphs

## Backward Dependencies (I)

```
for (i=0; i<LEN; i++) {  
S1 a[i] = b[i] + c[i]  
S2 d[i] = a[i+1] + (float) 1.0;  
}
```

backward  
dependence

i=0 {  
S1: a[0] = b[0] + c[0]  
S2: d[0] = a[1] + 1  
} ---  
i=1 {  
S1: a[1] = b[0] + c[0]  
S2: d[1] = a[2] + 1  
} ---



This loop cannot be vectorized as it is



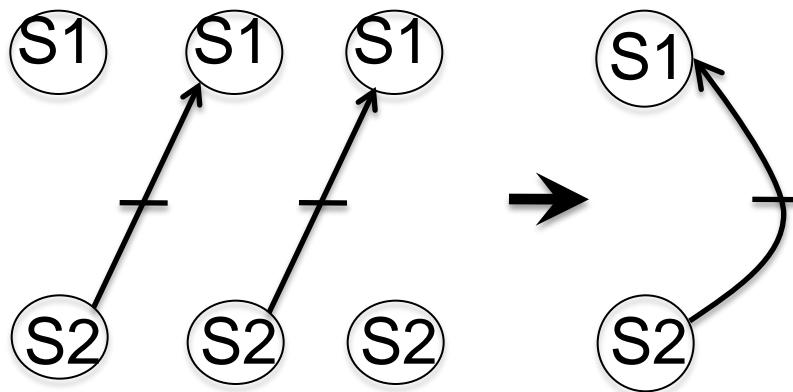
# Acyclic Dependenden Graphs

## Backward Dependencies (I)

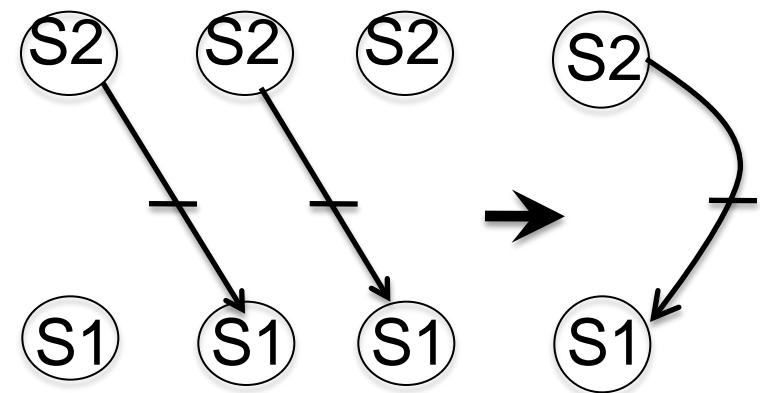
Reorder of statements

```
for (i=0; i<LEN; i++) {  
S1 a[i] = b[i] + c[i]  
S2 d[i] = a[i+1] + (float) 1.0;  
}
```

```
for (i=0; i<LEN; i++) {  
S2 d[i] = a[i+1]+(float) 1. 0;  
S1 a[i]= b[i] + c[i];  
}
```



backward  
depedence



forward  
depedence



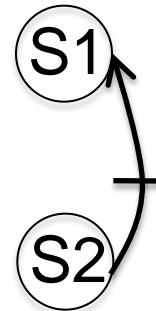
# Acyclic Dependenden Graphs

## Backward Dependencies (I)

S114

```
for (i=0; i<LEN; i++) {  
    a[i] = b[i] + c[i];  
    d[i] = a[i+1]+(float)1.0;  
}
```

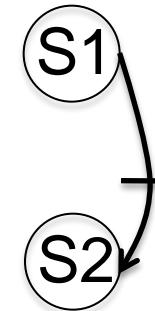
S114



S114\_1

```
for (i=0; i<LEN; i++) {  
    d[i] = a[i+1]+(float)1.0;  
    a[i] = b[i] + c[i];  
}
```

S114\_1



### Intel Nehalem

**Compiler report:** Loop was not vectorized. Existence of vector dependence

**Exec. Time scalar code:** 12.6

**Exec. Time vector code:** --

**Speedup:** --

### Intel Nehalem

**Compiler report:** Loop was vectorized

**Exec. Time scalar code:** 10.7

**Exec. Time vector code:** 6.2

**Speedup:** 1.72

**Speedup vs non-reordered code:** 2.03





# Acyclic Dependenden Graphs

## Backward Dependencies (I)

S114

```
for (i=0; i<LEN; i++) {  
    a[i] = b[i] + c[i];  
    d[i] = a[i+1]+(float)1.0;  
}
```

S114\_1

```
for (i=0; i<LEN; i++) {  
    d[i] = a[i+1]+(float)1.0;  
    a[i] = b[i] + c[i];  
}
```

The IBM XLC compiler generated the same code in both cases

S114

**IBM Power 7**  
**Compiler report:** Loop was SIMD vectorized  
**Exec. Time scalar code:** 1.2  
**Exec. Time vector code:** 0.6  
**Speedup:** 2.0

S114\_1

**IBM Power 7**  
**Compiler report:** Loop was SIMD vectorized  
**Exec. Time scalar code:** 1.2  
**Exec. Time vector code:** 0.6  
**Speedup:** 2.0



# Acyclic Dependenden Graphs

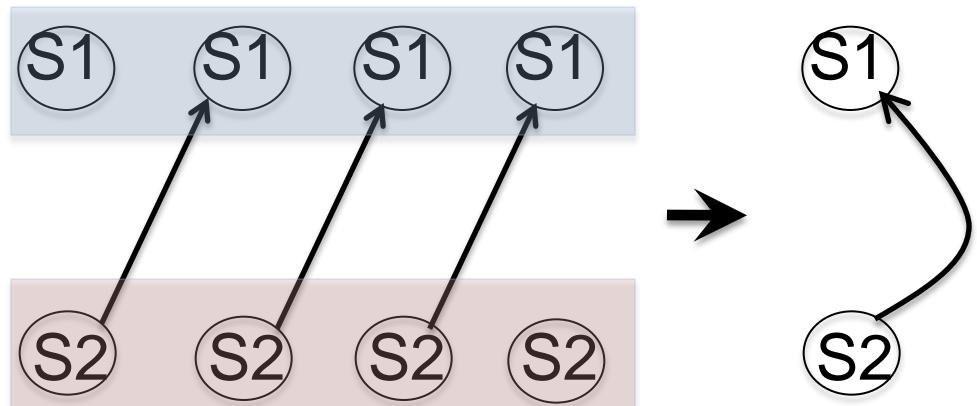
## Backward Dependencies (II)

```
for (int i = 1; i < LEN; i++) {  
S1 a[i] = d[i-1] + (float)sqrt(c[i]);  
S2 d[i] = b[i] + (float)sqrt(e[i]);  
}
```

backward  
dependence

$i=1 \left\{ \begin{array}{l} S1: a[1] = d[0] + \sqrt{c[1]} \\ S2: d[1] = b[1] + \sqrt{e[1]} \end{array} \right.$

$i=2 \left\{ \begin{array}{l} S1: a[2] = d[1] + \sqrt{c[2]} \\ S2: d[2] = b[2] + \sqrt{e[2]} \end{array} \right.$



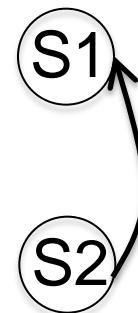
This loop cannot be vectorized as it is

# Acyclic Dependenden Graphs

## Backward Dependences (II)

S214

```
for (int i=1; i<LEN; i++) {  
    a[i]=d[i - 1]+(float)sqrt(c[i]);  
    d[i]=b[i ]+(float)sqrt(e[i ]);  
}
```



S214\_1

```
for (int i=1; i<LEN; i++) {  
    d[i]=b[i ]+(float)sqrt(e[i ]);  
    a[i ]=d[i - 1]+(float)sqrt(c[i ]);  
}
```



S114

### Intel Nehalem

**Compiler report:** Loop was not vectorized. Existence of vector dependence

**Exec. Time scalar code:** 7.6

**Exec. Time vector code:** --

**Speedup:** --

S114\_1

### Intel Nehalem

**Compiler report:** Loop was vectorized

**Exec. Time scalar code:** 7.6

**Exec. Time vector code:** 3.8

**Speedup:** 2.0





# Acyclic Dependenden Graphs

## Backward Dependences (II)

S114

```
for (i=0; i<LEN; i++) {  
    a[i] = b[i] + c[i];  
    d[i] = a[i+1]+(float)1.0;  
}
```

S114\_1

```
for (i=0; i<LEN; i++) {  
    d[i] = a[i+1]+(float)1.0;  
    a[i] = b[i] + c[i];  
}
```

The IBM XLC compiler generated the same code in both cases

S114

**IBM Power 7**  
**Compiler report:** Loop was SIMD vectorized  
**Exec. Time scalar code:** 3.3  
**Exec. Time vector code:** 1.8  
**Speedup:** 1.8

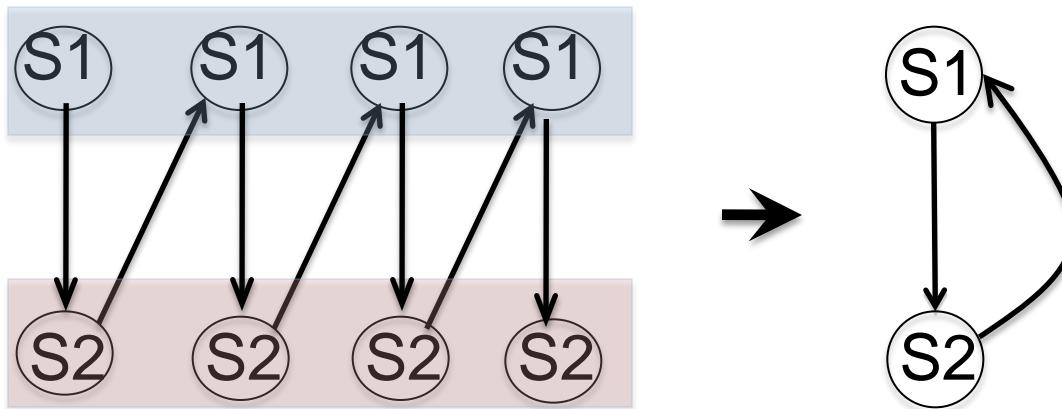
S114\_1

**IBM Power 7**  
**Compiler report:** Loop was SIMD vectorized  
**Exec. Time scalar code:** 3.3  
**Exec. Time vector code:** 1.8  
**Speedup:** 1.8



# Cycles in the DG (I)

```
for (int i=0; i<LEN- 1; i++) {  
S1   b[i] = a[i] + (float) 1.0;  
S2   a[i+1] = b[i] + (float) 2.0;  
}
```



This loop cannot be vectorized (as it is)  
Statements cannot be simply reordered

# Cycles in the DG (I)

S115

```
for (int i=0; i<LEN- 1; i++) {  
    b[i] = a[i] + (float) 1.0;  
    a[i+1] = b[i] + (float) 2.0;  
}
```

S115

## Intel Nehalem

**Compiler report:** Loop was not vectorized.

Existence of vector dependence

**Exec. Time scalar code:** 12.1

**Exec. Time vector code:** --

**Speedup:** --



# Cycles in the DG (I)

S115

```
for (int i=0; i<LEN- 1; i++) {  
    b[i] = a[i] + (float) 1.0;  
    a[i+1] = b[i] + (float) 2.0;  
}
```

S115

**IBM Power 7**

**Compiler report:** Loop was SIMD vectorized

**Exec. Time scalar code:** 3.1

**Exec. Time vector code:** 2.2

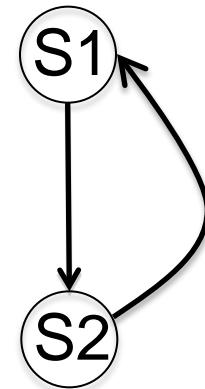
**Speedup:** 1.4



# Cycles in the DG (I)

S115

```
for (int i=0; i<LEN- 1; i++) {  
    b[i] = a[i] + (float) 1. 0;  
    a[i+1] = b[i] + (float) 2. 0;  
}
```



The IBM XLC compiler applies forward substitution and reordering to vectorize the code

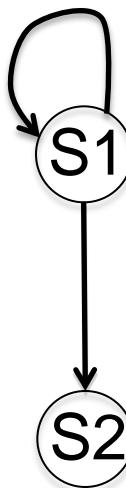
This loop is  
not vectorized

This loop is  
vectorized

compiler generated code

```
for (int i=0; i<LEN- 1; i++)  
    a[i+1]=a[i]+(float)1. 0+(float)2. 0;
```

```
for (int i=0; i<LEN- 1; i++)  
    b[i] = a[i] + (float) 1. 0;
```





# Cycles in the DG (I)

S115

```
for (int i=0; i<LEN- 1; i++) {  
    b[i] =a[i]+(float)1. 0;  
    a[i+1]=b[i]+(float)2. 0;  
}
```

S215

```
for (int i=0; i<LEN- 1; i++) {  
    b[i]=a[i]+d[i]*d[i]+c[i]*c[i]+c[i]*d[i];  
    a[i+1]=b[i]+(float)2. 0;  
}
```

Will the IBM XLC compiler  
vectorize this code as before?



# Cycles in the DG (I)

S115

```
for (int i=0; i<LEN- 1; i++) {  
    b[i] =a[i]+(float) 1. 0;  
    a[i+1]=b[i]+(float) 2. 0;  
}
```

S215

```
for (int i=0; i<LEN- 1; i++) {  
    b[i]=a[i]+d[i]*d[i]+c[i]*c[i]+c[i]*d[i];  
    a[i+1]=b[i]+(float) 2. 0;  
}
```

Will the IBM XLC compiler  
vectorize this code as before?

To vectorize, the compiler needs to do this

```
for (int i=0; i<LEN- 1; i++)  
    a[i+1]=a[i]+d[i]*d[i]+c[i]*c[i]+c[i]*d[i]+(float) 2. 0;  
  
for (int i=0; i<LEN- 1; i++)  
    b[i]=a[i]+d[i]*d[i]+c[i]*c[i]+c[i]*d[i]+(float) 1. 0;
```



# Cycles in the DG (I)

S115

```
for (int i=0; i<LEN- 1; i++) {  
    b[i] =a[i]+(float) 1. 0;  
    a[i+1]=b[i]+(float) 2. 0;  
}
```

```
for (int i=0; i<LEN- 1; i++)  
    a[i+1]=a[i]+d[i]*d[i]+c[i]*c[i]+c[i]*d[i]+(float) 2. 0;
```

```
for (int i=0; i<LEN- 1; i++)  
    b[i]=a[i]+d[i]*d[i]+c[i]*c[i]+c[i]*d[i]+(float) 1. 0;
```

S215

```
for (int i=0; i<LEN- 1; i++) {  
    b[i]=a[i]+d[i]*d[i]+c[i]*c[i]+c[i]*d[i];  
    a[i+1]=b[i]+(float) 2. 0;  
}
```

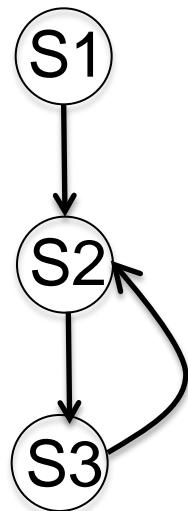
Will the IBM XLC compiler  
vectorize this code as before?

No, the compiler does not  
vectorize S215 because  
it is not cost-effective

# Cycles in the DG (II)

A loop can be partially vectorized

```
for (int i=1; i<LEN; i++) {  
S1  a[i] = b[i] + c[i];  
S2  d[i] = a[i] + e[i - 1];  
S3  e[i] = d[i] + c[i];  
}
```



S1 can be vectorized  
S2 and S3 cannot be vectorized (as they are)

# Cycles in the DG (II)

S116

```
for (int i=1; i<LEN; i++) {  
    a[i] = b[i] + c[i];  
    d[i] = a[i] + e[i-1];  
    e[i] = d[i] + c[i];  
}
```

S116

```
for (int i=1; i<LEN; i++) {  
    a[i] = b[i] + c[i];  
    d[i] = a[i] + e[i-1];  
    e[i] = d[i] + c[i];  
}
```

S116

**Intel Nehalem**  
**Compiler report:** Loop was partially vectorized  
**Exec. Time scalar code:** 14.7  
**Exec. Time vector code:** 18.1  
**Speedup:** 0.8

S116

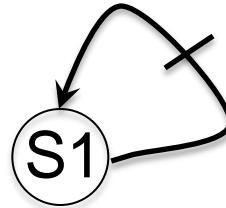
**IBM Power 7**  
**Compiler report:** Loop was not SIMD vectorized because a data dependence prevents SIMD vectorization  
**Exec. Time scalar code:** 13.5  
**Exec. Time vector code:** --  
**Speedup:** --



# Cycles in the DG (III)

```
for (int i=0; i<LEN- 1; i++) {  
S1  a[i]=a[i+1]+b[i];  
}
```

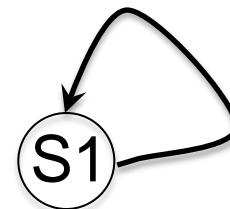
$$\begin{aligned}a[0] &= a[1]+b[0] \\a[1] &= a[2]+b[1] \\a[2] &= a[3]+b[2] \\a[3] &= a[4]+b[3]\end{aligned}$$



Self-antidependence  
can be vectorized

```
for (int i=1; i<LEN; i++) {  
S1  a[i]=a[i-1]+b[i];  
}
```

$$\begin{aligned}a[1] &= a[0]+b[1] \\a[2] &= a[1]+b[2] \\a[3] &= a[2]+b[3] \\a[4] &= a[3]+b[4]\end{aligned}$$



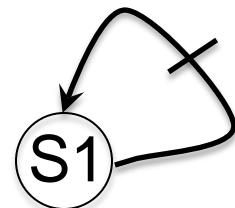
Self true-dependence  
**can not** vectorized  
(as it is)



# Cycles in the DG (III)

S117

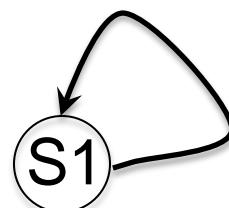
```
for (int i=0; i<LEN- 1; i++) {  
S1  a[i ]=a[i+1]+b[i ];  
}
```



S117

S118

```
for (int i=1; i<LEN; i++) {  
S1  a[i ]=a[i - 1]+b[i ];  
}
```



S118

## Intel Nehalem

**Compiler report:** Loop was vectorized

**Exec. Time scalar code:** 6.0

**Exec. Time vector code:** 2.7

**Speedup:** 2.2

## Intel Nehalem

**Compiler report:** Loop was not vectorized. Existence of vector dependence

**Exec. Time scalar code:** 7.2

**Exec. Time vector code:** --

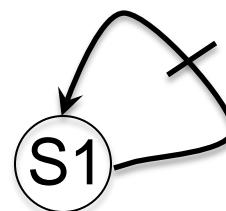
**Speedup:** --



# Cycles in the DG (III)

S117

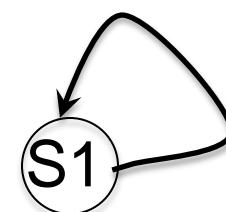
```
for (int i=0; i<LEN- 1; i++) {  
S1 a[i]=a[i+1]+b[i];  
}
```



S117

S118

```
for (int i=1; i<LEN; i++) {  
S1 a[i]=a[i - 1]+b[i];  
}
```



S118

## IBM Power 7

**Compiler report:** Loop was SIMD vectorized

**Exec. Time scalar code:** 2.0

**Exec. Time vector code:** 1.0

**Speedup:** 2.0

## IBM Power 7

**Compiler report:** Loop was not SIMD vectorized because a data dependence prevents SIMD vectorization

**Exec. Time scalar code:** 7.2

**Exec. Time vector code:** --

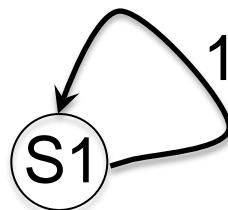
**Speedup:** --



# Cycles in the DG (IV)

```
for (int i=1; i<LEN; i++) {  
S1  a[i]=a[i-1]+b[i];  
}
```

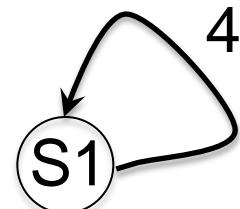
$$\begin{aligned}a[1] &= a[0]+b[1] \\a[2] &= a[1]+b[2] \\a[3] &= a[2]+b[3]\end{aligned}$$



Self true-dependence  
is not vectorized

```
for (int i=4; i<LEN; i++) {  
  a[i]=a[i-4]+b[i];  
}
```

$$\begin{aligned}i=4 \quad a[4] &= a[0]+b[4] \\i=5 \quad a[5] &= a[1]+b[5] \\i=6 \quad a[6] &= a[2]+b[6] \\i=7 \quad a[7] &= a[3]+b[7] \\i=8 \quad a[8] &= a[4]+b[8] \\i=9 \quad a[9] &= a[5]+b[9] \\i=10 \quad a[10] &= a[6]+b[10] \\i=11 \quad a[11] &= a[7]+b[11]\end{aligned}$$



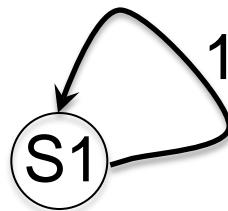
This is also a self-true dependence. But ...  
can it be vectorized?



# Cycles in the DG (IV)

```
for (int i=1; i<n; i++) {  
S1  a[i]=a[i-1]+b[i];  
}
```

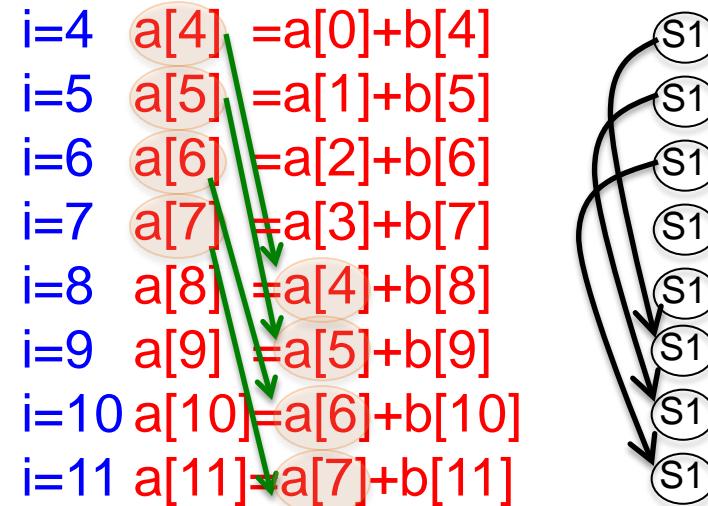
$$\begin{aligned}a[1] &= a[0]+b[1] \\a[2] &= a[1]+b[2] \\a[3] &= a[2]+b[3]\end{aligned}$$



Self true-dependence  
**cannot** be vectorized

```
for (int i=4; i<LEN; i++) {  
  a[i]=a[i-4]+b[i];  
}
```

$$\begin{aligned}i=4 \quad a[4] &= a[0]+b[4] \\i=5 \quad a[5] &= a[1]+b[5] \\i=6 \quad a[6] &= a[2]+b[6] \\i=7 \quad a[7] &= a[3]+b[7] \\i=8 \quad a[8] &= a[4]+b[8] \\i=9 \quad a[9] &= a[5]+b[9] \\i=10 \quad a[10] &= a[6]+b[10] \\i=11 \quad a[11] &= a[7]+b[11]\end{aligned}$$



Yes, it can be vectorized because the dependence distance is 4, which is the number of iterations that the SIMD unit can execute simultaneously.



# Cycles in the DG (IV)

S119

```
for (int i=4; i<LEN; i++) {  
    a[i]=a[i-4]+b[i];  
}
```

## Intel Nehalem

**Compiler report:** Loop was vectorized

**Exec. Time scalar code:** 8.4

**Exec. Time vector code:** 3.9

**Speedup:** 2.1

## IBM Power 7

**Compiler report:** Loop was SIMD vectorized

**Exec. Time scalar code:** 6.6

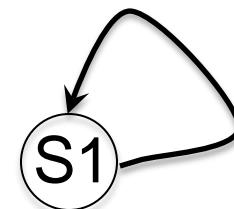
**Exec. Time vector code:** 1.8

**Speedup:** 3.7



# Cycles in the DG (V)

```
for (int i = 0; i < LEN-1; i++) {  
    for (int j = 0; j < LEN; j++)  
S1    a[i+1][j] = a[i][j] + b;  
}
```



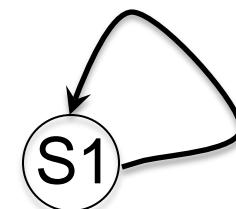
Can this loop be vectorized?

```
i=0, j=0: a[1][0] = a[0][0] + b  
          j=1: a[1][1] = a[0][1] + b  
          j=2: a[1][2] = a[0][2] + b  
i=1 j=0: a[2][0] = a[1][0] + b  
          j=1: a[2][1] = a[1][1] + b  
          j=2: a[2][2] = a[1][2] + b
```



# Cycles in the DG (V)

```
S1   for (int i = 0; i < LEN-1; i++) {  
      for (int j = 0; j < LEN; j++)  
         a[i+1][j] = a[i][j] + (float) 1.0;  
    }
```



Can this loop be vectorized?

```
i=0, j=0: a[1][0] = a[0][0] + 1  
          j=1: a[1][1] = a[0][1] + 1  
          j=2: a[1][2] = a[0][2] + 1  
i=1 j=0: a[2][0] = a[1][0] + 1  
          j=1: a[2][1] = a[1][1] + 1  
          j=2: a[2][2] = a[1][2] + 1
```

Dependences occur in the outermost loop.

- outer loop runs serially
- inner loop can be vectorized

```
for (int i=0; i<LEN; i++) {  
  a[i+1][0: LEN-1] = a[i][0: LEN-  
  1] + b;  
}
```



# Cycles in the DG (V)

S121

```
for (int i = 0; i < LEN- 1; i++) {  
    for (int j = 0; j < LEN; j++)  
        a[i+1][j] = a[i][j] + 1;  
}
```

## Intel Nehalem

**Compiler report:** Loop was vectorized

**Exec. Time scalar code:** 11.6

**Exec. Time vector code:** 3.2

**Speedup:** 3.5

## IBM Power 7

**Compiler report:** Loop was SIMD vectorized

**Exec. Time scalar code:** 3.9

**Exec. Time vector code:** 1.8

**Speedup:** 2.1



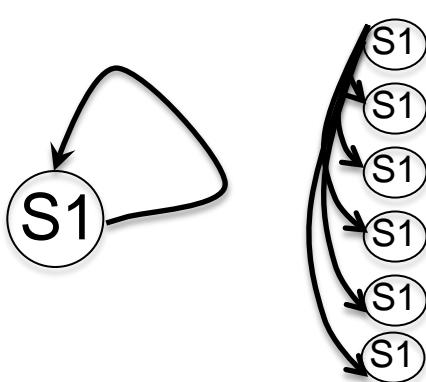
# Cycles in the DG (VI)

- Cycles can appear because the compiler does not know if there are dependences

```
for (int i=0; i<LEN; i++) {
```

```
S1   a[r[i]] = a[r[i]] * (float) 2.0;  
}
```

Is there a value of  $i$  such that  $r[i'] = r[i]$ , such that  $i' \neq i$ ?



Compiler cannot resolve the system

To be safe, it considers that a data dependence is possible for every instance of S1

# Cycles in the DG (VI)

- The compiler is conservative.
- The compiler only vectorizes when it can prove that it is safe to do it.

```
for (int i=0; i<LEN; i++) {  
    r[i] = i;  
    a[r[i]] = a[r[i]]* (float) 2.0;  
}
```

Does the compiler use the info that  $r[i] = i$  to compute data dependences?



# Cycles in the DG (VI)

S122

```
for (int i=0; i<LEN; i++) {  
    a[r[i]] = a[r[i]] * (float)2.0;  
}
```

S123

```
for (int i=0; i<LEN; i++) {  
    r[i] = i;  
    a[r[i]] = a[r[i]] * (float)2.0;  
}
```

Does the compiler uses the info that  
 $r[i] = i$  to compute data dependences?

S122

**Intel Nehalem**

**Compiler report:** Loop was not vectorized. Existence of vector dependence

**Exec. Time scalar code:** 5.0

**Exec. Time vector code:** --

**Speedup:** --

S123

**Intel Nehalem**

**Compiler report:** Partial Loop was vectorized

**Exec. Time scalar code:** 5.8

**Exec. Time vector code:** 5.7

**Speedup:** 1.01



# Cycles in the DG (VI)

S122

```
for (int i=0; i<LEN; i++) {  
    a[r[i]] = a[r[i]] * (float) 2.0;  
}
```

S123

```
for (int i=0; i<LEN; i++) {  
    r[i] = i;  
    a[r[i]] = a[r[i]] * (float) 2.0;  
}
```

Does the compiler uses the info that  
 $r[i] = i$  to compute data dependences?

S122

S123

## IBM Power 7

**Compiler report:** Loop was not vectorized because a data dependence prevents SIMD vectorization

**Exec. Time scalar code:** 2.6

**Exec. Time vector code:** 2.3

**Speedup:** 1.1

## IBM Power 7

**Compiler report:** Loop was SIMD vectorized

**Exec. Time scalar code:** 2.1

**Exec. Time vector code:** 0.9

**Speedup:** 2.3



# Dependence Graphs and Compiler Vectorization

- No dependences: Vectorized by the compiler
- Acyclic graphs:
  - All dependences are forward:
    - Vectorized by the compiler
  - Some backward dependences:
    - Sometimes vectorized by the compiler
- Cycles in the dependence graph
  - Self-antidependence:
    - Vectorized by the compiler
  - Recurrence:
    - Usually not vectorized by the compiler
  - Other examples



# Loop Transformations

- Compiler Directives
- Loop Distribution or loop fission
- Reordering Statements
- Node Splitting
- Scalar expansion
- Loop Peeling
- Loop Fusion
- Loop Unrolling
- Loop Interchanging



# Compiler Directives (I)

- When the compiler does not vectorize automatically due to dependences the programmer can inform the compiler that it is safe to vectorize:

```
#pragma ivdep (ICC compiler)
```

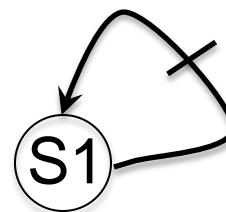
```
#pragma ibm independent_loop (XLC compiler)
```



# Compiler Directives (I)

- This loop can be vectorized when  $k < -3$  and  $k \geq 0$ .
- Programmer knows that  $k \geq 0$

```
for (int i=val ; i<LEN-k; i++)  
    a[i]=a[i+k]+b[i];
```

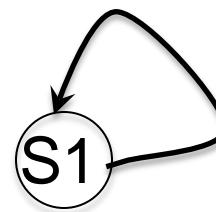


If ( $k \geq 0$ ) → no dependence  
or self-anti-dependence

$k = 1$

$$\begin{aligned}a[0] &= a[1] + b[0] \\a[1] &= a[2] + b[1] \\a[2] &= a[3] + b[2]\end{aligned}$$

Can  
be vectorized



If ( $k < 0$ ) → self-true dependence

$k = -1$

$$\begin{aligned}a[1] &= a[0] + b[0] \\a[2] &= a[1] + b[1] \\a[3] &= a[2] + b[2]\end{aligned}$$

Cannot  
be vectorized



# Compiler Directives (I)

- This loop can be vectorized when  $k < -3$  and  $k \geq 0$ .
- Programmer knows that  $k \geq 0$

How can the programmer tell the compiler that  $k \geq 0$

```
for (int i=val ; i<LEN-  
k; i++)  
    a[i]=a[i+k]+b[i];
```



# Compiler Directives (I)

- This loop can be vectorized when  $k < -3$  and  $k \geq 0$ .
- Programmer knows that  $k \geq 0$

Intel ICC provides the `#pragma ivdep` to tell the compiler that it is safe to ignore unknown dependences

```
#pragma ivdep
for (int i=val ; i<LEN-
k; i++)
    a[i]=a[i+k]+b[i];
```

wrong results will be obtained if loop is vectorized when  $-3 < k < 0$



# Compiler Directives (I)

S124

```
for (int i=0; i<LEN-k; i++)  
    a[i]=a[i+k]+b[i];
```

S124\_1

```
if (k>=0)  
    for (int i=0; i<LEN-k; i++)  
        a[i]=a[i+k]+b[i];  
if (k<0)  
    for (int i=0; i<LEN-k; i++)  
        a[i]=a[i+k]+b[i];
```

S124\_2

```
if (k>=0)  
#pragma ivdep  
for (int i=0; i<LEN-k; i++)  
    a[i]=a[i+k]+b[i];  
if (k<0)  
    for (int i=0; i<LEN-k; i++)  
        a[i]=a[i+k]+b[i];
```

S124 and S124\_1

**Intel Nehalem**

**Compiler report:** Loop was not vectorized. Existence of vector dependence

**Exec. Time scalar code:** 6.0

**Exec. Time vector code:** --

**Speedup:** --

S124\_2

**Intel Nehalem**

**Compiler report:** Loop was vectorized

**Exec. Time scalar code:** 6.0

**Exec. Time vector code:** 2.4

**Speedup:** 2.5



# Compiler Directives (I)

S124

```
for (int i=0; i<LEN-k; i++)  
    a[i]=a[i+k]+b[i];
```

S124\_1

```
if (k>=0)  
    for (int i=0; i<LEN-k; i++)  
        a[i]=a[i+k]+b[i];  
  
if (k<0)  
    for (int i=0; i<LEN-k; i++)  
        a[i]=a[i+k]+b[i];
```

S124\_2

```
if (k>=0)  
#pragma ibm independent_loop  
for (int i=0; i<LEN-k; i++)  
    a[i]=a[i+k]+b[i];  
  
if (k<0)  
    for (int i=0; i<LEN-k; i++)  
        a[i]=a[i+k]+b[i];
```

S124 and S124\_1

**IBM Power 7**

**Compiler report:** Loop was not vectorized because a data dependence prevents SIMD vectorization

**Exec. Time scalar code:** 2.2

**Exec. Time vector code:** --

**Speedup:** --

S124\_2

```
#pragma ibm independent_loop  
needs AIX OS (we ran the  
experiments on Linux)
```



# Compiler Directives (II)

- Programmer can disable vectorization of a loop when the when the vector code runs slower than the scalar code

```
#pragma novector (ICC compiler)
```

```
#pragma nosimd (XLC compiler)
```

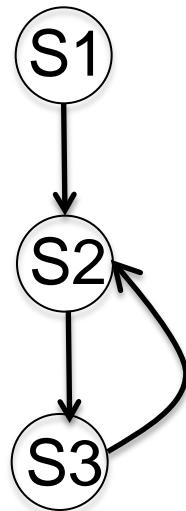


# Compiler Directives (II)

Vector code can run slower than scalar code

```
for (int i=1; i<LEN; i++) {  
S1  a[i] = b[i] + c[i];  
S2  d[i] = a[i] + e[i - 1];  
S3  e[i] = d[i] + c[i];  
}
```

Less locality when executing in vector mode



S1 can be vectorized  
S2 and S3 cannot be vectorized (as they are)

# Compiler Directives (II)

S116

```
#pragma novector  
for (int i=1; i<LEN; i++) {  
    a[i] = b[i] + c[i];  
    d[i] = a[i] + e[i-1];  
    e[i] = d[i] + c[i];  
}
```

S116

**Intel Nehalem**

**Compiler report:** Loop was partially vectorized

**Exec. Time scalar code:** 14.7

**Exec. Time vector code:** 18.1

**Speedup:** 0.8



# Loop Distribution

- It is also called loop fission.
- Divides loop control over different statements in the loop body.

```
for (i=1; i<LEN; i++) {  
    a[i] = (float)sqrt(b[i]) +  
           (float)sqrt(c[i]);  
    dummy(a, b, c);  
}
```

```
for (i=1; i<LEN; i++)  
    a[i] = (float)sqrt(b[i]) +  
           (float)sqrt(c[i]);  
  
for (i=1; i<LEN; i++)  
    dummy(a, b, c);
```

- Compiler cannot analyze the dummy function.  
As a result, the compiler cannot apply loop distribution, because it does not know if it is a legal transformation
- Programmer can apply loop distribution if legal.



# Loop Distribution

S126

```
for (i=1; i<LEN; i++) {  
    a[i] = (float)sqrt(b[i]) +  
           (float)sqrt(c[i]);  
    dummy(a, b, c);  
}
```

S126\_1

```
for (i=1; i<LEN; i++)  
    a[i] = (float)sqrt(b[i]) +  
           (float)sqrt(c[i]);  
for (i=1; i<LEN; i++)  
    dummy(a, b, c);
```

S126

**Intel Nehalem**  
**Compiler report:** Loop was not vectorized  
**Exec. Time scalar code:** 4.3  
**Exec. Time vector code:** --  
**Speedup:** --

S126\_1

**Intel Nehalem**  
**Compiler report:**

- Loop 1 was vectorized.
- Loop 2 was not vectorized

**Exec. Time scalar code:** 5.1  
**Exec. Time vector code:** 1.1  
**Speedup:** 4.6



# Loop Distribution

S126

```
for (i=1; i<LEN; i++) {  
    a[i] = (float)sqrt(b[i]) +  
           (float)sqrt(c[i]);  
    dummy(a, b, c);  
}
```

S126\_1

```
for (i=1; i<LEN; i++)  
    a[i] = (float)sqrt(b[i]) +  
           (float)sqrt(c[i]);  
for (i=1; i<LEN; i++)  
    dummy(a, b, c);
```

S126

**IBM Power 7**  
**Compiler report:** Loop was not SIMD vectorized  
**Exec. Time scalar code:** 1.3  
**Exec. Time vector code:** --  
**Speedup:** --

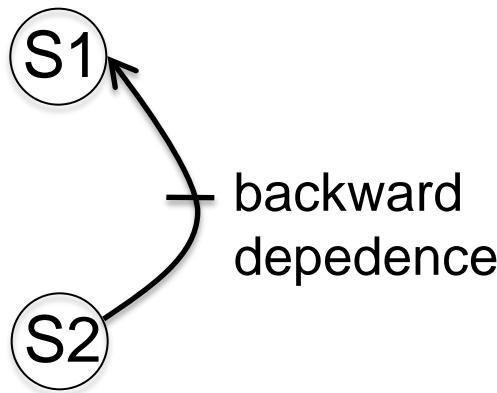
S126\_1

**IBM Power 7**  
**Compiler report:**  
- Loop 1 was SIMD vectorized.  
- Loop 2 was not SIMD vectorized  
**Exec. Time scalar code:** 1.14  
**Exec. Time vector code:** 1.0  
**Speedup:** 1.14

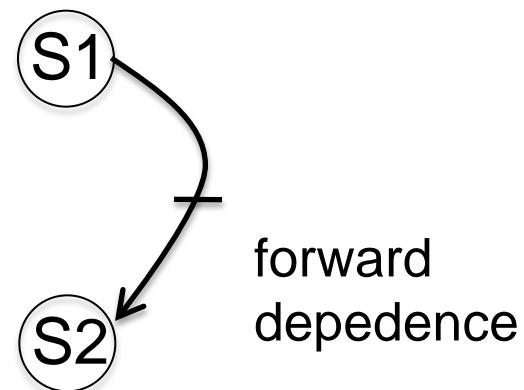


# Reordering Statements

```
for (i=0; i<LEN; i++) {  
S1 a[i] = b[i] + c[i];  
S2 d[i] = a[i+1]+(float)1.0;  
}
```



```
for (i=0; i<LEN; i++) {  
S1 d[i] = a[i+1]+(float)1.0;  
S2 a[i] = b[i] + c[i];  
}
```



# Reordering Statements

S114

```
for (i=0; i<LEN; i++) {  
    a[i] = b[i] + c[i];  
    d[i] = a[i+1]+(float)1.0;  
}
```

S114\_1

```
for (i=0; i<LEN; i++) {  
    d[i] = a[i+1]+(float)1.0;  
    a[i] = b[i] + c[i];  
}
```

S114

S114\_1

## Intel Nehalem

**Compiler report:** Loop was not vectorized. Existence of vector dependence

**Exec. Time scalar code:** 12.6

**Exec. Time vector code:** --

**Speedup:** --

## Intel Nehalem

**Compiler report:** Loop was vectorized.

**Exec. Time scalar code:** 10.7

**Exec. Time vector code:** 6.2

**Speedup:** 1.7





# Reordering Statements

S114

```
for (i=0; i<LEN; i++) {  
    a[i] = b[i] + c[i];  
    d[i] = a[i+1]+(float)1.0;  
}
```

S114\_1

```
for (i=0; i<LEN; i++) {  
    d[i] = a[i+1]+(float)1.0;  
    a[i] = b[i] + c[i];  
}
```

The IBM XLC compiler generated the same code in both cases

S114

**IBM Power 7**  
**Compiler report:** Loop was SIMD vectorized  
**Exec. Time scalar code:** 3.3  
**Exec. Time vector code:** 1.8  
**Speedup:** 1.8

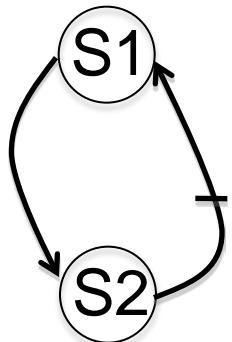
S114\_1

**IBM Power 7**  
**Compiler report:** Loop was SIMD vectorized  
**Exec. Time scalar code:** 3.3  
**Exec. Time vector code:** 1.8  
**Speedup:** 1.8

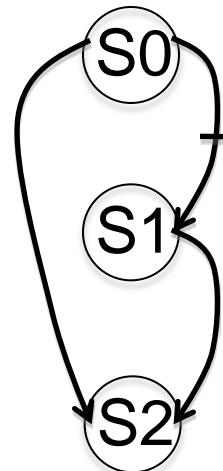


# Node Splitting

```
for (int i=0; i<LEN- 1; i++) {  
S1 a[i ]=b[i ]+c[i ];  
S2 d[i ]=(a[i ]+a[i +1]) *(float)0. 5;  
}
```



```
for (int i=0; i<LEN- 1; i++) {  
S0 temp[i ]=a[i +1];  
S1 a[i ]=b[i ]+c[i ];  
S2 d[i ]=(a[i ]+temp[i ]) *(float) 0. 5  
}
```



# Node Splitting

S126

```
for (int i=0; i<LEN- 1; i++) {  
    a[i ]=b[i ]+c[i ];  
    d[i ]=(a[i ]+a[i +1]) *(float)0. 5;  
}
```

S126\_1

```
for (int i=0; i<LEN- 1; i++) {  
    temp[i ]=a[i +1];  
    a[i ]=b[i ]+c[i ];  
    d[i ]=(a[i ]+temp[i ]) *(float)0. 5;  
}
```

S126

**Intel Nehalem**  
**Compiler report:** Loop was not vectorized. Existence of vector dependence  
**Exec. Time scalar code:** 12.6  
**Exec. Time vector code:** --  
**Speedup:** --

S126\_1

**Intel Nehalem**  
**Compiler report:** Loop was vectorized.  
**Exec. Time scalar code:** 13.2  
**Exec. Time vector code:** 9.7  
**Speedup:** 1.3



# Node Splitting

S126

```
for (int i=0; i<LEN- 1; i++) {  
S1 a[i]=b[i]+c[i];  
S2 d[i]=(a[i]+a[i+1])*(float)0.5;  
}
```

S126\_1

```
for (int i=0; i<LEN- 1; i++) {  
S0 temp[i]=a[i+1];  
S1 a[i]=b[i]+c[i];  
S2 d[i]=(a[i]+temp[i])*(float) 0.5  
}
```

S126

S126\_1

## IBM Power 7

**Compiler report:** Loop was SIMD vectorized

**Exec. Time scalar code:** 3.8

**Exec. Time vector code:** 1.7

**Speedup:** 2.2

## IBM Power 7

**Compiler report:** Loop was SIMD vectorized

**Exec. Time scalar code:** 5.1

**Exec. Time vector code:** 2.4

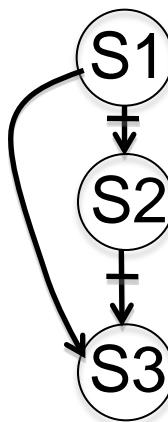
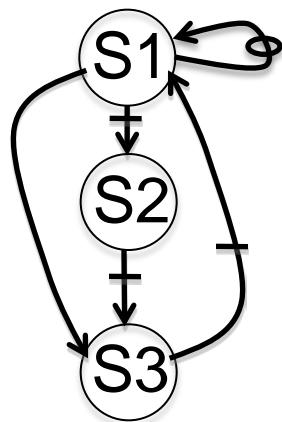
**Speedup:** 2.0



# Scalar Expansion

```
for (int i=0; i<n; i++) {  
S1  t = a[i];  
S2  a[i] = b[i];  
S3  b[i] = t;  
}
```

```
for (int i=0; i<n; i++) {  
S1  t[i] = a[i];  
S2  a[i] = b[i];  
S3  b[i] = t[i];  
}
```



# Scalar Expansion

S139

```
for (int i=0; i<n; i++) {  
    t = a[i];  
    a[i] = b[i];  
    b[i] = t;  
}
```

S139\_1

```
for (int i=0; i<n; i++) {  
    t[i] = a[i];  
    a[i] = b[i];  
    b[i] = t[i];  
}
```

S139

**Intel Nehalem**  
**Compiler report:** Loop was vectorized.  
**Exec. Time scalar code:** 0.7  
**Exec. Time vector code:** 0.4  
**Speedup:** 1.5

S139\_1

**Intel Nehalem**  
**Compiler report:** Loop was vectorized.  
**Exec. Time scalar code:** 0.7  
**Exec. Time vector code:** 0.4  
**Speedup:** 1.5



# Scalar Expansion

S139

```
for (int i=0; i<n; i++) {  
    t = a[i];  
    a[i] = b[i];  
    b[i] = t;  
}
```

S139\_1

```
for (int i=0; i<n; i++) {  
    t[i] = a[i];  
    a[i] = b[i];  
    b[i] = t[i];  
}
```

S139

S139\_1

## IBM Power 7

**Compiler report:** Loop was SIMD vectorized

**Exec. Time scalar code:** 0.28

**Exec. Time vector code:** 0.14

**Speedup:** 2

## IBM Power 7

**Compiler report:** Loop was SIMD vectorized

**Exec. Time scalar code:** 0.28

**Exec. Time vector code:** 0.14

**Speedup:** 2.0



# Loop Peeling

- Remove the first/s or the last/s iteration of the loop into separate code outside the loop
- It is always legal, provided that no additional iterations are introduced.
- When the trip count of the loop is not constant the peeled loop has to be protected with additional runtime tests.
- This transformation is useful to enforce a particular initial memory alignment on array references prior to loop vectorization.

```
for (i=0; i<LEN; i++) →  
    A[i] = B[i] + C[i];
```

```
A[0] = B[0] + C[0];  
for (i=1; i<LEN; i++)  
    A[i] = B[i] + C[i];
```



# Loop Peeling

- Remove the first/s or the last/s iteration of the loop into separate code outside the loop
- It is always legal, provided that no additional iterations are introduced.
- When the trip count of the loop is not constant the peeled loop has to be protected with additional runtime tests.
- This transformation is useful to enforce a particular initial memory alignment on array references prior to loop vectorization.

```
if (N>=1)  
    A[0] = B[0] + C[0];  
for (i=0; i<LEN; i++) → for (i=1; i<LEN; i++)  
    A[i] = B[i] + C[i];  
    A[0] = B[0] + C[0];
```

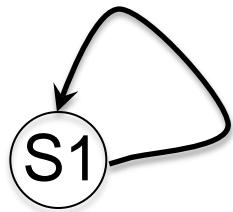


# Loop Peeling

```
S1 for (int i=0; i<LEN; i++) {  
    a[i] = a[i] + a[0];  
}
```

```
a[0]= a[0] + a[0];  
for (int i=1; i<LEN; i++) {  
    a[i] = a[i] + a[0]  
}
```

$$\begin{aligned} a[0] &= a[0] + a[0] \\ a[1] &= a[1] + a[0] \\ a[2] &= a[2] + a[0] \end{aligned}$$



Self true-dependence  
is not vectorized

After loop peeling, there are no dependences, and the loop can be vectorized

# Loop Peeling

S127

```
S1 for (int i=0; i<LEN; i++) {  
    a[i] = a[i] + a[0];  
}
```

S127\_1

```
a[0]= a[0] + a[0];  
for (int i=1; i<LEN; i++) {  
    a[i] = a[i] + a[0]  
}
```

S127

## Intel Nehalem

**Compiler report:** Loop was not vectorized. Existence of vector dependence

**Exec. Time scalar code:** 6.7  
**Exec. Time vector code:** --  
**Speedup:** --

S127\_1

## Intel Nehalem

**Compiler report:** Loop was vectorized.

**Exec. Time scalar code:** 6.6  
**Exec. Time vector code:** 1.2  
**Speedup:** 5.2



# Loop Peeling

S127

```
for (int i=0; i<LEN; i++)  
{  
    a[i] = a[i] + a[0];  
}
```

S127\_1

```
a[0]= a[0] + a[0];  
for (int i=1; i<LEN; i++)  
{  
    a[i] = a[i] + a[0];  
}
```

S127\_2

```
a[0]= a[0] + a[0];  
float t = a[0];  
for (int i=1; i<LEN; i++)  
{  
    a[i] = a[i] + t;  
}
```

S127

S127\_1

S127\_2

## IBM Power 7

**Compiler report:** Loop  
was not SIMD vectorized  
**Time scalar code:** 2.4  
**Time vector code:** --  
**Speedup:** --

## IBM Power 7

**Compiler report:** Loop  
was not SIMD vectorized  
**Exec. scalar code:** 2.4  
**Exec. vector code:** --  
**Speedup:** --

## IBM Power 7

**Compiler report:** Loop  
was vectorized  
**Exec. scalar code:** 1.58  
**Exec. vector code:** 0.62  
**Speedup:** 2.54



# Loop Interchanging

- This transformation switches the positions of one loop that is tightly nested within another loop.

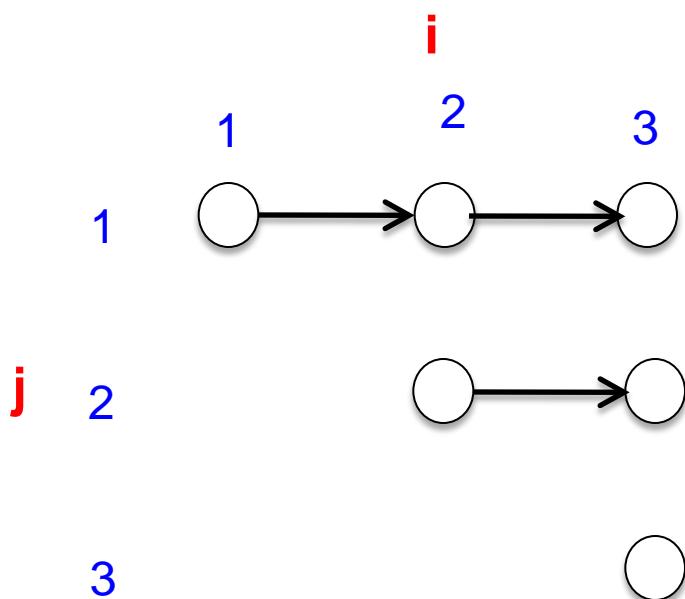
```
for (i=0; i<LEN; i++)  
  for (j=0; j<LEN; j++)  
    A[i][j]=0. 0;
```

```
for (j=0; j<LEN; j++)  
  for (i=0; i<LEN; i++)  
    A[i][j]=0. 0;
```



# Loop Interchanging

```
for (j=1; j<LEN; j++) {  
    for (i=j; i<LEN; i++) {  
        A[i][j]=A[i-1][j]+(float) 1.0;  
    } }
```



$j=1 \left\{ \begin{array}{l} i=1 A[1][1]=A[0][1] + 1 \\ i=2 A[2][1]=A[1][1] + 1 \\ i=3 A[3][1]=A[2][1] + 1 \end{array} \right.$

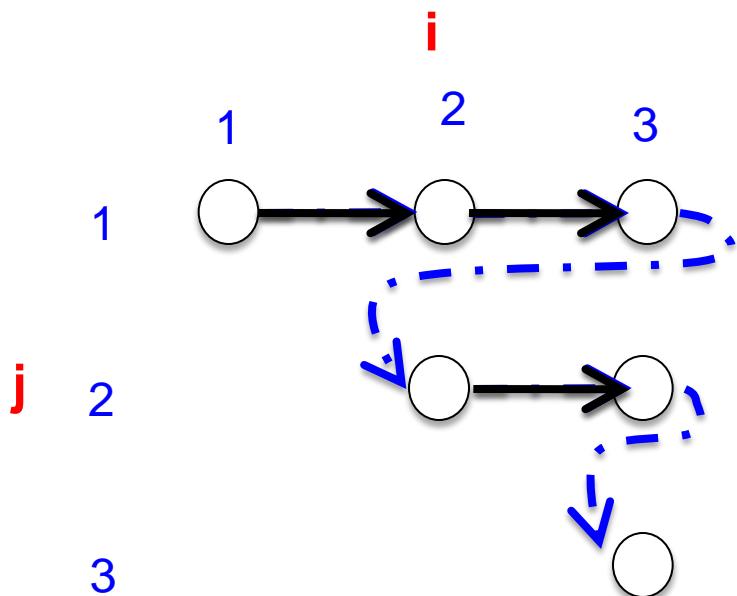
$j=2 \left\{ \begin{array}{l} i=2 A[2][2]=A[1][2] + 1 \\ i=3 A[3][2]=A[2][2] + 1 \end{array} \right.$

$j=3 \quad i=3 \quad A[3][3]=A[2][3] + 1$



# Loop Interchanging

```
for (j=1; j<LEN; j++) {  
    for (i=j; i<LEN; i++) {  
        A[i][j]=A[i-1][j]+(float) 1.0;  
    }  
}
```



$j=1 \left\{ \begin{array}{l} i=1 A[1][1]=A[0][1] + 1 \\ i=2 A[2][1]=A[1][1] + 1 \\ i=3 A[3][1]=A[2][1] + 1 \end{array} \right.$

$j=2 \left\{ \begin{array}{l} i=2 A[2][2]=A[1][2] + 1 \\ i=3 A[3][2]=A[2][2] + 1 \end{array} \right.$

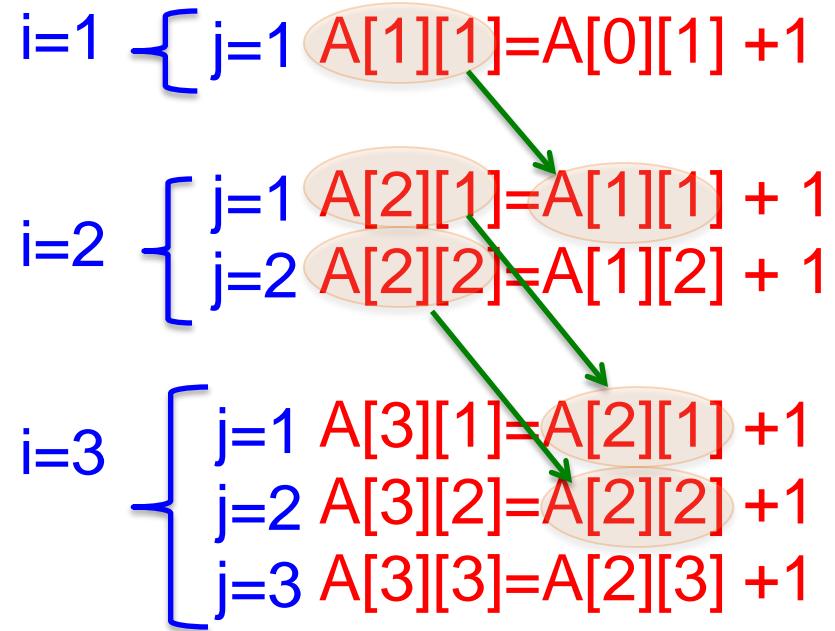
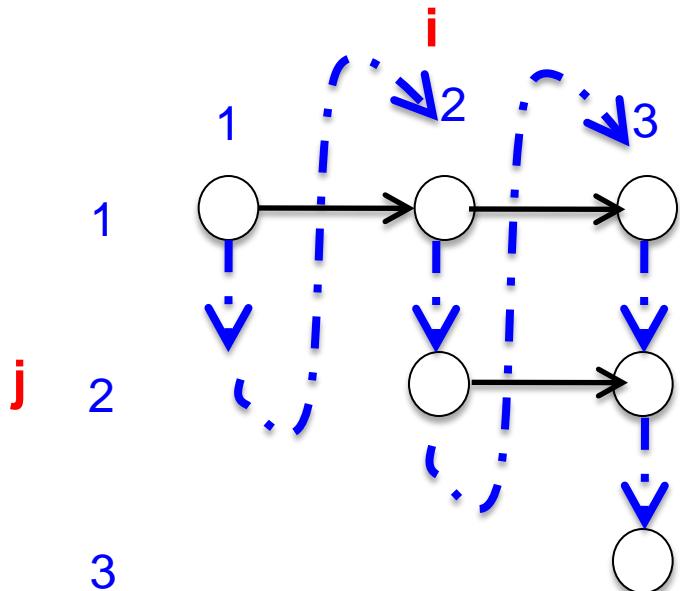
$j=3 \quad i=3 \quad A[3][3]=A[2][3] + 1$

Inner loop cannot be vectorized  
because of self-dependence



# Loop Interchanging

```
for (i=1; i<LEN; i++) {  
    for (j=1; j<i+1; j++) {  
        A[i][j]=A[i-1][j]+(float) 1.0;  
    }  
}
```



Loop interchange is legal  
No dependences in inner loop



# Loop Interchanging

S228

```
for (j=1; j<LEN; j++) {  
    for (i=j; i<LEN; i++) {  
        A[i][j]=A[i-1][j]+(float)1.0;  
    } }
```

S228\_1

```
for (i=1; i<LEN; i++) {  
    for (j=1; j<i+1; j++) {  
        A[i][j]=A[i-1][j]+(float)1.0;  
    } }
```

S228

**Intel Nehalem**  
**Compiler report:** Loop was not vectorized.  
**Exec. Time scalar code:** 2.3  
**Exec. Time vector code:** --  
**Speedup:** --

S228\_1

**Intel Nehalem**  
**Compiler report:** Loop was vectorized.  
**Exec. Time scalar code:** 0.6  
**Exec. Time vector code:** 0.2  
**Speedup:** 3



# Loop Interchanging

S228

```
for (j=1; j<LEN; j++) {  
    for (i=j; i<LEN; i++) {  
        A[i][j]=A[i-1][j]+(float)1.0;  
    } }
```

S228\_1

```
for (i=1; i<LEN; i++) {  
    for (j=1; j<i+1; j++) {  
        A[i][j]=A[i-1][j]+(float)1.0;  
    } }
```

S228

S228\_1

## IBM Power 7

**Compiler report:** Loop was not SIMD vectorized  
**Exec. Time scalar code:** 0.5  
**Exec. Time vector code:** --  
**Speedup:** --

## IBM Power 7

**Compiler report:** Loop was SIMD vectorized  
**Exec. Time scalar code:** 0.2  
**Exec. Time vector code:** 0.14  
**Speedup:** 1.42



# Outline

1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
  - Data Dependences
    - Reductions
  - Data Alignment
  - Aliasing
  - Non-unit strides
  - Conditional Statements
4. Vectorization using intrinsics

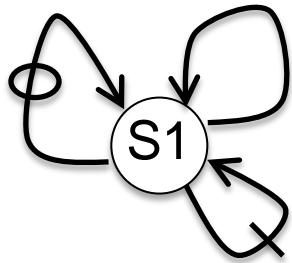


# Reductions

- Reduction is an operation, such as addition, which is applied to the elements of an array to produce a result of a lesser rank.

## Sum Reduction

```
sum =0;  
for (int i=0; i<LEN; ++i){  
    sum+= a[i];  
}
```



## Max Loc Reduction

```
x = a[0];  
index = 0;  
for (int i=0; i<LEN; ++i){  
    if (a[i] > x) {  
        x = a[i];  
        index = i;  
    } }
```

# Reductions

S131

```
sum =0;  
for (int i=0; i<LEN; ++i) {  
    sum+= a[i];  
}
```

S132

```
x = a[0];  
index = 0;  
for (int i=0; i<LEN; ++i) {  
    if (a[i] > x) {  
        x = a[i];  
        index = i;  
    } }
```

S131

**Intel Nehalem**  
**Compiler report:** Loop was vectorized.  
**Exec. Time scalar code:** 5.2  
**Exec. Time vector code:** 1.2  
**Speedup:** 4.1

S132

**Intel Nehalem**  
**Compiler report:** Loop was vectorized.  
**Exec. Time scalar code:** 9.6  
**Exec. Time vector code:** 2.4  
**Speedup:** 3.9



# Reductions

S131

```
sum =0;  
for (int i=0; i<LEN; ++i) {  
    sum+= a[i];  
}
```

S132

```
x = a[0];  
index = 0;  
for (int i=0; i<LEN; ++i) {  
    if (a[i] > x) {  
        x = a[i];  
        index = i;  
    } }
```

S131

**IBM Power 7**  
**Compiler report:** Loop was SIMD vectorized  
**Exec. Time scalar code:** 1.1  
**Exec. Time vector code:** 0.4  
**Speedup:** 2.4

S132

**IBM Power 7**  
**Compiler report:** Loop was not SIMD vectorized  
**Exec. Time scalar code:** 4.4  
**Exec. Time vector code:** --  
**Speedup:** --



# Reductions

S141\_1

```
for (int i = 0; i < 64; i++) {  
    max[i] = a[i];  
    loc[i] = i; }  
for (int i = 0; i < LEN; i+=64) {  
    for (int j=0, k=i; k<i+64;  
k++,j++) {  
        int cmp = max[j] < a[k];  
        max[j] = cmp ? a[k] : max[j];  
        loc[j] = cmp ? k : loc[j];  
    } }  
MAX = max[0];  
LOC = 0;  
for (int i = 0; i < 64; i++) {  
    if (MAX < max[i]) {  
        MAX = max[i];  
        LOC = loc[i];  
    } }
```

S141\_1

**IBM Power 7**  
**Compiler report:** Loop was SIMD  
vectorized  
**Exec. Time scalar code:** 10.2  
**Exec. Time vector code:** 2.7  
**Speedup:** 3.7

S141\_2

**IBM Power 7**  
**A version written with intrinsics**  
**runs in 1.6 secs.**



# Outline

1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
  - Data Dependences
    - Induction variables
  - Data Alignment
  - Aliasing
  - Non-unit strides
  - Conditional Statements
4. Vectorization with intrinsics

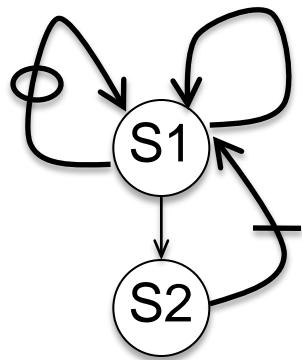


# Induction variables

- Induction variable is a variable that can be expressed as a function of the loop iteration variable

```
float s = (float)0.0;  
for (int i=0; i<LEN; i++) {  
    s += (float)2.;  
    a[i] = s * b[i];  
}
```

```
for (int i=0; i<LEN; i++) {  
    a[i] = (float)2. * (i+1) * b[i];  
}
```



# Induction variables

S133

```
float s = (float)0.0;  
for (int i=0; i<LEN; i++) {  
    s += (float)2.;  
    a[i] = s * b[i];  
}
```

S133\_1

```
for (int i=0; i<LEN; i++) {  
    a[i] = (float)2.*(i+1)*b[i];  
}
```

The Intel ICC compiler generated the same vector code in both cases

S133

**Intel Nehalem**  
**Compiler report:** Loop was  
vectorized.  
**Exec. Time scalar code:** 6.1  
**Exec. Time vector code:** 1.9  
**Speedup:** 3.1

S133\_1

**Intel Nehalem**  
**Compiler report:** Loop was  
vectorized.  
**Exec. Time scalar code:** 8.4  
**Exec. Time vector code:** 1.9  
**Speedup:** 4.2



# Induction variables

S133

```
float s = (float)0.0;  
for (int i=0; i<LEN; i++) {  
    s += (float)2.;  
    a[i] = s * b[i];  
}
```

S133\_1

```
for (int i=0; i<LEN; i++) {  
    a[i] = (float)2.*(i+1)*b[i];  
}
```

S133

S133\_1

## IBM Power 7

**Compiler report:** Loop was not SIMD vectorized  
**Exec. Time scalar code:** 2.7  
**Exec. Time vector code:** --  
**Speedup:** --

## IBM Power 7

**Compiler report:** Loop was SIMD vectorized  
**Exec. Time scalar code:** 3.7  
**Exec. Time vector code:** 1.4  
**Speedup:** 2.6



# Induction Variables

- Coding style matters:

```
for (int i=0; i<LEN; i++) {  
    *a = *b + *c;  
    a++; b++; c++;  
}
```

```
for (int i=0; i<LEN; i++) {  
    a[i] = b[i] + c[i];  
}
```

These codes are equivalent, but ...



# Induction Variables

S134

```
for (int i=0; i<LEN; i++) {  
    *a = *b + *c;  
    a++; b++; c++;  
}
```

S134\_1

```
for (int i=0; i<LEN; i++) {  
    a[i] = b[i] + c[i];  
}
```

S134

**Intel Nehalem**  
**Compiler report:** Loop was not vectorized.  
**Exec. Time scalar code:** 5.5  
**Exec. Time vector code:** --  
**Speedup:** --

S134\_1

**Intel Nehalem**  
**Compiler report:** Loop was vectorized.  
**Exec. Time scalar code:** 6.1  
**Exec. Time vector code:** 3.2  
**Speedup:** 1.8



# Induction Variables

S134

```
for (int i=0; i<LEN; i++) {  
    *a = *b + *c;  
    a++; b++; c++;  
}
```

S134\_1

```
for (int i=0; i<LEN; i++) {  
    a[i] = b[i] + c[i];  
}
```

The IBM XLC compiler generated the same code in both cases

S134

**IBM Power 7**  
**Compiler report:** Loop was SIMD  
vectorized  
**Exec. Time scalar code:** 2.2  
**Exec. Time vector code:** 1.0  
**Speedup:** 2.2

S134\_1

**IBM Power 7**  
**Compiler report:** Loop was SIMD  
vectorized  
**Exec. Time scalar code:** 2.2  
**Exec. Time vector code:** 1.0  
**Speedup:** 2.2



# Outline

1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
  - Data Dependences
  - **Data Alignment**
  - Aliasing
  - Non-unit strides
  - Conditional Statements
4. Vectorization with intrinsics



# Data Alignment

- Vector loads/stores load/store 128 consecutive bits to a vector register.
- Data addresses need to be 16-byte (128 bits) aligned to be loaded/stored
  - Intel platforms support aligned and unaligned load/stores
  - IBM platforms do not support unaligned load/stores

Is  $\&b[0]$  16-byte aligned?

```
void test1(float *a, float *b, float *c)
{
    for (int i=0; i<LEN; i++) {
        a[i] = b[i] + c[i];
    }
}
```

vector load loads  $b[0] \dots b[3]$



# Data Alignment

- To know if a pointer is 16-byte aligned, the last digit of the pointer address in hex must be 0.
- Note that if &b[0] is 16-byte aligned, and is a single precision array, then &b[4] is also 16-byte aligned

```
attribute ((aligned(16))) float B[1024];
```

```
int main() {
    printf("%p, %p\n", &B[0], &B[4]);
}
```

Output:

0x7fff1e9d8580, 0x7fff1e9d8590



# Data Alignment

- In many cases, the compiler cannot statically know the alignment of the address in a pointer
- The compiler assumes that the base address of the pointer is 16-byte aligned and adds a run-time checks for it
  - if the runtime check is false, then it uses another code (which may be scalar)



# Data Alignment

- Manual 16-byte alignment can be achieved by forcing the base address to be a multiple of 16.

```
__attribute__((aligned(16))) float b[N];
float* a = (float*) memalign(16, N*sizeof(float));
```

- When the pointer is passed to a function, the compiler should be aware of where the 16-byte aligned address of the array starts.

```
void func1(float *a, float *b,
          float *c) {
    __assume_aligned(a, 16);
    __assume_aligned(b, 16);
    __assume_aligned(c, 16);
    for int (i=0; i<LEN; i++) {
        a[i] = b[i] + c[i];
    }
```



# Data Alignment - Example

```
float A[N] __attribute__((aligned(16)));
float B[N] __attribute__((aligned(16)));
float C[N] __attribute__((aligned(16)));

void test(){
    for (int i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
    }
}
```



# Data Alignment - Example

```
float A[N] __attribute__((aligned(16)));
float B[N] __attribute__((aligned(16)));
float C[N] __attribute__((aligned(16)));


void test1() {
    _m128 rA, rB, rC;
    for (int i = 0; i < N; i+=4) {
        rA = _mm_load_ps(&A[i]);
        rB = _mm_load_ps(&B[i]);
        rC = _mm_add_ps(rA, rB);
        _mm_store_ps(&C[i], rC);
    }
}
```

```
void test3() {
    _m128 rA, rB, rC;
    for (int i = 1; i < N-3; i+=4) {
        rA = _mm_loadu_ps(&A[i]);
        rB = _mm_loadu_ps(&B[i]);
        rC = _mm_add_ps(rA, rB);
        _mm_storeu_ps(&C[i], rC);
    }
}
```

```
void test2() {
    _m128 rA, rB, rC;
    for (int i = 0; i < N; i+=4) {
        rA = _mm_loadu_ps(&A[i]);
        rB = _mm_loadu_ps(&B[i]);
        rC = _mm_add_ps(rA, rB);
        _mm_storeu_ps(&C[i], rC);
    }
}
```

Nanosecond per iteration

|                        | Core 2 Duo | Intel i7 | Power 7 |
|------------------------|------------|----------|---------|
| Aligned                | 0.577      | 0.580    | 0.156   |
| Aligned (unaligned Id) | 0.689      | 0.581    | 0.241   |
| Unaligned              | 2.176      | 0.629    | 0.243   |



# Alignment in a struct

```
struct st{  
    char A;  
    int B[64];  
    float C;  
    int D[64];  
};  
  
int main(){  
    st s1;  
    printf("%p, %p, %p, %p\n", &s1.A, s1.B, &s1.C, s1.D); } 
```

Output:

0x7ffe6765f00, 0x7ffe6765f04, 0x7ffe6766004, 0x7ffe6766008

- Arrays B and D are not 16-bytes aligned (see the address)



# Alignment in a struct

```
struct st{  
    char A;  
    int B[64] __attribute__((aligned(16)));  
    float C;  
    int D[64] __attribute__((aligned(16)));  
};  
  
int main(){  
    st s1;  
    printf("%p, %p, %p, %p\n", &s1.A, s1.B, &s1.C, s1.D); } 
```

Output:

0x7fff1e9d8580, 0x7fff1e9d8590, 0x7fff1e9d8690, 0x7fff1e9d86a0

- Arrays A and B are aligned to 16-bytes (notice the 0 in the 4 least significant bits of the address)
- Compiler automatically does padding



# Outline

1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
  - Data Dependences
  - Data Alignment
  - **Aliasing**
  - Non-unit strides
  - Conditional Statements
4. Vectorization with intrinsics



# Aliasing

- Can the compiler vectorize this loop?

```
void func1(float *a, float *b, float *c) {  
    for (int i = 0; i < LEN; i++) {  
        a[i] = b[i] + c[i];  
    }  
}
```





# Aliasing

- Can the compiler vectorize this loop?

```
float* a = &b[1];  
...  
void func1(float *a, float *b, float *c)  
{  
    for (int i = 0; i < LEN; i++)  
        a[i] = b[i] + c[i];  
}
```

$$b[1] = b[0] + c[0]$$

$$b[2] = b[1] + c[1]$$



# Aliasing

- Can the compiler vectorize this loop?

```
float* a = &b[1];
```

...

```
void func1(float *a, float *b, float *c)
{
    for (int i = 0; i < LEN; i++)
        a[i] = b[i] + c[i];
}
```

a and b are aliasing  
There is a self-true dependence  
Vectorizing this loop would  
be illegal



# Aliasing

- To vectorize, the compiler needs to guarantee that the pointers are not aliased.
- When the compiler does not know if two pointer are alias, it still vectorizes, but needs to add up-to  $O(n^2)$  run-time checks, where  $n$  is the number of pointers

When the number of pointers is large, the compiler may decide to not vectorize

```
void func1(float *a, float *b, float *c) {  
    for (int i=0; i<LEN; i++)  
        a[i] = b[i] + c[i];  
}
```





# Aliasing

- Two solutions can be used to avoid the run-time checks
  1. static and global arrays
  2. `__restrict__` attribute





# Aliasing

## 1. Static and Global arrays

```
__attribute__((aligned(16))) float a[LEN];
__attribute__((aligned(16))) float b[LEN];
__attribute__((aligned(16))) float c[LEN];
```

```
void func1(){
    for (int i=0; i<LEN; i++)
        a[i] = b[i] + c[i];
}
```

```
int main() {
    ...
    func1();
}
```



# Aliasing

## 1. `__restrict__` keyword

```
void func1(float* __restrict__ a, float* __restrict__ b,
float* __restrict__ c) {
    __assume_aligned(a, 16);
    __assume_aligned(b, 16);
    __assume_aligned(c, 16);
    for int (i=0; i<LEN; i++)
        a[i] = b[i] + c[i];
}
int main() {
    float* a=(float*) malloc(16, LEN*sizeof(float));
    float* b=(float*) malloc(16, LEN*sizeof(float));
    float* c=(float*) malloc(16, LEN*sizeof(float));
    ...
    func1(a, b, c);
}
```



# Aliasing – Multidimensional arrays

- Example with 2D arrays: pointer-to-pointer declaration.

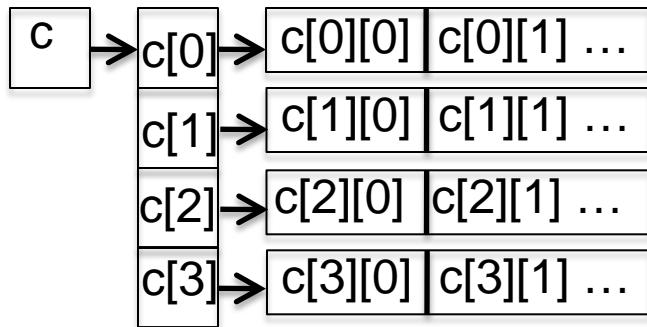
```
void func1(float** __restrict__ a, float**  
          __restrict__ b, float** __restrict__ c) {  
    for (int i=0; i<LEN; i++)  
        for (int j=1; j<LEN; j++)  
            a[i][j] = b[i][j - 1] * c[i][j];  
}
```



# Aliasing – Multidimensional arrays

- Example with 2D arrays: pointer-to-pointer declaration.

```
void func1(float** __restrict__ a, float** __restrict__  
          b, float** __restrict__ c) {  
    for (int i=0; i<LEN; i++)  
        for (int j=1; j<LEN; j++)  
            a[i][j] = b[i][j - 1] * c[i][j];  
}
```



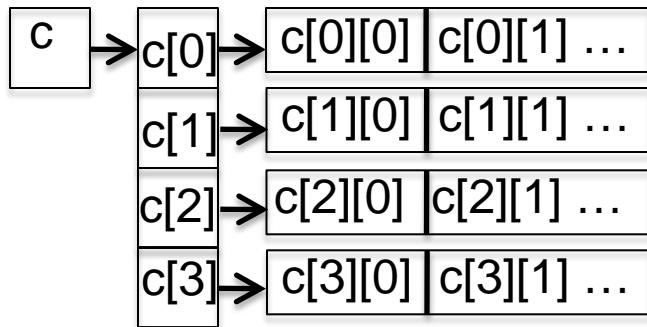
`__restrict__` only qualifies  
the first dereferencing of *c*;

Nothing is said about the  
arrays that can be accessed  
through *c[i]*

# Aliasing – Multidimensional arrays

- Example with 2D arrays: pointer-to-pointer declaration.

```
void func1(float** __restrict__ a, float** __restrict__  
          b, float** __restrict__ c) {  
    for (int i=0; i<LEN; i++)  
        for (int j=1; j<LEN; j++)  
            a[i][j] = b[i][j - 1] * c[i][j];  
}
```



`__restrict__` only qualifies  
the first dereferencing of `c`;

Nothing is said about the  
arrays that can be accessed  
through `c[i]`

Intel ICC compiler, version 11.1 will vectorize this code.

Previous versions of the Intel compiler or compilers from  
other vendors, such as IBM XLC, will not vectorize it.



# Aliasing – Multidemensional Arrays

- Three solutions when `__restrict__` does not enable vectorization
  - 1. Static and global arrays
  - 2. Linearize the arrays and use `__restrict__` keyword
  - 3. Use compiler directives



# Aliasing – Multidimensional arrays

## 1. Static and Global declaration

```
__attribute__((aligned(16))) float a[N][N];
void t() {
    a[i][j]...
}

int main() {
    ...
    t();
}
```



# Aliasing – Multidimensional arrays

## 2. Linearize the arrays

```
void t(float* __restrict__ A) {
    //Access to Element A[i][j] is now A[i * 128 + j]
    ...
}
```

```
int main() {
    float* A = (float*) memalign(16, 128 * 128 * sizeof(float));
    ...
    t(A);
}
```



# Aliasing – Multidimensional arrays

## 3. Use compiler directives:

```
#pragma ivdep (Intel ICC)
#pragma disjoint (IBM XLC)
```

```
void func1(float **a, float **b, float **c) {
    for (int i=0; i<m; i++) {
        #pragma ivdep
        for (int j=0; j<LEN; j++)
            c[i][j] = b[i][j] * a[i][j];
    }
}
```



# Outline

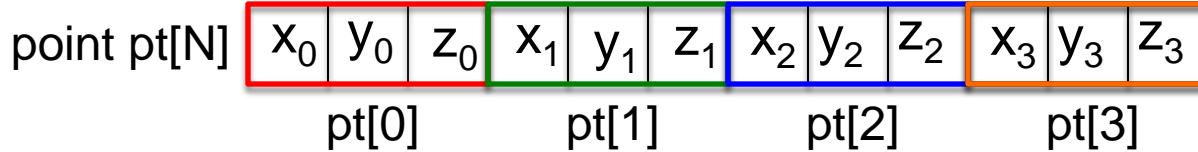
1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
  - Data Dependences
  - Data Alignment
  - Aliasing
  - **Non-unit strides**
  - Conditional Statements
4. Vectorization with intrinsics



# Non-unit Stride – Example I

- Array of a struct

```
typedef struct{int x, y, z}  
point;  
point pt[LEN];  
  
for (int i=0; i<LEN; i++) {  
    pt[i].y *= scale;  
}
```

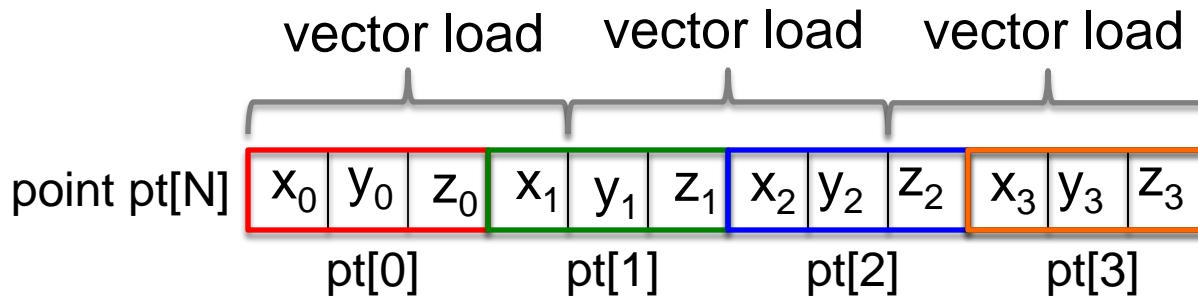


# Non-unit Stride – Example I

- Array of a struct

```
typedef struct{int x, y, z}  
point;  
point pt[LEN];
```

```
for (int i=0; i<LEN; i++) {  
    pt[i].y *= scale;  
}
```

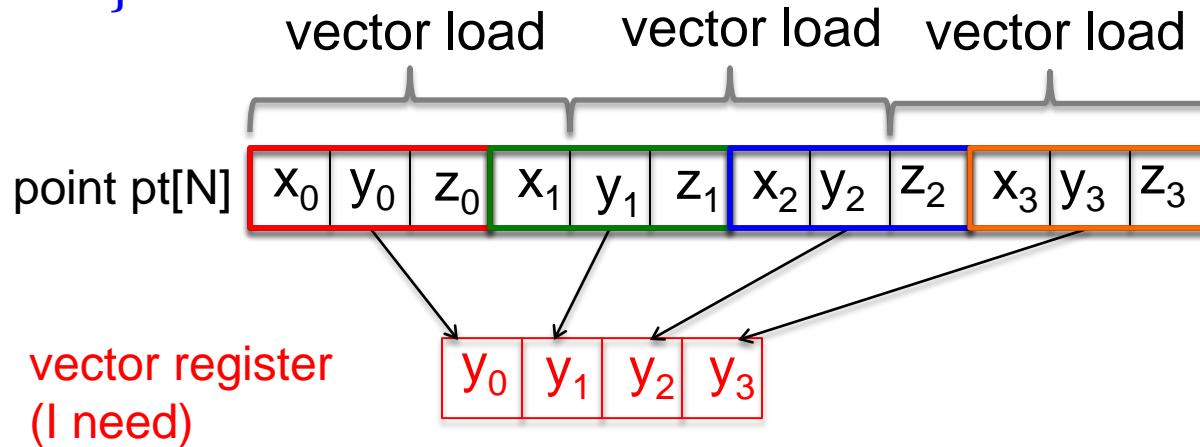


# Non-unit Stride – Example I

- Array of a struct

```
typedef struct{int x, y, z}  
point;  
point pt[LEN];
```

```
for (int i=0; i<LEN; i++) {  
    pt[i].y *= scale;  
}
```

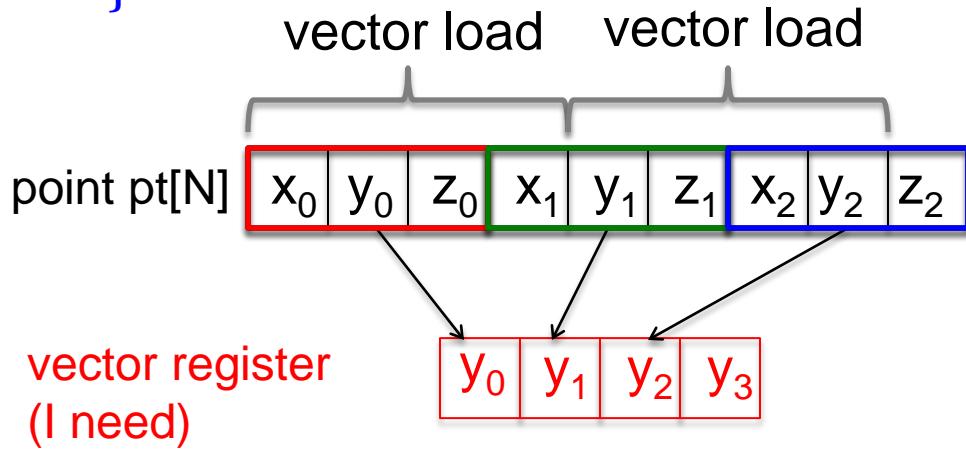


# Non-unit Stride – Example I

- Array of a struct

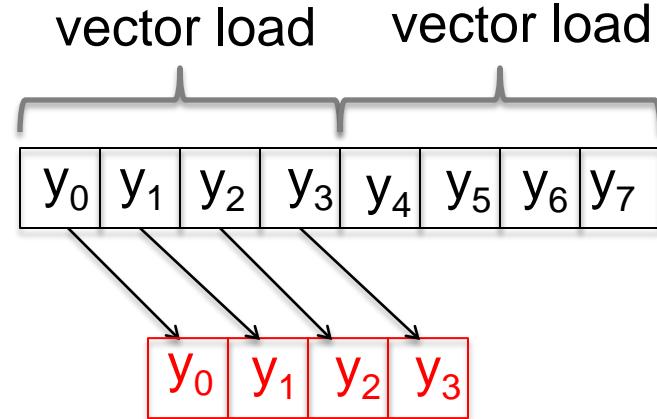
```
typedef struct{int x, y, z}  
point;  
point pt[LEN];
```

```
for (int i=0; i<LEN; i++) {  
    pt[i].y *= scale;  
}
```



- Arrays

```
int ptx[LEN], int pty[LEN],  
int ptz[LEN];  
  
for (int i=0; i<LEN; i++) {  
    pty[i] *= scale;  
}
```



# Non-unit Stride – Example I

S135

```
typedef struct{int x, y, z}  
point;  
point pt[LEN];  
  
for (int i=0; i<LEN; i++) {  
    pt[i].y *= scale;  
}
```

S135

S135\_1

```
int ptx[LEN], int pty[LEN],  
int ptz[LEN];  
  
for (int i=0; i<LEN; i++) {  
    pty[i] *= scale;  
}
```

S135\_1

## Intel Nehalem

**Compiler report:** Loop was not vectorized. Vectorization possible but seems inefficient

**Exec. Time scalar code:** 6.8  
**Exec. Time vector code:** --  
**Speedup:** --



## Intel Nehalem

**Compiler report:** Loop was vectorized.

**Exec. Time scalar code:** 4.8  
**Exec. Time vector code:** 1.3  
**Speedup:** 3.7

# Non-unit Stride – Example I

S135

```
typedef struct{int x, y, z}  
point;  
point pt[LEN];  
  
for (int i=0; i<LEN; i++) {  
    pt[i].y *= scale;  
}
```

S135

S135\_1

```
int ptx[LEN], int pty[LEN],  
int ptz[LEN];  
  
for (int i=0; i<LEN; i++) {  
    pty[i] *= scale;  
}
```

S135\_1

## IBM Power 7

**Compiler report:** Loop was not SIMD vectorized because it is not profitable to vectorize

**Exec. Time scalar code:** 2.0

**Exec. Time vector code:** --

**Speedup:** --

## IBM Power 7

**Compiler report:** Loop was SIMD vectorized

**Exec. Time scalar code:** 1.8

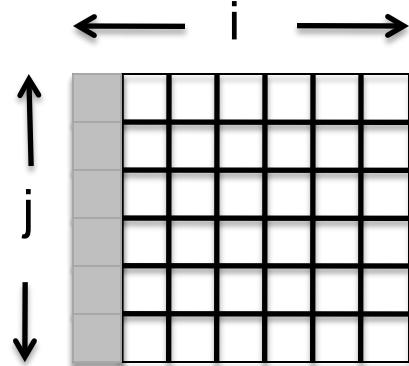
**Exec. Time vector code:** 1.5

**Speedup:** 1.2



# Non-unit Stride – Example II

```
for (int i=0; i<LEN; i++) {  
    sum = 0;  
    for (int j=0; j<LEN; j++) {  
        sum += A[j][i];  
    }  
    B[i] = sum;  
}
```



```
for (int i=0; i<size; i++) {  
    sum[i] = 0;  
    for (int j=0; j<size; j++) {  
        sum[i] += A[j][i];  
    }  
    B[i] = sum[i];  
}
```

# Non-unit Stride – Example II

S136

```
for (int i=0; i<LEN; i++) {  
    sum = (float) 0. 0;  
    for (int j=0; j<LEN; j++) {  
        sum += A[j][i];  
    }  
    B[i] = sum;  
}
```

S136\_1

```
for (int i=0; i<LEN; i++)  
    sum[i] = (float) 0. 0;  
    for (int j=0; j<LEN; j++) {  
        sum[i] += A[j][i];  
    }  
    B[i]=sum[i];  
}
```

S136\_2

```
for (int i=0; i<LEN; i++)  
    B[i] = (float) 0. 0;  
    for (int j=0; j<LEN; j++) {  
        B[i] += A[j][i];  
    }  
}
```

S136

**Intel Nehalem**

**Compiler report:** Loop was not vectorized. Vectorization possible but seems inefficient

**Exec. Time scalar code:** 3.7

**Exec. Time vector code:** --

**Speedup:** --

S136\_1

**Intel Nehalem**  
**report:** Permuted loop was vectorized.

**scalar code:** 1.6

**vector code:** 0.6

**Speedup:** 2.6

S136\_2

**Intel Nehalem**  
**report:** Permuted loop was vectorized.

**scalar code:** 1.6

**vector code:** 0.6

**Speedup:** 2.6



# Non-unit Stride – Example II

S136

```
for (int i=0; i<LEN; i++) {  
    sum = (float) 0. 0;  
    for (int j=0; j<LEN; j++) {  
        sum += A[j][i];  
    }  
    B[i] = sum;  
}
```

S136\_1

```
for (int i=0; i<LEN; i++)  
    sum[i] = (float) 0. 0;  
    for (int j=0; j<LEN; j++) {  
        sum[i] += A[j][i];  
    }  
    B[i]=sum[i];  
}
```

S136\_2

```
for (int i=0; i<LEN; i++)  
    B[i] = (float) 0. 0;  
    for (int j=0; j<LEN; j++) {  
        B[i] += A[j][i];  
    }  
}
```

S136

**IBM Power 7**  
**Compiler report:** Loop was not SIMD vectorized  
**Exec. Time scalar code:** 2.0  
**Exec. Time vector code:** --  
**Speedup:** --

S136\_1

**IBM Power 7**  
**report:** Loop interchanging applied.  
Loop was SIMD vectorized  
**scalar code:** 0.4  
**vector code:** 0.2  
**Speedup:** 2.0

S136\_2

**IBM Power 7**  
**report:** Loop interchanging applied.  
Loop was SIMD  
**scalar code:** 0.4  
**vector code:** 0.16  
**Speedup:** 2.7



# Outline

1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
  - Data Dependences
  - Data Alignment
  - Aliasing
  - Non-unit strides
  - Conditional Statements
4. Vectorization with intrinsics





# Conditional Statements – I

- Loops with conditions need `#pragma vector always`
  - Since the compiler does not know if vectorization will be profitable
  - The condition may prevent from an exception

```
#pragma vector always
for (int i = 0; i < LEN; i++) {
    if (c[i] < (float) 0.0)
        a[i] = a[i] * b[i] + d[i];
}
```



# Conditional Statements – I

S137

```
for (int i = 0; i < LEN; i++) {  
    if (c[i] < (float) 0.0)  
        a[i] = a[i] * b[i] + d[i];  
}
```

S137\_1

```
#pragma vector always  
for (int i = 0; i < LEN; i++) {  
    if (c[i] < (float) 0.0)  
        a[i] = a[i] * b[i] + d[i];  
}
```

S137

**Intel Nehalem**  
**Compiler report:** Loop was not vectorized. Condition may protect exception  
**Exec. Time scalar code:** 10.4  
**Exec. Time vector code:** --  
**Speedup:** --

S137\_1

**Intel Nehalem**  
**Compiler report:** Loop was vectorized.  
**Exec. Time scalar code:** 10.4  
**Exec. Time vector code:** 5.0  
**Speedup:** 2.0



# Conditional Statements – I

S137

```
for (int i = 0; i < LEN; i++) {  
    if (c[i] < (float) 0.0)  
        a[i] = a[i] * b[i] + d[i];  
}
```

S137\_1

```
for (int i = 0; i < LEN; i++) {  
    if (c[i] < (float) 0.0)  
        a[i] = a[i] * b[i] + d[i];  
}
```

compiled with flag -qdebug=alwayspec

S137

**IBM Power 7**  
**Compiler report:** Loop was SIMD vectorized  
**Exec. Time scalar code:** 4.0  
**Exec. Time vector code:** 1.5  
**Speedup:** 2.5

S137\_1

**IBM Power 7**  
**Compiler report:** Loop was SIMD vectorized  
**Exec. Time scalar code:** 4.0  
**Exec. Time vector code:** 1.5  
**Speedup:** 2.5





# Conditional Statements

- Compiler removes *if conditions* when generating vector code

```
for (int i = 0; i < LEN; i++) {  
    if (c[i] < (float) 0.0)  
        a[i] = a[i] * b[i] + d[i];  
}
```



# Conditional Statements

```
for (int i=0; i<1024; i++) {  
    if (c[i] < (float) 0.0)  
        a[i]=a[i]*b[i]+d[i];  
}
```

|       |       |      |       |      |
|-------|-------|------|-------|------|
| rC    | 2     | -1   | 1     | -2   |
| rCmp  | False | True | False | True |
| rThen | 0     | 3.2  | 0     | 3.2  |
| rElse | 1.    | 0    | 1.    | 0    |
| rS    | 1.    | 3.2  | 1.    | 3.2  |

```
vector bool char = rCmp  
vector float r0={0., 0., 0., 0.};  
vector float rA, rB, rC, rD, rS, rT,  
rThen, rElse;  
for (int i=0; i<1024; i+=4){  
    // load rA, rB, and rD;  
    rCmp = vec_cmplt(rC, r0);  
    rT= rA*rB+rD;  
    rThen = vec_and(rT, rCmp);  
    rElse = vec_andc(rA, rCmp);  
    rS = vec_or(rthen, relse);  
    //store rS  
}
```



# Conditional Statements

```
for (int i=0; i<1024; i++) {  
    if (c[i] < (float) 0.0)  
        a[i]=a[i]*b[i]+d[i];  
}
```

Speedups will depend on the values on c[i]

Compiler tends to be conservative, as the condition may prevent from segmentation faults

```
vector bool char = rCmp  
vector float r0={0., 0., 0., 0.};  
vector float rA, rB, rC, rD, rS, rT,  
rThen, rElse;  
for (int i=0; i<1024; i+=4){  
    // load rA, rB, and rD;  
    rCmp = vec_cmplt(rC, r0);  
    rT= rA*rB+rD;  
    rThen = vec_and(rT, rCmp);  
    rElse = vec_andc(rA, rCmp);  
    rS = vec_or(rthen, relse);  
    //store rS  
}
```



# Compiler Directives

- Compiler vectorizes many loops, but many more can be vectorized if the appropriate directives are used

| Compiler Hints for Intel ICC                   | Semantics                               |
|------------------------------------------------|-----------------------------------------|
| #pragma ivdep                                  | Ignore assume data dependences          |
| #pragma vector always                          | override efficiency heuristics          |
| #pragma novector                               | disable vectorization                   |
| <code>__restrict__</code>                      | assert exclusive access through pointer |
| <code>__attribute__((aligned(int-val)))</code> | request memory alignment                |
| <code>memalign(int-val,size);</code>           | malloc aligned memory                   |
| <code>__assume_aligned(exp, int-val)</code>    | assert alignment property               |



# Compiler Directives

- Compiler vectorizes many loops, but many more can be vectorized if the appropriate directives are used

| Compiler Hints for IBM XLC         | Semantics                               |
|------------------------------------|-----------------------------------------|
| #pragma ibm independent_loop       | Ignore assumed data dependences         |
| #pragma nosimd                     | disable vectorization                   |
| __restrict__                       | assert exclusive access through pointer |
| __attribute__ ((aligned(int-val))) | request memory alignment                |
| memalign(int-val,size);            | malloc aligned memory                   |
| __alignx (int-val, exp)            | assert alignment property               |



# Outline

1. Intro
2. Data Dependences (Definition)
3. Overcoming limitations to SIMD-Vectorization
  - Data Dependences
  - Data Alignment
  - Aliasing
  - Non-unit strides
  - Conditional Statements
4. Vectorization with intrinsics



# Access the SIMD through intrinsics

- Intrinsics are vendor/architecture specific
- We will focus on the Intel vector intrinsics
- Intrinsics are useful when
  - the compiler fails to vectorize
  - when the programmer thinks it is possible to generate better code than the one produced by the compiler



# The Intel SSE intrinsics Header file

- SSE can be accessed using intrinsics.
- You must use one of the following header files:
  - #include <xmmi ntrin. h> (for SSE)
  - #include <emmi ntrin. h> (for SSE2)
  - #include <pmmi ntrin. h> (for SSE3)
  - #include <smmi ntrin. h> (for SSE4)
- These include the prototypes of the intrinsics.



# Intel SSE intrinsics Data types

- We will use the following data types:
  - `__m128` packed single precision (vector XMM register)
  - `__m128d` packed double precision (vector XMM register)
  - `__m128i` packed integer (vector XMM register)
- Example

```
#include <xmm intrinsic.h>
int main () {
    ...
    __m128 A, B, C; /* three packed s. p. variables */
    ...
}
```



# Intel SSE intrinsic Instructions

- Intrinsics operate on these types and have the format:  
\_mm\_instruction\_suffix(...)
- Suffix can take many forms. Among them:
  - ss** scalar single precision
  - ps** packed (vector) single precision
  - sd** scalar double precision
  - pd** packed double precision
  - si #** scalar integer (8, 16, 32, 64, 128 bits)
  - su#** scalar unsigned integer (8, 16, 32, 64, 128 bits)



# Intel SSE intrinsics Instructions – Examples

- Load four 16-byte aligned single precision values in a vector:

```
float a[4]={1.0, 2.0, 3.0, 4.0}; //a must be 16-byte aligned
__m128 x = _mm_load_ps(a);
```

- Add two vectors containing four single precision values:

```
__m128 a, b;
__m128 c = _mm_add_ps(a, b);
```



# Intrinsics (SSE)

```
#define n 1024
__attribute__((aligned(16)))
float a[n], b[n], c[n];

int main() {
    for (i = 0; i < n; i++) {
        c[i] = a[i] * b[i];
    }
}
```

```
#include <xmmmintrin.h>
#define n 1024
__attribute__((aligned(16))) float
a[n], b[n], c[n];

int main() {
    __m128 rA, rB, rC;
    for (i = 0; i < n; i += 4) {
        rA = _mm_load_ps(&a[i]);
        rB = _mm_load_ps(&b[i]);
        rC = _mm_mul_ps(rA, rB);
        _mm_store_ps(&c[i], rC);
    }
}
```



# Intel SSE intrinsics

## A complete example

```
#define n 1024
```

Header file

```
int main() {
float a[n], b[n], c[n];
for (i = 0; i < n; i+=4) {
    c[i:i+3]=a[i:i+3]+b[i:i+3];
}
}
```

```
#include <xmmi ntrin.h>
#define n 1024
__attribute__((aligned(16))) float
a[n], b[n], c[n];

int main() {
__m128 rA, rB, rC;
for (i = 0; i < n; i+=4) {
    rA = _mm_load_ps(&a[i]);
    rB = _mm_load_ps(&b[i]);
    rC= _mm_mul_ps(rA, rB);
    _mm_store_ps(&c[i], rC);
}}
```



# Intel SSE intrinsics

## A complete example

```
#define n 1024
```

```
int main() {
    float a[n], b[n], c[n];
    for (i = 0; i < n; i += 4) {
        c[i:i+3]=a[i:i+3]
    }
}
```

Declare 3 vector registers

```
#include <xmmmintrin.h>
#define n 1024
__attribute__((aligned(16))) float
a[n], b[n], c[n];

int main() {
    __m128 rA, rB, rC;
    for (i = 0; i < n; i += 4) {
        rA = _mm_load_ps(&a[i]);
        rB = _mm_load_ps(&b[i]);
        rC= _mm_mul_ps(rA, rB);
        _mm_store_ps(&c[i], rC);
    }
}
```



# Intel SSE intrinsics

## A complete example

```
#define n 1000

int main() {
    float a[n], b[n], c[n];
    for (i = 0; i < n; i+=4) {
        c[i:i+3]=a[i:i+3]+b[i:i+3];
    }
}
```



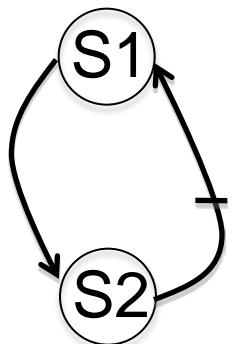
```
#include <xmmmintrin.h>
#define n 1024
__attribute__((aligned(16))) float
a[n], b[n], c[n];

int main() {
    __m128 rA, rB, rC;
    for (i = 0; i < n; i+=4) {
        rA = __mm_load_ps(&a[i]);
        rB = __mm_load_ps(&b[i]);
        rC= __mm_mul_ps(rA, rB);
        __mm_store_ps(&c[i], rC);
    }
}}
```

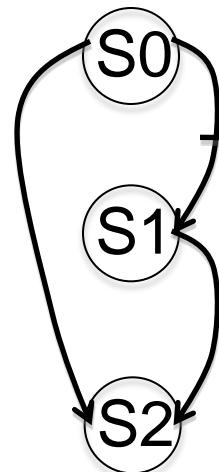


# Node Splitting

```
for (int i=0; i<LEN- 1; i++) {  
S1 a[i ]=b[i ]+c[i ];  
S2 d[i ]=(a[i ]+a[i +1]) *(float)0. 5;  
}
```



```
for (int i=0; i<LEN- 1; i++) {  
S0 temp[i ]=a[i +1];  
S1 a[i ]=b[i ]+c[i ];  
S2 d[i ]=(a[i ]+temp[i ]) *(float) 0. 5  
}
```



# Node Splitting with intrinsics

```
for (int i=0; i<LEN- 1; i++) {  
    a[i]=b[i]+c[i];  
    d[i]=(a[i]+a[i+1]) *(float)0.5;  
}  
  
for (int i=0; i<LEN- 1; i++) {  
    temp[i]=a[i+1];  
    a[i]=b[i]+c[i];  
    d[i]=(a[i]+temp[i]) *(float)0.5;  
}
```

Which code runs faster ?

Why?

```
#include <xmmi ntrin.h>  
#define n 1000  
  
int main() {  
    __m128 rA1, rA2, rB, rC, rD;  
    __m128 r5=_mm_set1_ps((float)0.5)  
for (i = 0; i < LEN-4; i+=4) {  
    rA2= _mm_loadu_ps(&a[i+1]);  
    rB= _mm_load_ps(&b[i]);  
    rC= _mm_load_ps(&c[i]);  
    rA1= _mm_add_ps(rB, rC);  
    rD= _mm_mul_ps(_mm_add_ps(rA1, rA2), r5);  
    _mm_store_ps(&a[i], rA1);  
    _mm_store_ps(&d[i], rD);  
}}
```



# Node Splitting with intrinsics

S126

```
for (int i=0; i<LEN- 1; i++) {  
    a[i]=b[i]+c[i];  
    d[i]=(a[i]+a[i+1]) *(float)0.5;  
}
```

S126\_1

```
for (int i=0; i<LEN- 1; i++) {  
    temp[i]=a[i+1];  
    a[i]=b[i]+c[i];  
    d[i]=(a[i]+temp[i]) *(float)0.5;  
}
```

S126\_2

```
#include <xmmi ntrin.h>  
#define n 1000  
  
int main() {  
    __m128 rA1, rA2, rB, rC, rD;  
    __m128 r5=_mm_set1_ps((float)0.5)  
for (i = 0; i < LEN-4; i+=4) {  
    rA2= _mm_loadu_ps(&a[i+1]);  
    rB= _mm_load_ps(&b[i]);  
    rC= _mm_load_ps(&c[i]);  
    rA1= _mm_add_ps(rB, rC);  
    rD= _mm_mul_ps(_mm_add_ps(rA1, rA2), r5);  
    _mm_store_ps(&a[i], rA1);  
    _mm_store_ps(&d[i], rD);  
}}
```



# Node Splitting with intrinsics

S126

S126\_1

## Intel Nehalem

**Compiler report:** Loop was not vectorized. Existence of vector dependence

**Exec. Time scalar code:** 12.6

**Exec. Time vector code:** --

**Speedup:** --

## Intel Nehalem

**Compiler report:** Loop was vectorized.

**Exec. Time scalar code:** 13.2

**Exec. Time vector code:** 9.7

**Speedup:** 1.3

S126\_2

## Intel Nehalem

**Exec. Time intrinsics:** 6.1

**Speedup (versus vector code):** 1.6



# Node Splitting with intrinsics

S126

**IBM Power 7**  
**Compiler report:** Loop was SIMD vectorized  
**Exec. Time scalar code:** 3.8  
**Exec. Time vector code:** 1.7  
**Speedup:** 2.2

S126\_1

**IBM Power 7**  
**Compiler report:** Loop was SIMD vectorized  
**Exec. Time scalar code:** 5.1  
**Exec. Time vector code:** 2.4  
**Speedup:** 2.0

S126\_2

**IBM Power 7**  
**Exec. Time intrinsics:** 1.6  
**Speedup (versus vector code):** 1.5



# Summary

- Microprocessor vector extensions can contribute to improve program performance and the amount of this contribution is likely to increase in the future as vector lengths grow.
- Compilers are only partially successful at vectorizing
- When the compiler fails, programmers can
  - add compiler directives
  - apply loop transformations
- If after transforming the code, the compiler still fails to vectorize (or the performance of the generated code is poor), the only option is to program the vector extensions directly using intrinsics or assembly language.



# Data Dependencies

- The correctness of many many loop transformations including vectorization can be decided using dependences.
- A good introduction to the notion of dependence and its applications can be found in D. Kuck, R. Kuhn, D. Padua, B. Leasure, M. Wolfe: Dependence Graphs and Compiler Optimizations. POPL 1981.



# Compiler Optimizations

- For a longer discussion see:
  - Kennedy, K. and Allen, J. R. 2002 Optimizing Compilers for Modern Architectures: a Dependence-Based Approach. Morgan Kaufmann Publishers Inc.
  - U. Banerjee. Dependence Analysis for Supercomputing. Kluwer Academic Publishers, Norwell, Mass., 1988.
  - Advanced Compiler Optimizations for Supercomputers, by David Padua and Michael Wolfe in Communications of the ACM, December 1986, Volume 29, Number 12.
  - Compiler Transformations for High-Performance Computing, by David Bacon, Susan Graham and Oliver Sharp, in ACM Computing Surveys, Vol. 26, No. 4, December 1994.



# Algorithms

- W. Daniel Hillis and Guy L. Steele, Jr.. 1986.  
Data parallel algorithms. *Commun. ACM* 29, 12  
(December 1986), 1170-1183.
- Shyh-Ching Chen, D.J. Kuck, "Time and Parallel  
Processor Bounds for Linear Recurrence  
Systems," IEEE Transactions on Computers, pp.  
701-717, July, 1975



# Thank you

## Questions?

María Garzarán, Saeed Maleki

William Gropp and David Padua

{garzaran,maleki,wgropp,padua}@illinois.edu



# Program Optimization Through Loop Vectorization

María Garzarán, Saeed Maleki

William Gropp and David Padua

{garzaran,maleki,wgropp,padua}@illinois.edu

*Department of Computer Science*

*University of Illinois at Urbana-Champaign*



# **Back-up Slides**



# Measuring execution time

```
time1 = time();  
for (i=0; i<32000; i++)  
    c[i] = a[i] + b[i];
```

```
time2 = time();
```



# Measuring execution time

- Added an outer loop that runs (serially)
  - to increase the running time of the loop

```
time1 = time();
for (j=0; j<200000; j++) {
    for (i=0; i<32000; i++)
        c[i] = a[i] + b[i];

}
time2 = time();
```



# Measuring execution times

- Added an outer loop that runs (serially)
  - to increase the running time of the loop
- Call a dummy () function that is compiled separately
  - to avoid loop interchange or dead code elimination

```
time1 = time();
for (j=0; j<200000; j++) {
    for (i=0; i<32000; i++)
        c[i] = a[i] + b[i];
    dummy();
}
time2 = time();
```



# Measuring execution times

- Added an outer loop that runs (serially)
  - to increase the running time of the loop
- Call a dummy () function that is compiled separately
  - to avoid loop interchange or dead code elimination
- Access the elements of one output array and print the result
  - to avoid dead code elimination

```
time1 = time();
for (j=0; j<200000; j++) {
    for (i=0; i<32000; i++)
        c[i] = a[i] + b[i];
    dummy();
}
time2 = time();
for (j=0; j<32000; j++)
    ret+= a[i];
printf ("Time %f, result %f", (time2 -time1), ret);
```



# Compiling

- Intel icc scalar code

```
icc -O3 -no-vec dummy.o tsc.o -o runnovec
```

- Intel icc vector code

```
icc -O3 -vec-report[n] -xSSE4.2 dummy.o tsc.o -o runvec
```

[n] can be 0,1,2,3,4,5

- **vec-report0**, no report is generated

- **vec-report1**, indicates the line number of the loops that were vectorized

- **vec-report2 .. 5**, gives a more detailed report that includes the loops that were not vectorized and the reason for that.



# Compiling

```
flags = -O3 -qaltivec -qhot -qarch=pwr7 -qtune=pwr7  
-qipa=malloc16 -qdebug=NSIMDCOST  
-qdebug=alwayspec -qreport
```

- IBM xlc scalar code  
`xlc -qnoenablevmx dummy.o tsc.o -o runovec`
- IBM vector code  
`xlc -qenablevmx dummy.o tsc.o -o runvec`



# Strip Mining

This transformation improves locality and is usually combined with vectorization



# Strip Mining

```
for (i=1; i<LEN; i++)  
{  
    a[i] = b[i];  
    c[i] = c[i - 1] +  
    a[i];
```

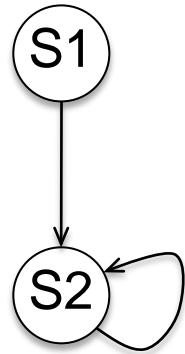
- first statement can be vectorized
- second statement cannot be vectorized because of self-true dependence



```
for (i=1; i<LEN; i++)  
    a[i] = b[i];  
  
for (i=1; i<LEN; i++)  
    c[i] = c[i - 1] + a[i];
```

By applying loop distribution the compiler will vectorize the first statement

But, ... loop distribution will increase the cache miss ratio if array a[] is large





# Strip Mining

## Loop Distribution

```
for (i=1; i<LEN; i++)  
    a[i] = b[i];  
for (i=1; i<LEN; i++)  
    c[i] = c[i - 1] + a[i];
```

## Strip Mining

```
for (i=1; i<LEN;  
     i+=strip_size) {  
    →   for (j=i; j<strip_size; j++)  
        a[j] = b[j];  
    for (j=i; j<strip_size; j++)  
        c[j] = c[j - 1] + a[j];  
}
```

strip\_size is usually a small value (4, 8, 16 or 32).



# Strip Mining

- Another example

```
int v[N];
...
for (int i=0; i<N; i++) {
    Transform (v[i]);
}
for (int i=0; i<N; i++) {
    Light (v[i]);
}
```

```
int v[N];
...
for (int i=0; i<N; i+=strip_size) {
    for (int j=i;j<strip_size;j++) {
        Transform (v[j]);
    }
    for (int j=i;j<strip_size;j++) {
        Light (v[j]);
    }
}
```



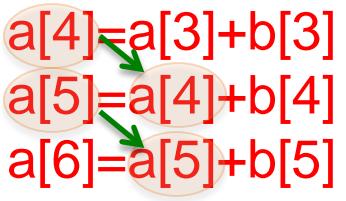
# Evolution of Intel Vector Instructions

- MMX (1996, Pentium)
  - *CPU-based MPEG decoding*
  - Integers only, 64-bit divided into 2 x 32 to 8 x 8
  - Phased out with SSE4
- SSE (1999, Pentium III)
  - *CPU-based 3D graphics*
  - 4-way float operations, single precision
  - 8 new 128 bit Register, 100+ instructions
- SSE2 (2001, Pentium 4)
  - *High-performance computing*
  - Adds 2-way float ops, double-precision; same registers as 4-way single-precision
  - Integer SSE instructions make MMX obsolete
- SSE3 (2004, Pentium 4E Prescott)
  - *Scientific computing*
  - New 2-way and 4-way vector instructions for complex arithmetic
- SSSE3 (2006, Core Duo)
  - Minor advancement over SSE3
- SSE4 (2007, Core2 Duo Penryn)
  - *Modern codecs, cryptography*
  - New integer instructions
  - Better support for unaligned data, super shuffle engine

More details at [http://en.wikipedia.org/wiki/Streaming SIMD\\_Extensions](http://en.wikipedia.org/wiki/Streaming SIMD_Extensions)

# Run-Time Symbolic Resolution

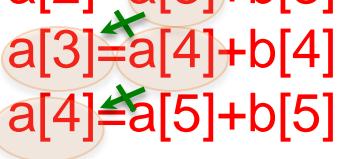
```
for (int i=3; i<n; i++) {  
S1  a[i+t] = a[i] + b[i];  
}
```

If ( $t > 0$ )  $\rightarrow$  self true dependence  
 $t = 1$     

a[4]=a[3]+b[3]  
a[5]=a[4]+b[4]  
a[6]=a[5]+b[5]

Cannot be vectorized

If ( $t \leq 0$ )  $\rightarrow$  no dependence or  
self anti-dependence

$t = -1$     

a[2]=a[3]+b[3]  
a[3]=a[4]+b[4]  
a[4]=a[5]+b[5]

Can be vectorized



# Loop Vectorization – Example I

S113

```
for (i=0; i<LEN; i++) {  
    a[i] = b[i] + c[i]  
    d[i] = a[i] + (float) 1.0;  
}
```

S113\_1

```
for (i=0; i<LEN; i++)  
    a[i] = b[i] + c[i]  
for (i=0; i<LEN; i++)  
    d[i] = a[i] + (float) 1.0;
```

The Intel ICC compiler generated the same code in both cases

S113

**Intel Nehalem**

**Compiler report:** Loop was vectorized in both cases

**Exec. Time scalar code:** 10.2

**Exec. Time vector code:** 6.3

**Speedup:** 1.6

S113\_1

**Intel Nehalem**

**Compiler report:** Fused loop was vectorized

**Exec. Time scalar code:** 10.2

**Exec. Time vector code:** 6.3

**Speedup:** 1.6





# Loop Vectorization – Example I

S113

```
for (i=0; i<LEN; i++) {  
    a[i] = b[i] + c[i]  
    d[i] = a[i] + (float) 1.0;  
}
```

S113\_1

```
for (i=0; i<LEN; i++)  
    a[i] = b[i] + c[i]  
for (i=0; i<LEN; i++)  
    d[i] = a[i] + (float) 1.0;
```

S113

**IBM Power 7**  
**Compiler report:** Loop was SIMD  
vectorized  
**Exec. Time scalar code:** 3.1  
**Exec. Time vector code:** 1.5  
**Speedup:** 2.0

S113\_1

**IBM Power 7**  
**Compiler report:** Loop was SIMD  
vectorized  
**Exec. Time scalar code:** 3.7  
**Exec. Time vector code:** 2.3  
**Speedup:** 1.6



# How do we access the SIMD units?

- Three choices
  1. C code and a vectorizing compiler
  1. Macros or Vector Intrinsics
  1. Assembly Language



# How do we access the SIMD units?

- Three choices

1. C code and a vectorizing compiler

```
for (i=0; i<32000; i++)
    c[i] = a[i] + b[i];
```

1. Macros or Vector Intrinsics

1. Assembly Language



# How do we access the SIMD units?

- Three choices
  1. C code and a vectorizing compiler

## 1. Macros or Vector Intrinsics

```
void example() {
    __m128 rA, rB, rC;
    for (int i = 0; i < 32000; i += 4) {
        rA = _mm_load_ps(&a[i]);
        rB = _mm_load_ps(&b[i]);
        rC = _mm_add_ps(rA, rB);
        _mm_store_ps(&c[i], rC);
    }
}
```

## 1. Assembly Language



# How do we access the SIMD units?

- Three choices
  1. C code and a vectorizing compiler
  1. Macros or Vector Intrinsics

## 1. Assembly Language

```
. . B8. 5
    movaps      a(, %rdx, 4), %xmm0
    movaps      16+a(, %rdx, 4), %xmm1
    addps      b(, %rdx, 4), %xmm0
    addps      16+b(, %rdx, 4), %xmm1
    movaps      %xmm0, c(, %rdx, 4)
    movaps      %xmm1, 16+c(, %rdx, 4)
    addq        $8, %rdx
    cmpq        $32000, %rdx
    jl         . . B8. 5
```



# What are the speedups?

- Speedups obtained by XLC compiler

| Test Suite Collection             | Average Speed Up |
|-----------------------------------|------------------|
| Automatically by XLC              | 1.73             |
| By adding classic transformations | 3.48             |
| By adding new transformations     | 3.64             |
| By adding manual vectorization    | 3.78             |



# Another example: matrix-matrix multiplication

S111

```
void MMM(float** a, float** b, float** c) {  
    for (int i=0; i<LEN; i++)  
        for (int j=0; j<LEN; j++) {  
            c[i][j] = (float) 0.;  
            for (int k=0; k<LEN; k++)  
                c[i][j] += a[i][k] * b[k][j];  
        }  
}
```

## Intel Nehalem

**Compiler report:** Loop was not vectorized: existence of vector dependence

**Exec. Time scalar code:** 2.5 sec

**Exec. Time vector code:** --

**Speedup:** --



## IBM Power 7

**Compiler report:** Loop was not SIMD vectorized because a data dependence prevents SIMD vectorization.

**Exec. Time scalar code:** 0.74 sec

**Exec. Time vector code:** --

**Speedup:** --

# Another example: matrix-matrix multiplication

S111\_1

Added compiler  
directives

```
void MMM(float** __restrict__ a, float** __restrict__ b,  
float** __restrict__ c) {  
    for (int i=0; i<LEN; i++)  
        for (int j=0; j<LEN; j++) {  
            c[i][j] = (float) 0.;  
            for (int k=0; k<LEN; k++)  
                c[i][j] += a[i][k] * b[k][j];  
    } }
```

## Intel Nehalem

**Compiler report:** Loop was not vectorized: existence of vector dependence

**Exec. Time scalar code:** 2.5 sec

**Exec. Time vector code:** --

**Speedup:** --



## IBM Power 7

**Compiler report:** Loop was not SIMD vectorized because a data dependence prevents SIMD vectorization.

**Exec. Time scalar code:** 0.74 sec

**Exec. Time vector code:** --

**Speedup:** --

# Another example: matrix-matrix multiplication

S111\_3

Loop  
interchagne

```
void MMM(float** __restrict__ a, float** __restrict__ b,  
float** __restrict__ c) {  
    for (int i = 0; i < LEN2; i++)  
        for (int j = 0; j < LEN2; j++)  
            C[i][j] = (float)0.;  
    for (int i=0; i<LEN; i++)  
        for (int k=0; k<LEN; j++) {  
            for (int j=0; j<LEN; k++)  
                c[i][j] += a[i][k] * b[k][j];  
    }  
}
```

Intel Nehalem

**Compiler report:** Loop was  
vectorized

**Exec. Time scalar code:** 0.8 sec

**Exec. Time vector code:** 0.3 sec

**Speedup:** 2.7



IBM Power 7

**Compiler report:** Loop was not SIMD  
vectorized because a data dependence  
prevents SIMD vectorization.

**Exec. Time scalar code:** 0.74 sec

**Exec. Time vector code:** --

**Speedup:** --

# Definition of Dependence

- A statement S is said to be data dependent on another statement T if
  - S accesses the same data being accessed by an earlier execution of T
  - S, T or both write the data.
- Dependence analysis can be used to discover data dependences between statements



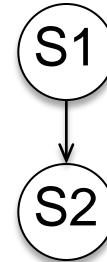


# Data Dependence

Flow dependence (True dependence)

S1:  $X = A + B$

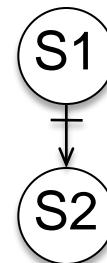
S2:  $C = X + A$



Anti dependence

S1:  $A = X + B$

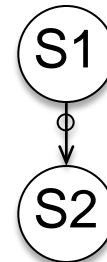
S2:  $X = C + D$



Output dependence

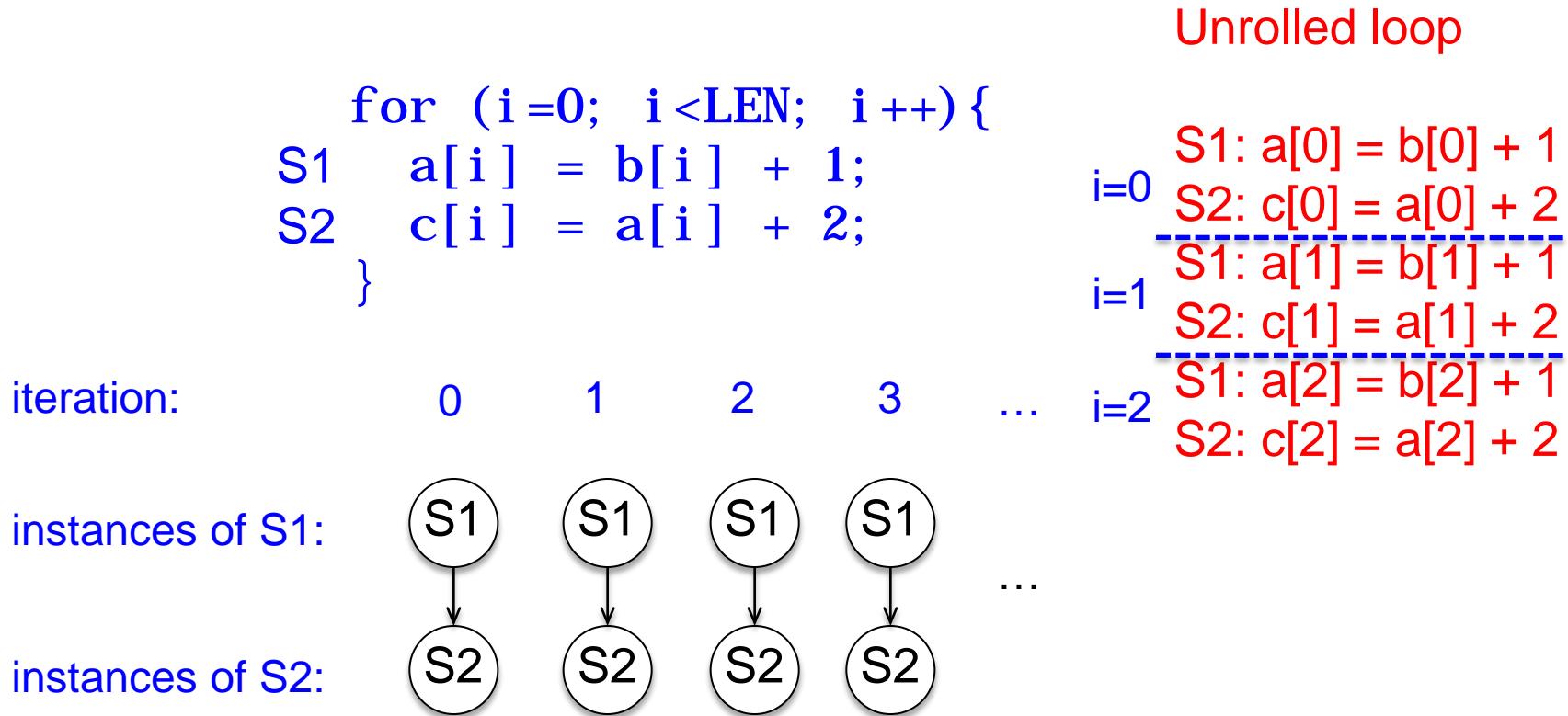
S1:  $X = A + B$

S2:  $X = C + D$



# Dependences in Loops

- Dependences in loops are easy to understand if loops are unrolled.  
Now the dependences are between statement “instances”



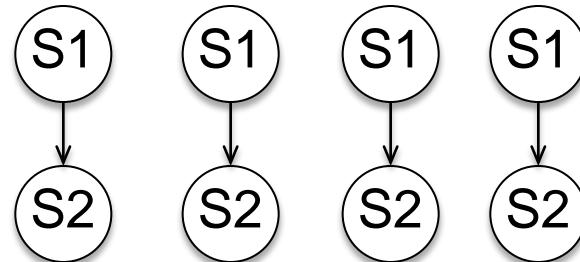
# Dependences in Loops

- Dependences in loops are easy to understand if loops are unrolled.  
Now the dependences are between statement “instances”

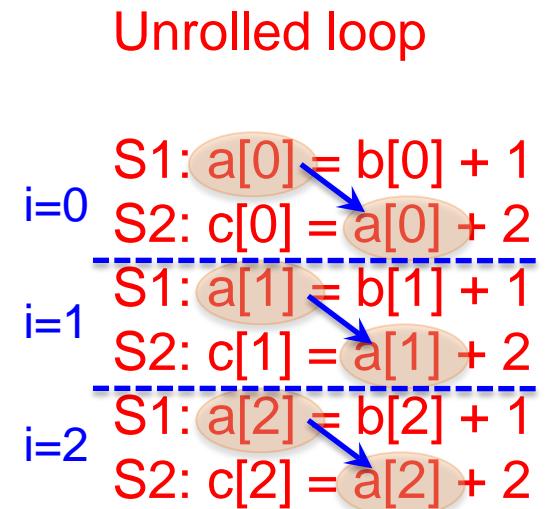
```
for (i=0; i<LEN; i++) {  
    S1 a[i] = b[i] + 1;  
    S2 c[i] = a[i] + 2;  
}
```

iteration:      0      1      2      3      ...

instances of S1:



instances of S2:



Loop independent:  
dependence is within  
the loop iteration boundaries



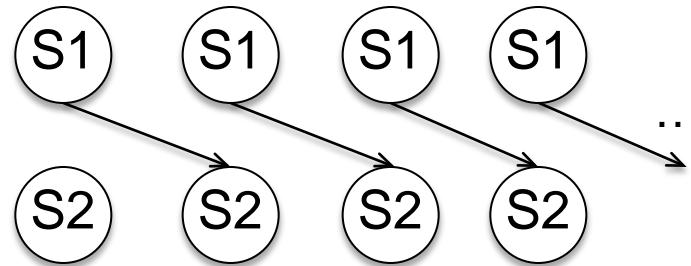
# Dependences in Loops

- A slightly more complex example

```
for (i=1; i<LEN; i++) {  
S1    a[i] = b[i] + 1;  
S2    c[i] = a[i - 1] + 2;  
}
```

iteration:      1      2      3      4      ...

instances of S1:



instances of S2:

Unrolled loop

i=1    S1: a[1] = b[1] + 1  
         S2: c[1] = a[0] + 2

i=2    S1: a[2] = b[2] + 1  
         S2: c[2] = a[1] + 2

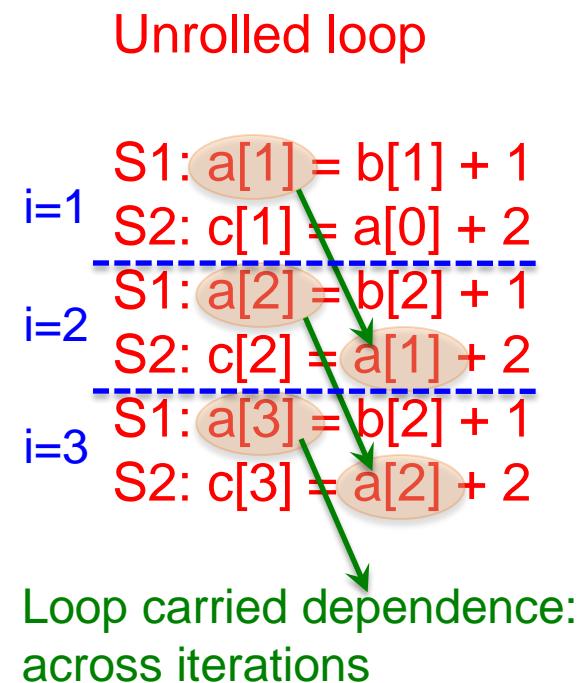
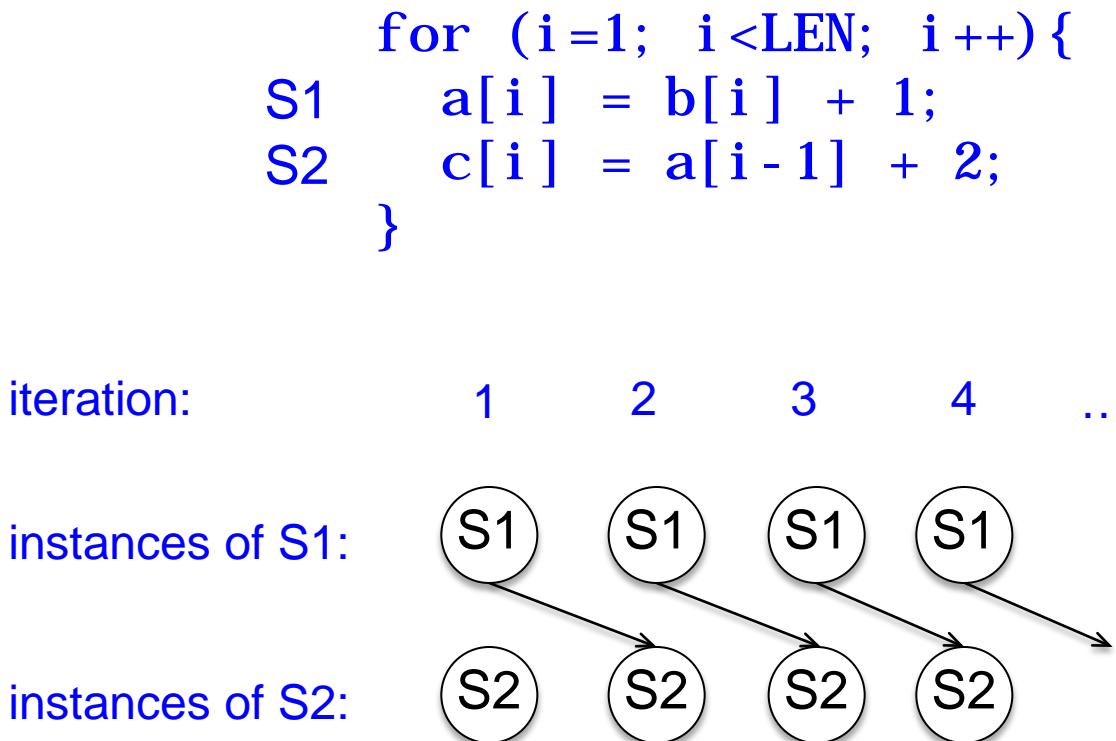
i=3    S1: a[3] = b[3] + 1  
         S2: c[3] = a[2] + 2

...



# Dependences in Loops

- A slightly more complex example



# Dependences in Loops

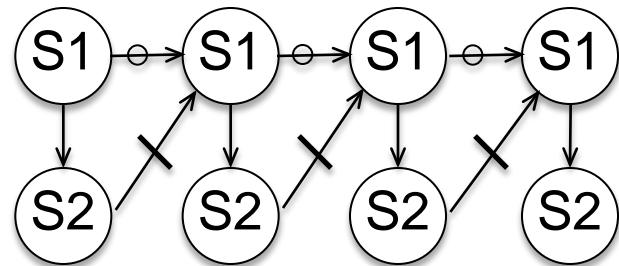
- Even more complex

```
for (i=0; i<LEN; i++) {  
S1    a = b[i] + 1;  
S2    c[i] = a + 2;  
}
```

iteration:

0      1      2      3      ...

instances of S1:



instances of S2:

Unrolled loop

i=0    S1: a = b[0] + 1  
         S2: c[0] = a + 2

i=1    S1: a = b[1] + 1  
         S2: c[1] = a + 2

i=2    S1: a = b[2] + 1  
         S2: c[2] = a + 2

...



# Dependences in Loops

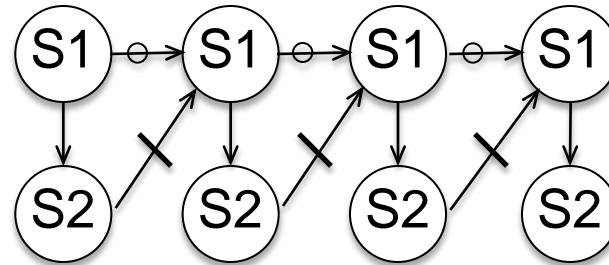
- Even more complex

```
for (i=0; i<LEN; i++) {  
    S1    a = b[i] + 1;  
    S2    c[i] = a + 2;  
}
```

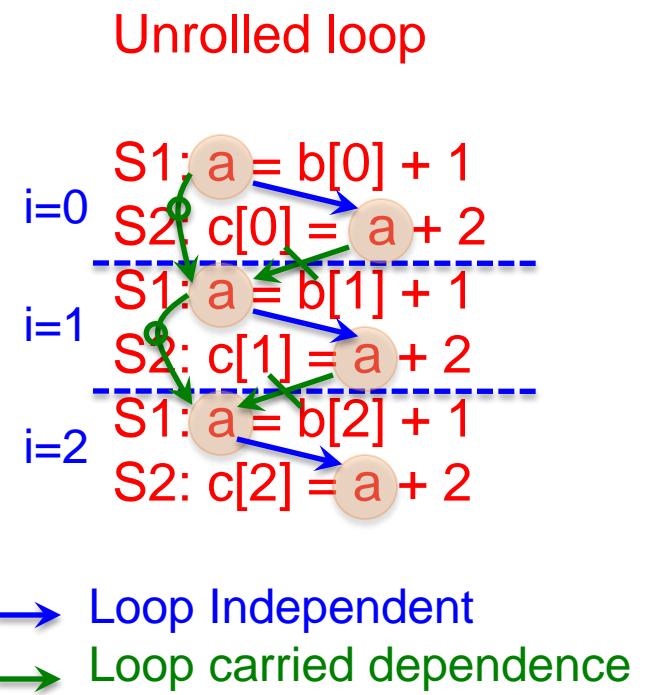
iteration:

0      1      2      3

instances of S1:



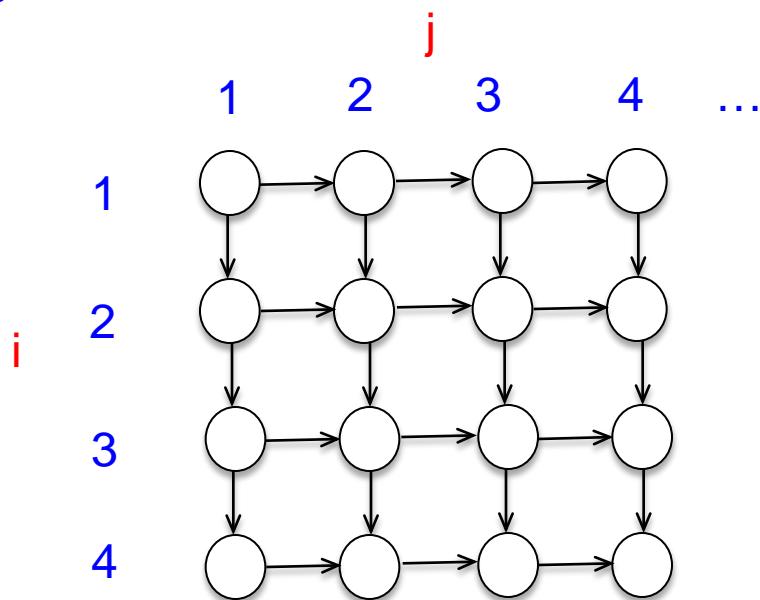
instances of S2:



# Dependences in Loops

- Two dimensional

```
for (i=1; i<LEN; i++) {  
    for (j=1; j<LEN; j++) {  
S1    a[i][j]=a[i][j-1]+a[i]  
    }  
}
```



# Unrolled loop

```

    ];
    i=1   j=1 a[1][1] = a[1][0] + a[0][1]
          j=2 a[1][2] = a[1][1] + a[0][2]
          j=3 a[1][3] = a[1][2] + a[0][3]
          j=4 a[1][4] = a[1][3] + a[0][4]
    -----
    i=2   j=1 a[2][1] = a[2][0] + a[1][1]
          j=2 a[2][2] = a[2][1] + a[1][2]
          j=3 a[2][3] = a[2][2] + a[1][3]
          j=4 a[2][4] = a[2][3] + a[1][4]

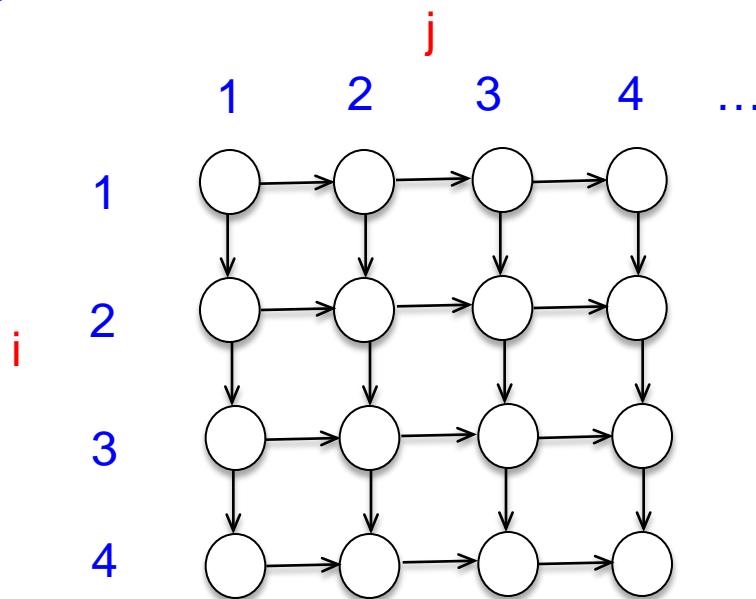
```



# Dependences in Loops

- Two dimensional

```
for (i=1; i<LEN; i++) {  
    for (j=1; j<LEN; j++) {  
S1    a[i][j]=a[i][j-1]+a[i-1][j];  
    } }
```



Unrolled loop

$i=1$

$$\begin{cases} j=1 \ a[1][1] = a[1][0] + a[0][1] \\ j=2 \ a[1][2] = a[1][1] + a[0][2] \\ j=3 \ a[1][3] = a[1][2] + a[0][3] \\ j=4 \ a[1][4] = a[1][3] + a[0][4] \end{cases}$$

$i=2$

$$\begin{cases} j=1 \ a[2][1] = a[2][0] + a[1][1] \\ j=2 \ a[2][2] = a[2][1] + a[1][2] \\ j=3 \ a[2][3] = a[2][2] + a[1][3] \\ j=4 \ a[2][4] = a[2][3] + a[1][4] \end{cases}$$

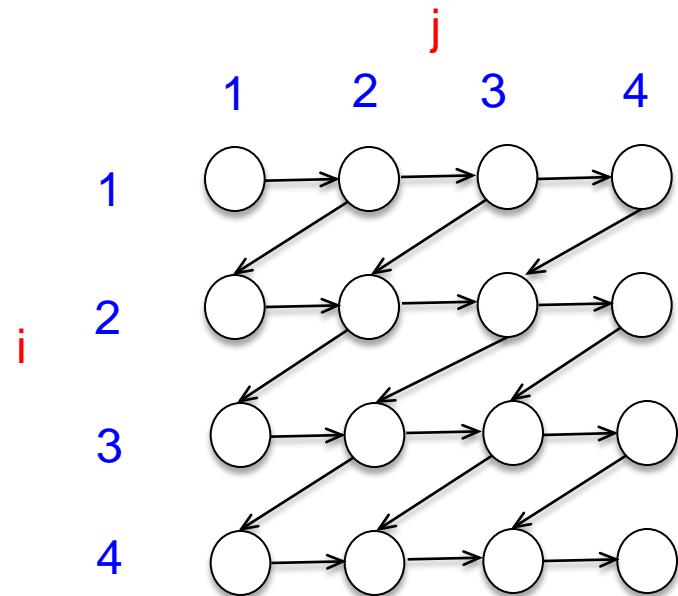
Loop carried dependence



# Dependences in Loops

- Another two dimensional loop

```
for (i=1; i<LEN; i++) {  
    for (j=1; j<LEN; j++) {  
        a[i][j]=a[i][j - 1]+a[i - 1][j + 1];  
    }  
}
```



Unrolled loop

$$\begin{array}{ll} & \left. \begin{array}{l} j=1 \quad a[1][1] = a[1][0] + a[0][2] \\ j=2 \quad a[1][2] = a[1][1] + a[0][3] \\ j=3 \quad a[1][3] = a[1][2] + a[1][4] \\ j=4 \quad a[1][4] = a[1][3] + a[1][5] \end{array} \right\} i=1 \\ & \left. \begin{array}{l} j=1 \quad a[2][1] = a[2][0] + a[1][2] \\ j=2 \quad a[2][2] = a[2][1] + a[1][3] \\ j=3 \quad a[2][3] = a[2][2] + a[1][4] \\ j=4 \quad a[2][4] = a[2][3] + a[1][5] \end{array} \right\} i=2 \end{array}$$

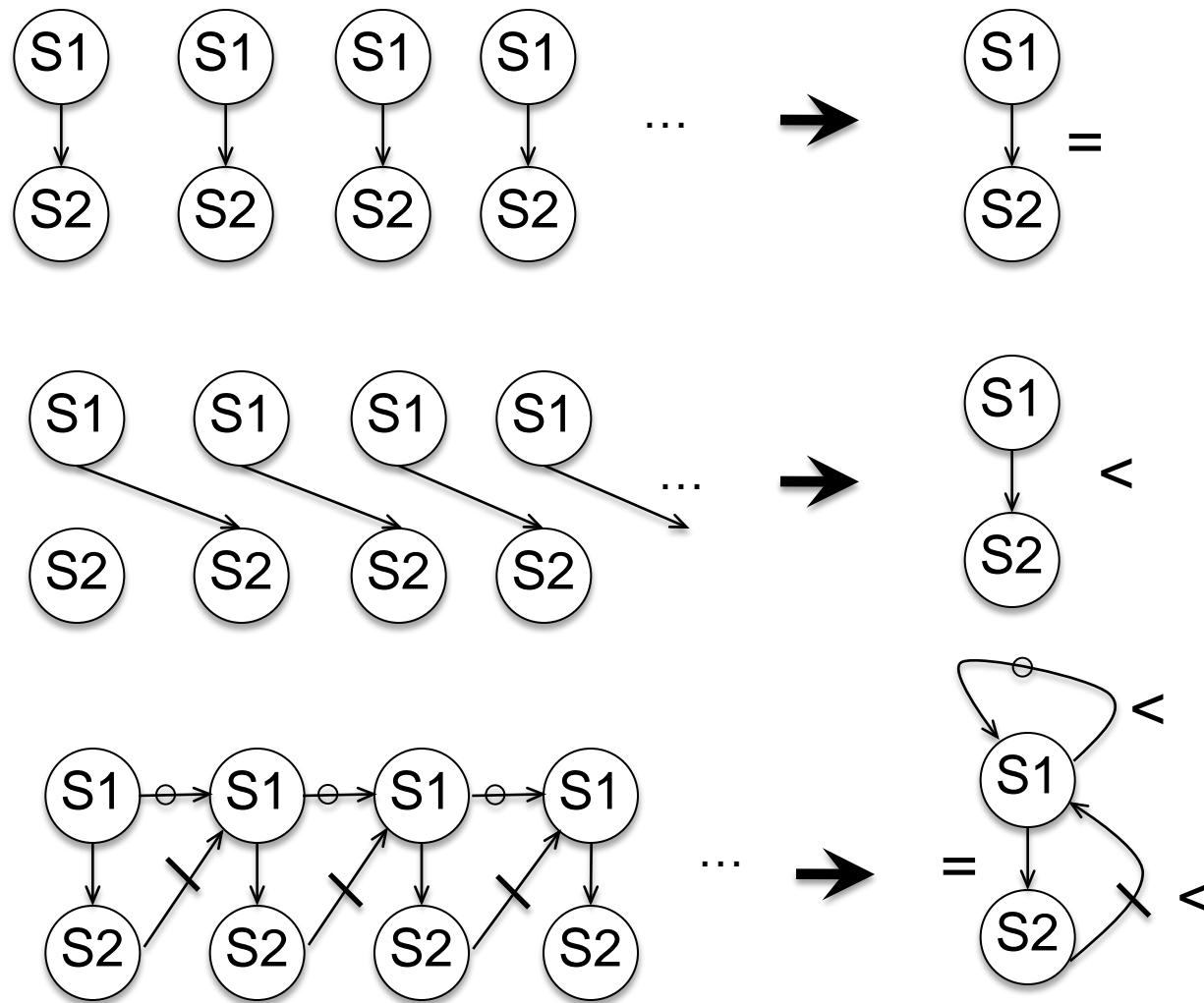
Loop carried dependence



# Dependences in Loops

- The representation of these dependence graphs inside the compiler is a “collapsed” version of the graph where the arcs are annotated with direction (or distances) to reduce ambiguities.
- In the collapsed version each statement is represented by a node and each ordered pair of variable accesses is represented by an arc





# Loop Vectorization

- When the loop has several statements, it is better to first strip-mine the loop and then distribute

strip\_size is usually a small value (4, 8, 16, 32)

Scalar code

```
for (i=0; i<LEN; i++) {  
S1 a[i]=b[i]+(float)1.0;  
S2 c[i]=b[i]+(float)2.0;  
}
```

Scalar code,  
loops are distributed

```
for (i=0; i<LEN; i++)  
    a[i]=b[i]+(float)1.0;  
for (i=0; i<LEN; i++)  
    c[i]=b[i]+(float)2.0;
```



loop distribution will increase  
the cache miss ratio if array  
b[] is large

Scalar code,  
loops are strip-mined

```
for (i=0; i<LEN; i+=strip_size){  
    for (j=i; j<i+strip_size; j++) {  
        a[j]=b[j]+(float)1.0;  
        c[j]=b[j]+(float)2.0;  
    }  
}
```

and then distributed

```
for (i=0; i<LEN; i+=strip_size){  
    for (j=i; j<i+strip_size; j++)  
        a[j]=b[j]+(float)1.0;  
    for (j=i; j<i+strip_size; j++)  
        c[j]=b[j]+(float)2.0;  
}
```



# Loop Vectorization

Scalar code

```
for (i=0; i<LEN; i++) {  
    a[i]=b[i]+(float)1.0;  
    c[i]=b[i]+(float)2.0;  
}
```

Scalar code,  
loops are distributed

```
for (i=0; i<LEN; i++)  
    a[i]=b[i]+(float)1.0;  
for (i=0; i<LEN; i++)  
    c[i]=b[i]+(float)2.0;
```

Scalar code,  
loops are strip-mined  
and distributed

```
for (i=0; i<LEN; i+=strip_size){  
    for (j=i; j<i+strip_size; j++)  
        a[j]=b[j]+(float)1.0;  
    for (j=i; j<i+strip_size; j++)  
        c[j]=b[j]+(float)2.0;  
}
```

Vector code

```
a[0:LEN-1]=b[0:LEN-1]+(float)1.0;  
c[0:LEN-1]=d[0:LEN-1]+(float)2.0;
```

Vector code

```
for (i=0; i<LEN; i+=4{  
    a[i:i+3]=b[i:i+3]+(float)1.0;  
    c[i:i+3]=d[i:i+3]+(float)2.0;  
}
```



# Loop Vectorization

S112

Scalar code

```
for (i=0; i<LEN; i++) {  
    a[i]=b[i]+(float)1.0;  
    c[i]=b[i]+(float)2.0;  
}
```

S112\_1

Scalar code  
loops are distributed

```
for (i=0; i<LEN; i++)  
    a[i]=b[i]+(float)1.0;  
for (i=0; i<LEN; i++)  
    c[i]=b[i]+(float)2.0;
```

S112\_2

Scalar code  
loops are strip-mined

```
for (i=0; i<LEN; i+=4) {  
    for (j=i; j<i+4; j++) {  
        a[j]=b[j]+(float)1.0;  
        c[j]=b[j]+(float)2.0;  
    }  
}
```

The Intel ICC generated the same vector code (the equivalent to the strip-mined version) in all the cases

S112 and S112\_2

**Intel Nehalem**

**Compiler report:** Loop was vectorized

**Exec. Time scalar code:** 9.6

**Exec. Time vector code:** 5.5

**Speedup:** 1.7



S112\_1

**Intel Nehalem**

**Compiler report:** Fused loop was vectorized

**Exec. Time scalar code:** 9.6

**Exec. Time vector code:** 5.5

**Speedup:** 1.7

# Loop Vectorization

S112

Scalar code

```
for (i=0; i<LEN; i++) {  
    a[i]=b[i]+(float)1.0;  
    c[i]=b[i]+(float)2.0;  
}
```

S112\_1

Scalar code  
loops are distributed

```
for (i=0; i<LEN; i++)  
    a[i]=b[i]+(float)1.0;  
for (i=0; i<LEN; i++)  
    c[i]=b[i]+(float)2.0;
```

S112\_2

Scalar code  
loops are strip-mined

```
for (i=0; i<LEN; i+=64) {  
    for (j=i; j<i+64; j++) {  
        a[j]=b[j]+(float)1.0;  
        c[j]=b[j]+(float)2.0;  
    }  
}
```

S112

**IBM Power 7**

**Compiler report:** Loop was SIMD vectorized

**Exec. Time scalar code:** 2.7

**Exec. Time vector code:** 1.2

**Speedup:** 2.1

S112\_1

**IBM Power 7**

**report:** Loop was SIMD vectorized

**scalar code:** 2.7

**vector code:** 1.2

**Speedup:** 2.1

S112\_2

**IBM Power 7**

**report:** Loop was SIMD vectorized

**scalar code:** 2.9

**vector code:** 1.4

**Speedup:** 2.07



# Loop Vectoriation

- Our observations
  - Compiler generates vector code when it can apply loop distribution.
    - compiler may have to transform the code so that loop distribution is legal



# Loop Vectorization – Example II

S114

```
for (i=0; i<LEN; i++) {  
    a[i] = b[i] + c[i]  
    d[i] = a[i+1] + (float) 1.0;  
}
```

S114

S114

## Intel Nehalem

**Compiler report:** Loop was not vectorized. Existence of vector dependence

**Exec. Time scalar code:** 12.6

**Exec. Time vector code:** --

**Speedup:** --

## IBM Power 7

**Compiler report:** Loop was SIMD vectorized

**Exec. Time scalar code:** 3.3

**Exec. Time vector code:** 1.8

**Speedup:** 1.8



# Loop Vectorization – Example II

We have observed that  
ICC usually vectorizes only  
if all the dependences  
are forward (except for  
reduction and induction  
variables)

S114

```
for (i=0; i<LEN; i++) {  
    a[i] = b[i] + c[i]  
    d[i] = a[i+1] + (float) 1.0;  
}
```

S114

S114

## Intel Nehalem

**Compiler report:** Loop was not  
vectorized. Existence of vector  
dependence

**Exec. Time scalar code:** 12.6  
**Exec. Time vector code:** --  
**Speedup:** --

## IBM Power 7

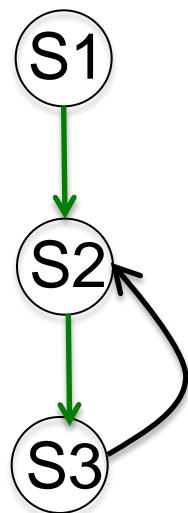
**Compiler report:** Loop was SIMD  
vectorized  
**Exec. Time scalar code:** 3.3  
**Exec. Time vector code:** 1.8  
**Speedup:** 1.8



# Loop vectorization – Example IV

A loop can be partially vectorized

```
for (int i=1; i<LEN; i++) {  
S1  a[i] = b[i] + c[i];  
S2  d[i] = a[i] + e[i-1];  
S3  e[i] = d[i] + c[i];  
}
```



```
for (int i=1; i<LEN; i++) {  
S1  a[i] = b[i] + c[i];  
}  
for (int i=1; i<LEN; i++) {  
S2  d[i] = a[i] + e[i-1];  
S3  e[i] = d[i] + c[i];  
}
```

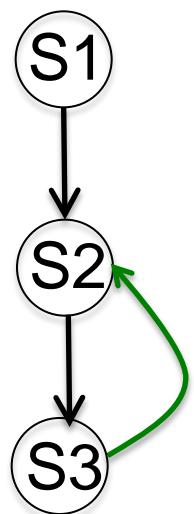
S1 can be vectorized  
S2 and S3 cannot be vectorized  
(A loop with a cycle in the dependence graph cannot be vectorized)



# Loop vectorization – Example IV

A loop can be partially vectorized

```
for (int i=1; i<LEN; i++) {  
S1  a[i] = b[i] + c[i];  
S2  d[i] = a[i] + e[i-1];  
S3  e[i] = d[i] + c[i];  
}
```



```
for (int i=1; i<LEN; i++) {  
S1  a[i] = b[i] + c[i];  
}  
for (int i=1; i<LEN; i++) {  
S2  d[i] = a[i] + e[i-1];  
S3  e[i] = d[i] + c[i];  
}
```

S1 can be vectorized  
S2 and S3 cannot be vectorized  
(A loop with a cycle in the dependence graph cannot be vectorized)



# Loop vectorization – Example IV

S116

```
for (int i=1; i<LEN; i++) {  
    a[i] = b[i] + c[i];  
    d[i] = a[i] + e[i-1];  
    e[i] = d[i] + c[i];  
}
```

The INTEL ICC compiler generates  
the same code in both cases

S116\_1

```
for (int i=1; i<LEN; i++) {  
    a[i] = b[i] + c[i];  
}  
for (int i=1; i<LEN; i++) {  
    d[i] = a[i] + e[i-1];  
    e[i] = d[i] + c[i];  
}
```

S116

**Intel Nehalem**  
**Compiler report:** Loop was partially vectorized  
**Exec. Time scalar code:** 14.7  
**Exec. Time vector code:** 18.1  
**Speedup:** 0.8

S116\_1

**Intel Nehalem**  
**Compiler report:** Loop was vectorized  
**Exec. Time scalar code:** 14.7  
**Exec. Time vector code:** 18.1  
**Speedup:** 0.8



# Loop vectorization – Example IV

S116

```
for (int i=1; i<LEN; i++) {  
    a[i] = b[i] + c[i];  
    d[i] = a[i] + e[i-1];  
    e[i] = d[i] + c[i];  
}
```

S116

S116\_1

```
for (int i=1; i<LEN; i++) {  
    a[i] = b[i] + c[i];  
}  
for (int i=1; i<LEN; i++) {  
    d[i] = a[i] + e[i-1];  
    e[i] = d[i] + c[i];  
}
```

S116\_1

## IBM Power 7

**Compiler report:** Loop was not SIMD vectorized because a data dependence prevents SIMD vectorization

**Exec. Time scalar code:** 13.5

**Exec. Time vector code:** --

**Speedup:** --



## IBM Power 7

**Compiler report:** Loops were fused. Loop was not SIMD vectorized because a data dependence prevents SIMD vectorization.

**Exec. Time scalar code:** 13.5

**Exec. Time vector code:** --

**Speedup:** --

# Blue Water

- Illinois has a long tradition in vectorization.
- Most recent work: vectorization for Blue Waters



Source: Thom Dunning: Blue Waters Project



# Compiler work

## **ADVANCED COMPILER OPTIMIZATIONS FOR SUPERCOMPUTERS**

*Compilers for vector or multiprocessor computers must have certain optimization features to successfully generate parallel code.*

DAVID A. PADUA and MICHAEL J. WOLFE

Communications of the ACM, December 1986 Volume 29, Number 12



# Compiler work

## **ADVANCED COMPILER OPTIMIZATIONS FOR SUPERCOMPUTERS**

*Compilers for vector or multiprocessor computers must have certain optimization features to successfully generate parallel code.*

DAVID A. PADUA and MICHAEL J. WOLFE

Communications of the ACM, December 1986 / volume 29, Number 12



# Cache coherence in shared-memory architectures

Adapted from a lecture by Ian Watson, University of Manchester

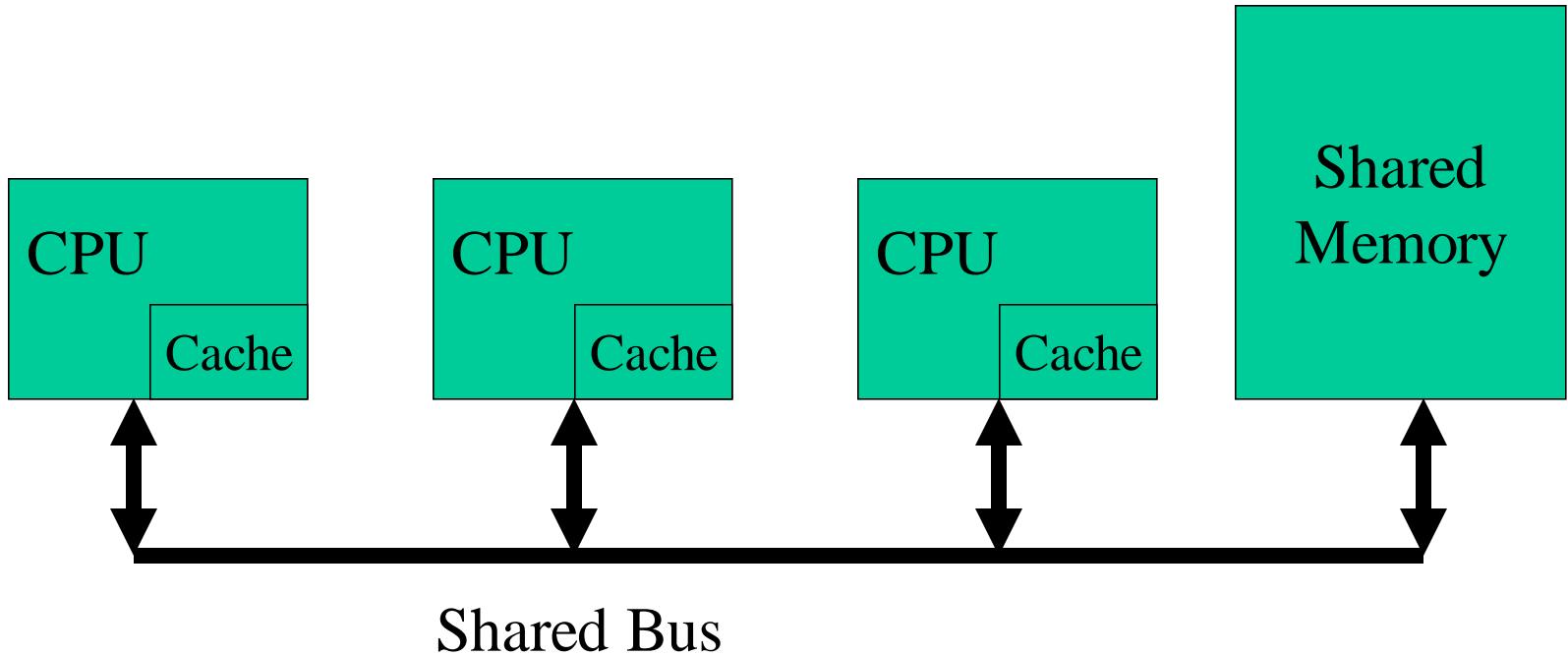
# Overview

- We have talked about optimizing performance on single cores
  - Locality
  - Vectorization
- Now let us look at optimizing programs for a shared-memory multiprocessor.
- Two architectures:
  - Bus-based shared-memory machines (small-scale)
  - Directory-based shared-memory machines (large-scale)

# Bus-based Shared Memory

## Organization

Basic picture is simple :-



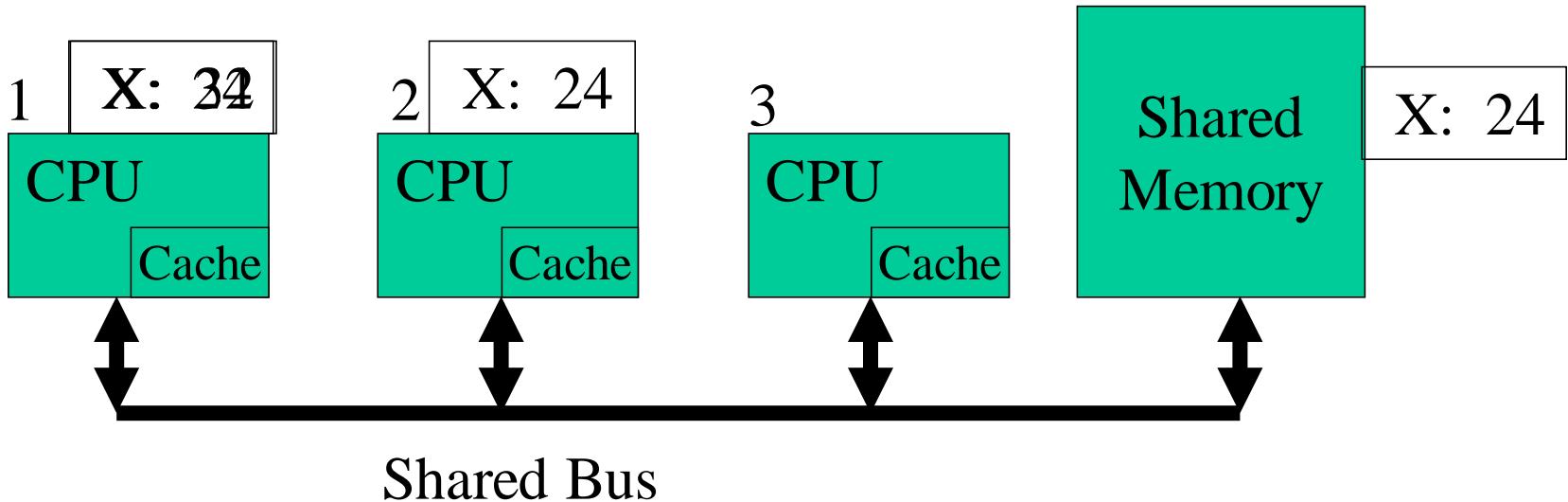
# Organization

- Bus is usually simple physical connection (wires)
- Bus bandwidth limits no. of CPUs
- Could be multiple memory elements
- For now, assume that each CPU has only a single level of cache

# Problem of Memory Coherence

- Assume just single level caches and main memory
- Processor writes to location in its cache
- Other caches may hold shared copies - these will be out of date
- Updating main memory alone is not enough

# Example



Processor 1 reads X: obtains 24 from memory and caches it  
Processor 2 reads X: obtains 24 from memory and caches it  
Processor 1 writes 32 to X: its locally cached copy is updated  
Processor 3 reads X: what value should it get?

Memory and processor 2 think it is 24  
Processor 1 thinks it is 32

Notice that having write-through caches is not good enough<sup>6</sup>

# Bus Snooping

- Each CPU (cache system) ‘snoops’ (i.e. watches continually) for write activity concerned with data addresses which it has cached.
- This assumes a bus structure which is ‘global’, i.e all communication can be seen by all.
- More scalable solution: ‘directory based’ coherence schemes

# Snooping Protocols

- Write Invalidate
  - CPU wanting to write to an address, grabs a bus cycle and sends a ‘write invalidate’ message
  - All snooping caches invalidate their copy of appropriate cache line
  - CPU writes to its cached copy (assume for now that it also writes through to memory)
  - Any shared read in other CPUs will now miss in cache and re-fetch new data.

# Snooping Protocols

- Write Update
  - CPU wanting to write grabs bus cycle and broadcasts new data as it updates its own copy
  - All snooping caches update their copy
- Note that in both schemes, problem of simultaneous writes is taken care of by bus arbitration - only one CPU can use the bus at any one time.

# Update or Invalidate?

- Update looks the simplest, most obvious and fastest, but:-
  - Multiple writes to same word (no intervening read) need only one invalidate message but would require an update for each
  - Writes to same block in (usual) multi-word cache block require only one invalidate but would require multiple updates.

# Update or Invalidate?

- Due to both spatial and temporal locality, previous cases occur often.
- Bus bandwidth is a precious commodity in shared memory multi-processors
- Experience has shown that invalidate protocols use significantly less bandwidth.
- Will consider implementation details only of invalidate.

# Implementation Issues

- In both schemes, knowing if a cached value is not shared (copy in another cache) can avoid sending any messages.
- Invalidate description assumed that a cache value update was written through to memory. If we used a ‘copy back’ scheme other processors could re-fetch old value on a cache miss.
- We need a protocol to handle all this.

# MESI Protocol (1)

- A practical multiprocessor invalidate protocol which attempts to minimize bus usage.
- Allows usage of a ‘write back’ scheme - i.e. main memory not updated until ‘dirty’ cache line is displaced
- Extension of usual cache tags, i.e. invalid tag and ‘dirty’ tag in normal write back cache.

# MESI Protocol (2)

Any cache line can be in one of 4 states (2 bits)

- **Modified** - cache line has been modified, is different from main memory - is the only cached copy. (multiprocessor ‘dirty’)
- **Exclusive** - cache line is the same as main memory and is the only cached copy
- **Shared** - Same as main memory but copies may exist in other caches.
- **Invalid** - Line data is not valid (as in simple cache)

## MESI Protocol (3)

- Cache line changes state as a function of memory access events.
- Event may be either
  - Due to local processor activity (i.e. cache access)
  - Due to bus activity - as a result of snooping
- Cache line has its own state affected only if address matches

# MESI Protocol (4)

- Operation can be described informally by looking at action in local processor
  - Read Hit
  - Read Miss
  - Write Hit
  - Write Miss
- More formally by state transition diagram

# MESI Local Read Hit

- Line must be in one of MES
- This must be correct local value (if M it must have been modified locally)
- Simply return value
- No state change

# MESI Local Read Miss (1)

- No other copy in caches
  - Processor makes bus request to memory
  - Value read to local cache, marked E
- One cache has E copy
  - Processor makes bus request to memory
  - Snooping cache puts copy value on the bus
  - Memory access is abandoned
  - Local processor caches value
  - Both lines set to S

# MESI Local Read Miss (2)

- Several caches have S copy
  - Processor makes bus request to memory
  - One cache puts copy value on the bus (arbitrated)
  - Memory access is abandoned
  - Local processor caches value
  - Local copy set to S
  - Other copies remain S

# MESI Local Read Miss (3)

- One cache has M copy
  - Processor makes bus request to memory
  - Snooping cache puts copy value on the bus
  - Memory access is abandoned
  - Local processor caches value
  - Local copy tagged S
  - **Source (M) value copied back to memory**
  - Source value M -> S

# MESI Local Write Hit (1)

Line must be one of MES

- M
  - line is exclusive and already ‘dirty’
  - Update local cache value
  - no state change
- E
  - Update local cache value
  - State E -> M

# MESI Local Write Hit (2)

- S
  - Processor broadcasts an invalidate on bus
  - Snooping processors with S copy change S->I
  - Local cache value is updated
  - Local state change S->M

# MESI Local Write Miss (1)

Detailed action depends on copies in other processors

- No other copies
  - Value read from memory to local cache (?)
  - Value updated
  - Local copy state set to M

# MESI Local Write Miss (2)

- Other copies, either one in state E or more in state S
  - Value read from memory to local cache - bus transaction marked RWITM (read with intent to modify)
  - Snooping processors see this and set their copy state to I
  - Local copy updated & state set to M

# MESI Local Write Miss (3)

Another copy in state M

- Processor issues bus transaction marked RWITM
- Snooping processor sees this
  - Blocks RWITM request
  - Takes control of bus
  - Writes back its copy to memory
  - Sets its copy state to I

# MESI Local Write Miss (4)

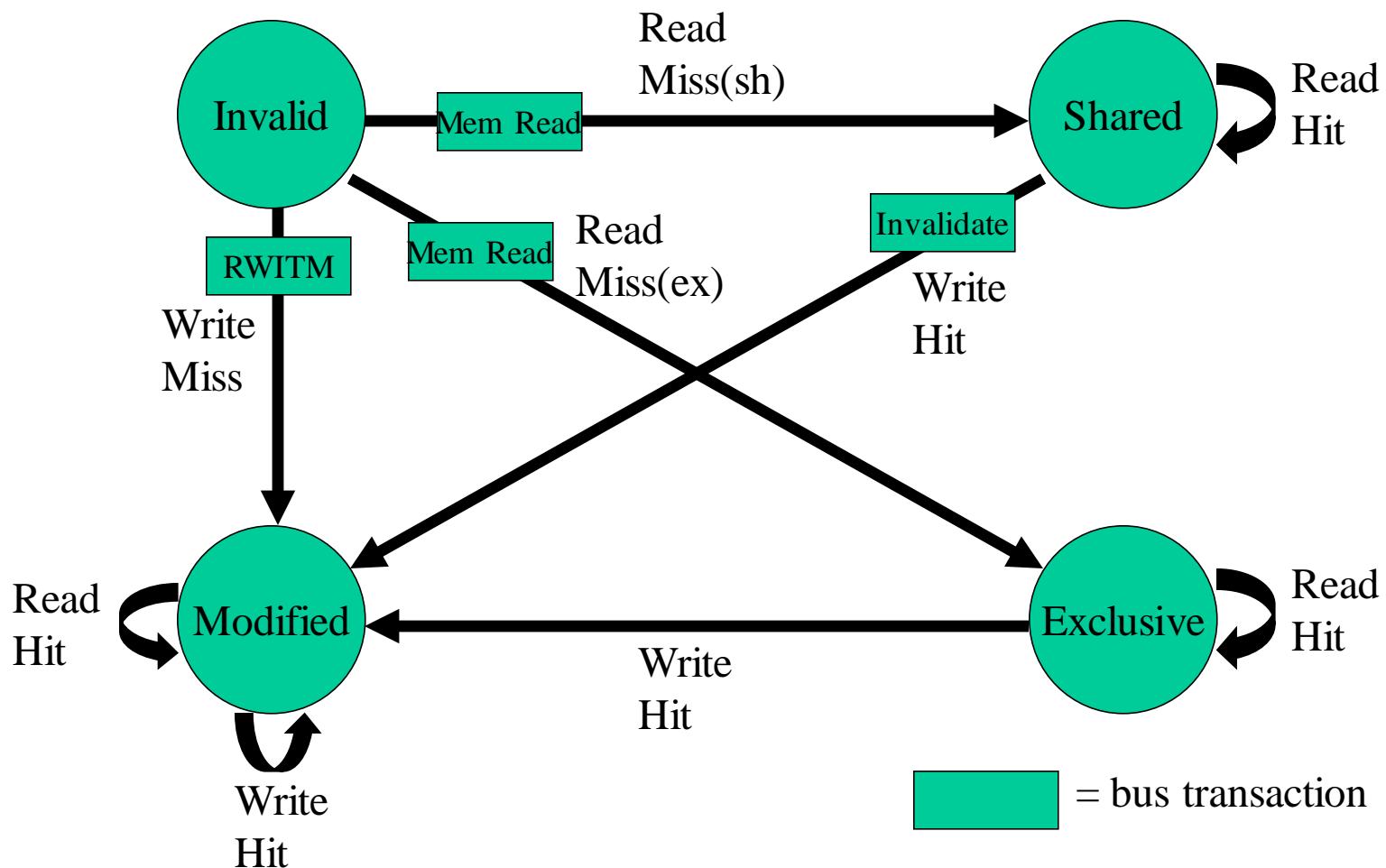
Another copy in state M (continued)

- Original local processor re-issues RWITM request
- Is now simple no-copy case
  - Value read from memory to local cache
  - Local copy value updated
  - Local copy state set to M

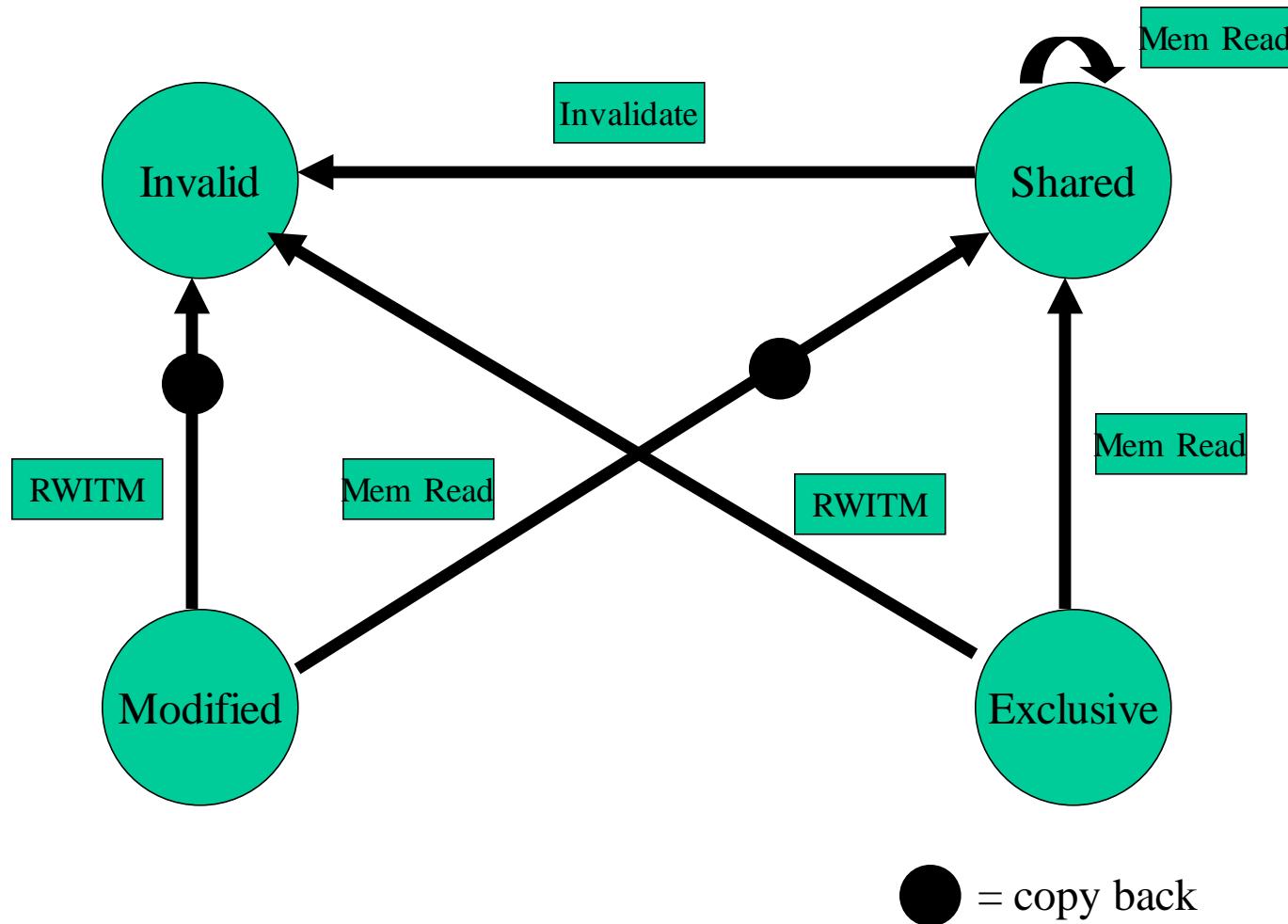
# Putting it all together

- All of this information can be described compactly using a state transition diagram
- Diagram shows what happens to a cache line in a processor as a result of
  - memory accesses made by that processor (read hit/miss, write hit/miss)
  - memory accesses made by other processors that result in bus transactions observed by this snoopy cache (Mem read, RWITM, Invalidate)

# MESI – locally initiated accesses



# MESI – remotely initiated accesses



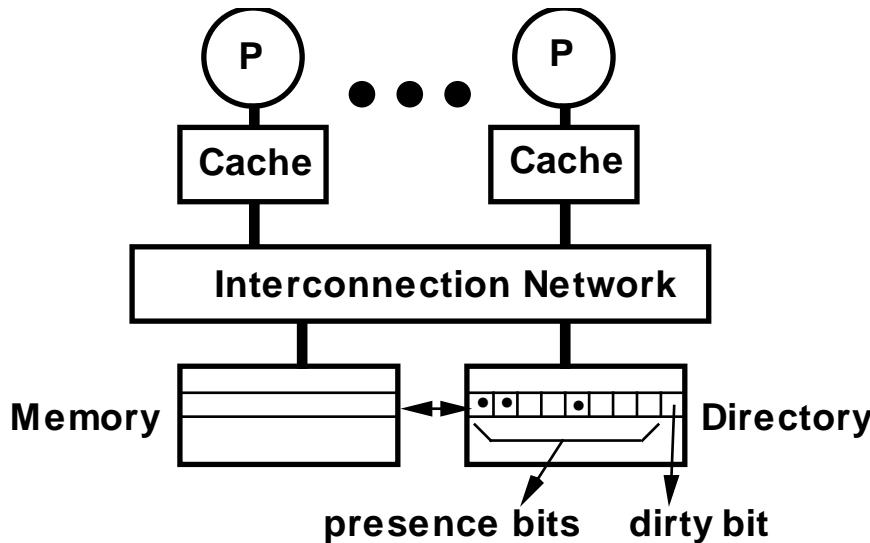
# MESI notes

- There are minor variations (particularly to do with write miss)
- Normal ‘write back’ when cache line is evicted is done if line state is M
- Multi-level caches
  - If caches are inclusive, only the lowest level cache needs to snoop on the bus

# Directory Schemes

- Snoopy schemes do not scale because they rely on broadcast
- Directory-based schemes allow scaling.
  - avoid broadcasts by keeping track of all PEs caching a memory block, and then using point-to-point messages to maintain coherence
  - they allow the flexibility to use any scalable point-to-point network

# Basic Scheme (Censier & Feautrier)



- Assume "k" processors.
- With each cache-block in memory:  
k presence-bits, and 1 dirty-bit
- With each cache-block in cache:  
1 valid bit, and 1 dirty (owner) bit

– Read from main memory by PE-i:

- If dirty-bit is OFF then { read from main memory; turn  $p[i]$  ON; }
- if dirty-bit is ON then { recall line from dirty PE (cache state to shared); update memory; turn dirty-bit OFF; turn  $p[i]$  ON; supply recalled data to PE-i; }

– Write to main memory:

- If dirty-bit OFF then { send invalidations to all PEs caching that block; turn dirty-bit ON; turn  $P[i]$  ON; ... }
- ...

# Key Issues

- Scaling of memory and directory bandwidth
  - Can not have main memory or directory memory centralized
  - Need a distributed memory and directory structure
- Directory memory requirements do not scale well
  - Number of presence bits grows with number of PEs
  - Many ways to get around this problem
    - limited pointer schemes of many flavors
- Industry standard
  - SCI: Scalable Coherent Interface

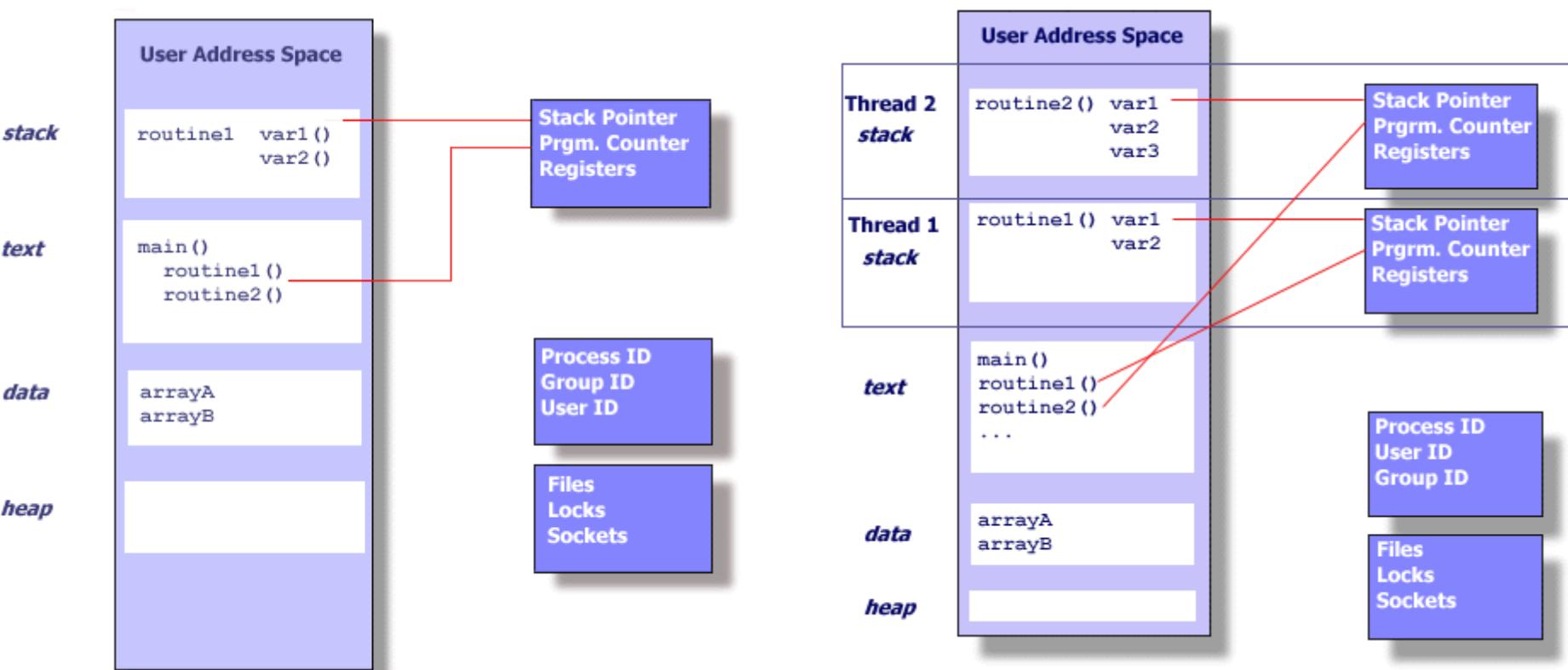
# Programming Shared-memory Machines

**Some slides adapted from Ananth Grama,  
Anshul Gupta, George Karypis, and Vipin  
Kumar ``Introduction to Parallel Computing'',  
Addison Wesley, 2003.**

# Overview

- Thread Basics
- The POSIX Thread API
- Synchronization primitives in Pthreads
  - locks
  - try-locks
- Deadlocks and how to avoid them
- Composite synchronization constructs
- Controlling Thread and Synchronization Attributes
- OpenMP: a Standard for Directive Based Parallel Programming

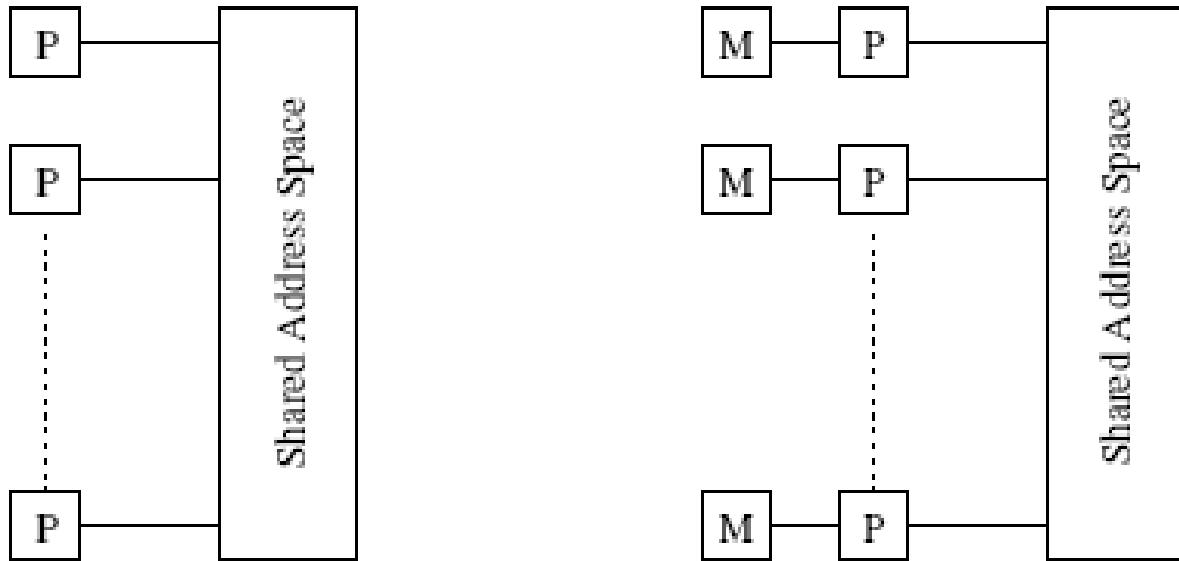
# Process vs Threads



# Thread Basics

- Each thread has its own stack, SP, PC, registers, etc.
- Threads share global variables and heap.
- Caveat: writing programs in which shared space is treated as a “flat” address space may give poor performance
  - Locality is just as important in shared-memory machines as it is in distributed-memory machines

# Thread Basics



- Logical machine model of a **thread-based** programming paradigm
  - Each thread has its own stack and registers
  - Globals and heap are shared by all threads

# The POSIX Thread API

- Commonly referred to as Pthreads, POSIX has emerged as the standard threads API, supported by most vendors.
- The concepts discussed here are largely independent of the API and can be used for programming with other thread APIs (NT threads, Solaris threads, Java threads, etc.) as well.

# Thread Basics: Creation and Termination

- Creating Pthreads:

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

- Thread is created and it starts to execute **thread\_function** with parameter **arg**
- Thread handle: name for thread

# Terminating threads

- Thread terminated when:
  - it returns from its starting routine, or
  - it makes a call to `pthread_exit()`
- Main thread
  - exits with `pthread_exit()`: other threads will continue to execute
  - Otherwise: other threads automatically terminated
- Cleanup:
  - `pthread_exit()` routine does not close files
  - any files opened inside the thread will remain open after the thread is terminated.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid) {
    printf("\n%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0;t<NUM_THREADS;t++){
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){ printf("ERROR; return code from pthread_create() is %d\n", rc);
                  exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

# Output

Creating thread 0

Creating thread 1

0: Hello World!

1: Hello World!

Creating thread 2

Creating thread 3

2: Hello World!

3: Hello World!

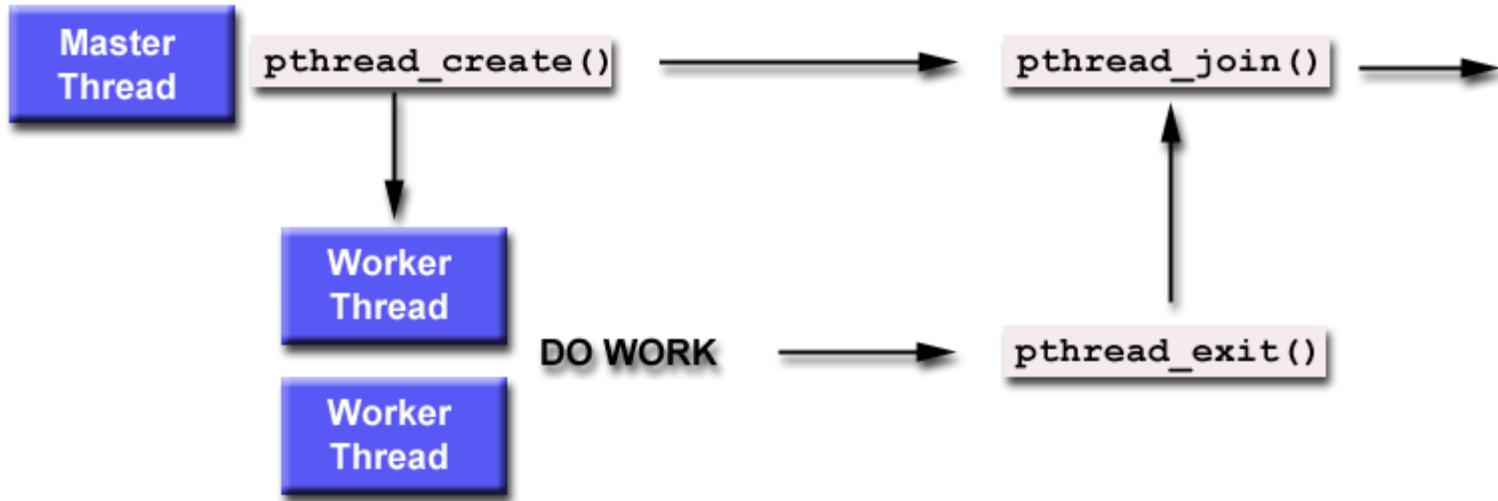
Creating thread 4

4: Hello World!

# Synchronizing threads

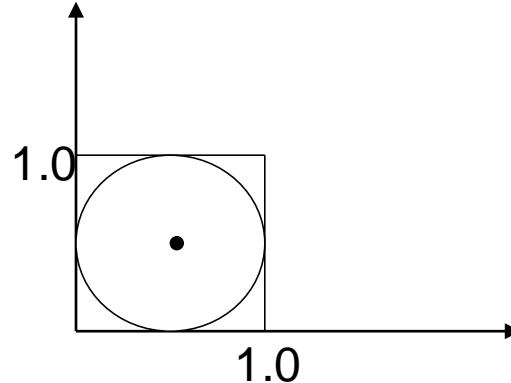
- "Joining" is one way to synchronize threads (not used very often)

`pthread_join (threadid,status)`



- The `pthread_join()` function blocks the calling thread until the specified thread terminates.
- The programmer can obtain the target thread's termination return status if it was specified in the target thread's call to `pthread_exit()`.

# Threads: Example 2



- Area of circle =  $\pi * 0.25$
- Area of square = 1
- So if we shoot randomly into square, probability of hitting circle is  $\pi * 0.25$
- Estimating value of  $\pi$ :
  - generate a large number of random values inside the unit square
  - see what fraction of them fall inside circle and multiply by 4
- Simple example of Monte Carlo method: estimate some value by repeated sampling of some space
- Monte Carlo method can be easily parallelized provided each parallel thread generates independent random numbers

# Generating random numbers

- `int rand_r(unsigned int *seedp)`
  - each call generates one number in a pseudo-random number sequence
  - call it multiple times to generate the sequence
  - thread-safe: can be called safely by multiple threads without interference
  -

# Threads: Example2

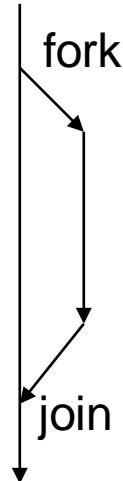
```
#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 512
void *compute_pi (void *);
.....
main() {
    ...
    pthread_t p_threads[MAX_THREADS];
    pthread_attr_t attr;
    pthread_attr_init (&attr);
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
                      (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    ...
}
```

# Threads: Example2 (contd.)

```
void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double rand_no_x, rand_no_y;
    int local_hits;
    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            local_hits++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

# Synchronizing threads

- Style of computing shown in Example 2 is sometimes called fork-join parallelism



- This style of parallel execution in which threads only synchronize at the end is quite rare
- Usually, threads need to synchronize during their execution

# Need for synchronization

- Two common scenarios:
  - Mutual exclusion
    - Shared “resource” such as variable or device
    - Only one thread at a time can access resource
    - **Critical section**: portion of code that should be executed by only thread at a time
  - Producer-consumer
    - One thread (producer) generates a sequence of values
    - Another thread (consumer) reads these values
    - Values are communicated by writing them into a shared buffer
    - Producer must block if buffer is full
    - Consumer must block if buffer is empty

# Need for Mutual Exclusion

- When multiple threads attempt to manipulate the same data item, the results can often be incorrect if proper care is not taken to synchronize them.
- Consider:

```
/* each thread tries to update variable best_cost as follows */  
if (my_cost < best_cost)  
    best_cost = my_cost;
```
- Assume that there are two threads, the initial value of best\_cost is 100, and the values of my\_cost are 50 and 75 at threads t1 and t2.
- Depending on the schedule of the threads, the value of best\_cost could be 50 or 75!
  - Thread 1 reads best\_cost (100)
  - Thread 2 reads best\_cost (100)
  - Thread 1 writes best\_cost (50)
  - Thread 2 writes best\_cost (75)
- The value 75 does not “seem right” because it would not arise in a sequential execution of the same algorithm

# General problem

- The code in the previous example is called a **critical section**
  - Several threads may try to execute code in critical section but only one should succeed at a time
- Problem arises very often when writing threaded code
  - Thread A want to read and write one or more variables in critical section
  - While it is doing that, other threads should be excluded from accessing those variables
- Solution: lock
  - Threads compete for “acquiring” lock
  - Pthreads implementation guarantees that only one thread will succeed in acquiring lock
  - Successful thread enters critical section, performs its activity
  - When critical section is done, lock is “released”

# Mutex in Pthreads

- The Pthreads API provides the following functions for handling mutex-locks:

- Lock creation

```
int pthread_mutex_init (
    pthread_mutex_t *mutex_lock,
    const pthread_mutexattr_t *lock_attr);
```

- Acquiring lock

```
int pthread_mutex_lock (
    pthread_mutex_t *mutex_lock);
```

- Releasing lock

```
int pthread_mutex_unlock (
    pthread_mutex_t *mutex_lock);
```

# Implementation (see next time)

- Lock is implemented by
  - variable with two states: *available* or *not\_available*
  - queue that can hold ids of threads waiting for the lock
- Lock acquire:
  - If state of lock is *available*, its state is changed to *not\_available*, and control returns to application program
  - If state of lock is *not\_available*, thread-id is queued up at the lock, and control returns to application program only when lock is acquired by that thread
  - Key invariant: once a thread tries to acquire lock, control returns to thread only after lock has been awarded to that thread
- Lock release:
  - next thread in queue is informed it has acquired lock, and it can proceed
- “Fairness”: any thread that wants to acquire a lock can succeed ultimately even if other threads want to acquire the lock an unbounded number of times

# Correct Mutual Exclusion

- We can now write our previously incorrect critical section as:

```
pthread_mutex_t minimum_value_lock;  
...  
main( ) {  
    ...  
    pthread_mutex_init(&minimum_value_lock, NULL);  
    ...  
}  
void *find_min(void *list_ptr) {  
    ...  
    pthread_mutex_lock(&minimum_value_lock);  
    if (my_min < minimum_value)  
        minimum_value = my_min;           critical section  
    /* and unlock the mutex */  
    pthread_mutex_unlock(&minimum_value_lock);  
}
```

# Critical sections

- For performance, it is important to keep critical sections as small as possible
- While one thread is within critical section, all other threads that want to enter the critical section are blocked
- It is up to the programmer to ensure that locks are used correctly to protect variables in critical sections

Thread A

lock(l)

X:= ..X..

unlock(l)

Thread B

lock(l)

X:= ..X..

unlock(l)

Thread C

X:= ...X

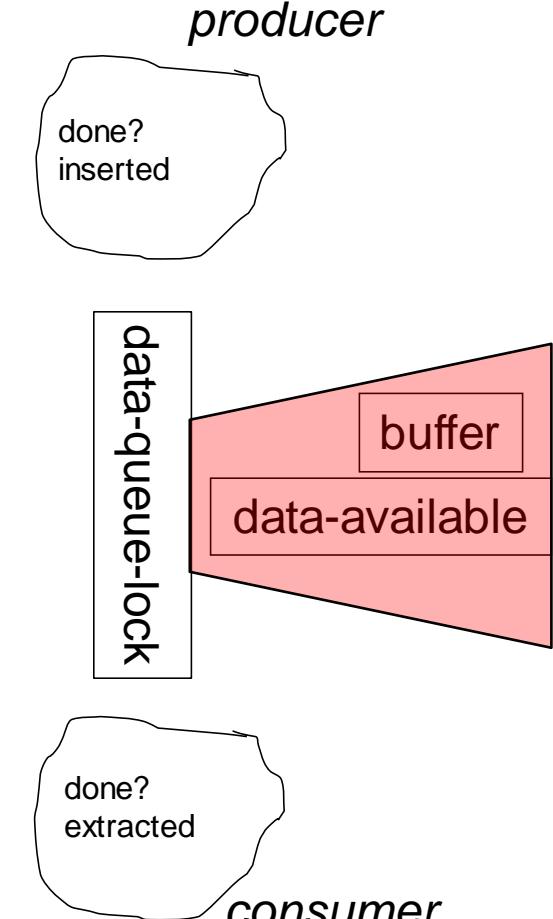
This program may fail to execute correctly because  
programmer forgot to use locks in Thread C

# Producer-Consumer Using Locks

- Two threads
  - Producer: produces data
  - Consumer: consumes data
- Shared buffer is used to communicate data from producer to consumer
  - Buffer can contain one data value (in this example)
  - Flag is associated with buffer to indicate buffer has valid data
- Consumer must not read data from buffer unless there is valid data
- Producer must not overwrite data in buffer before it is read by consumer

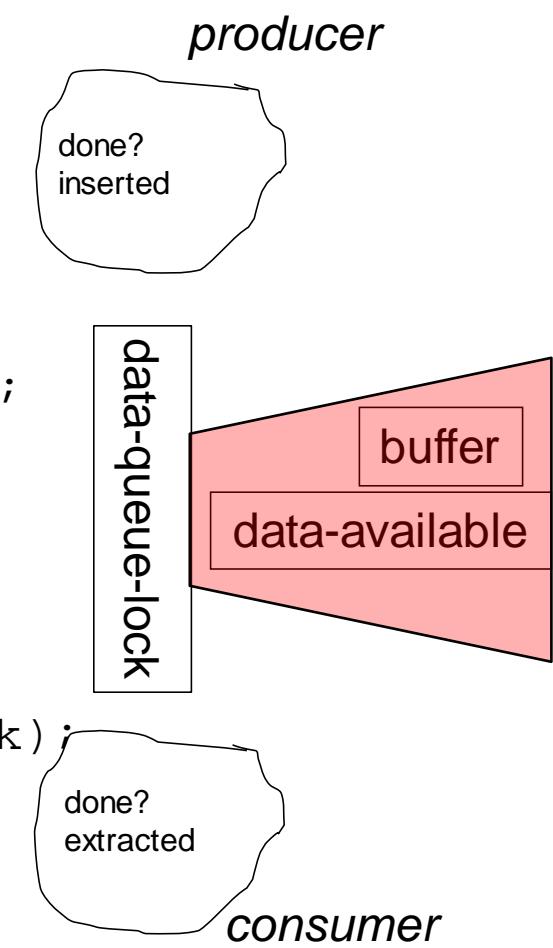
# Producer-Consumer Using Locks

```
pthread_mutex_t data_queue_lock;
int data_available; //1 if buffer is full
...
main() {
    ...
    data_available = 0;
    pthread_mutex_init(&data_queue_lock, NULL);
    ...
}
void *producer(void *producer_thread_data) {
    ...
    while (!done()) {
        create_data(&my_data);
        inserted = 0;
        while (inserted == 0) {
            pthread_mutex_lock(&data_queue_lock);
            if (data_available == 0) {
                insert_data(my_data);
                data_available = 1;
                inserted = 1;
            }
            pthread_mutex_unlock(&data_queue_lock);
        }
    }
}
```



# Producer-Consumer Using Locks

```
void *consumer(void *consumer_thread_data) {  
    int extracted;  
    struct data my_data;  
    /* local data structure declarations */  
    while (!done()) {  
        extracted = 0;  
        while (extracted == 0) {  
            pthread_mutex_lock(&data_queue_lock);  
            if (data_available == 1) {  
                extract_data(&my_data);  
                data_available = 0;  
                extracted = 1;  
            }  
            pthread_mutex_unlock(&data_queue_lock);  
        }  
        process_data(my_data);  
    }  
}
```



# Types of Mutexes

- Pthreads supports three types of mutexes - normal, recursive, and error-check.
- A normal mutex deadlocks if a thread that already has a lock tries a second lock on it.
- A recursive mutex allows a single thread to lock a mutex as many times as it wants. It simply increments a count on the number of locks. A lock is relinquished by a thread when the count becomes zero.
- An error check mutex reports an error when a thread with a lock tries to lock it again (as opposed to deadlocking in the first case, or granting the lock, as in the second case).
- The type of the mutex can be set in the attributes object before it is passed at time of initialization.

# Reducing lock overhead

- Another kind of lock: trylock.

```
int pthread_mutex_trylock (  
    pthread_mutex_t *mutex_lock);
```

- If lock is available, acquire it; otherwise, return a “busy” error code (EBUSY)
- Faster than `pthread_mutex_lock` on typical systems since it does not have to deal with queues associated with locks for multiple threads waiting on the lock.

# Alleviating Locking Overhead (Example)

```
/* Finding k matches in a list */
void *find_entries(void *start_pointer) {
    /* This is the thread function */
    struct database_record *next_record;
    int count;
    current_pointer = start_pointer;
    do {
        next_record = find_next_entry(current_pointer);
        count = output_record(next_record);
    } while (count < requested_number_of_records);
}
int output_record(struct database_record *record_ptr) {
    int count;
    pthread_mutex_lock(&output_count_lock);
    output_count++;
    count = output_count;
    pthread_mutex_unlock(&output_count_lock);
    if (count <= requested_number_of_records)
        print_record(record_ptr);
    return (count);
}
```

# Alleviating Locking Overhead (Example)

```
/* rewritten output_record function */
int output_record(struct database_record
*record_ptr) {
    int count;
    int lock_status;
    lock_status=pthread_mutex_trylock(&output_count_lock);
    if (lock_status == EBUSY) {
        insert_into_local_list(record_ptr);
        return(0);
    }
    else {
        count = output_count;
        output_count += number_on_local_list + 1;
        pthread_mutex_unlock(&output_count_lock);
        print_records(record_ptr, local_list,
                      requested_number_of_records - count);
        return(count + number_on_local_list + 1);
    }
}
```

# Problems with locks

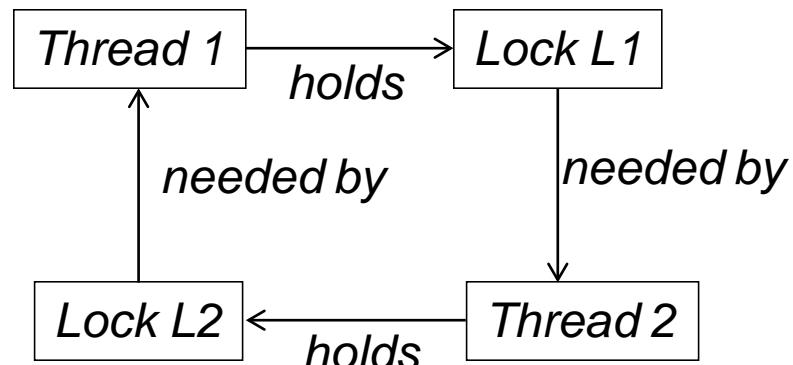
- Locks are most dangerous when a thread needs to acquire multiple locks before releasing locks
- Two main problems:
  - deadlock
  - livelock
- Deadlock:
  - Threads A and B need locks L1 and L2
  - Thread A acquires L1 and wants L2
  - Thread B acquires L2 and wants L1
  - In general, there will be a cycle of threads in which each thread holds some locks and is waiting for locks held by other threads in the cycle
- Livelock:
  - may arise in some solutions to deadlock

# Deadlock

- Code snippet shows example of possible deadlock
- Subtle point:
  - deadlock may happen in some executions and not in others!
- “Deadly embrace”: Dijkstra
- How do we ensure deadlocks cannot occur?

*Thread 1:*  
...  
lock(L1);  
lock(L2);  
....

*Thread 2:*  
...  
lock(L2);  
lock(L1);  
...



# Deadlock: four conditions

- Mutual exclusion:
  - thread has exclusive control over resource it acquires
- Hold-and-wait:
  - thread does not release resource it holds if it is waiting for another resource
- No pre-emption:
  - No external agency forces a thread to release resources if thread is waiting for another resource
- Circular wait:
  - There is a cycle of threads such that each thread holds one or more resources needed by the next thread in the cycle

You prevent deadlocks by ensuring that one or more of these conditions cannot arise in your program.

# Prevent circular wait

- Assign a logical total order to locks
  - (eg) name them L1,L2,L3,...
- Ensure that threads will never try to acquire a lower numbered lock while holding a higher numbered lock
  - (eg) if thread owns L3, it can try to acquire L4, L5, L6,... but it cannot try to acquire locks L1 or L2 (unless it already owns them and locks are re-entrant)
- Useful software engineering principle when you have control over the entire code base and you know what locks are required where
- However
  - easy to make mistakes
  - tension with encapsulation:
    - requires detailed knowledge of entire code base

# Prevent hold-and-wait

- Try to acquire all locks atomically
- One implementation:
  - single global lock to get permission to acquire locks you need
- Problem:
  - not scalable
  - conflicts with modularity and encapsulation
- You might encounter a hidden version of this problem if thread has to enter the kernel to perform some function like storage allocation
  - kernel lock is like the global-lock in our example

```
...
lock(global-lock);
lock(l1);
lock(l2);
unlock(global-lock);
...
```

# Self-preemption

- Coding discipline:
  - Use only try-locks
  - If a thread cannot acquire a lock while it is holding other locks, it releases all locks it holds and tries again
  - Variation: OS or some other agency steps in and preempts a thread
- Problems:
  - Encapsulation
  - Livelock: threads can keep on acquiring and releasing locks without making progress because no thread ever gets all the locks it needs
  - One solution to livelock: (Ethernet) backoff: thread does not retry until some randomly chosen amount of time has passed

```
loop:  
//start of lock acquires  
....  
if (trylock(Lj) == EBUSY) {  
//unlock all locks you hold  
    goto loop;  
}  
....  
endloop:  
  
//compute with resources  
//release locks
```

# Lock-free synchronization

- Use more powerful hardware instructions that perform atomic computations on variables
  - no notion of “holding” resources like locks
  - these atomic computations are enough for many applications but in general, they need to be composed and this can be tricky
- Example: **CompareAndSwap** instruction

```
int CompareAndSwap(int *address, int expected, int new)
    if (*address == expected) {
        *address = new;
        return SUCCESS;
    }
    else return FAIL;
```

```
void AtomicIncrement(int *value; int amount) {
    do {int old = *value;
        } while (CompareAndSwap(value,old,old+amount) == FAIL)
```

# Controlling Thread and Synchronization Attributes

- The Pthreads API allows a programmer to change the default attributes of entities using *attributes objects*.
- An attributes object is a data-structure that describes entity (thread, mutex, condition variable) properties.
- Once these properties are set, the attributes object can be passed to the method initializing the entity.
- Enhances modularity, readability, and ease of modification.

# Attributes Objects for Threads

- Use `pthread_attr_init` to create an attributes object.
- Individual properties associated with the attributes object can be changed using the following functions:

`pthread_attr_setdetachstate`,

`pthread_attr_setguardsize_np`,

`pthread_attr_setstacksize`,

`pthread_attr_setinheritsched`,

`pthread_attr_setschedpolicy`, and

`pthread_attr_setschedparam`

# Attributes Objects for Mutexes

- Initialize the attrributes object using function:  
`pthread_mutexattr_init.`
- The function `pthread_mutexattr_settype_np` can be used for setting the type of mutex specified by the mutex attributes object.  
`pthread_mutexattr_settype_np (`  
    `pthread_mutexattr_t *attr,`  
    `int type);`
- Here, `type` specifies the type of the mutex and can take one of:
  - `PTHREAD_MUTEX_NORMAL_NP`
  - `PTHREAD_MUTEX_RECURSIVE_NP`
  - `PTHREAD_MUTEX_ERRORCHECK_NP`

# Types of threads

- Thread implementations:
  - User-level threads:
    - Implemented by user-level runtime library
    - OS is unaware of threads
    - Portable, thread scheduling can be tuned to application requirements
    - Problem: cannot leverage multiprocessors, entire process blocks when one thread blocks
  - Kernel-level threads:
    - OS is aware of each thread and schedules them
    - Thread operations are performed by OS
    - Can leverage multiprocessors
    - Problem: higher overhead, usually not quite as portable
  - Hybrid-level threads: Solaris
    - OS provides some number of kernel level threads, and each of these can create multiple user-level threads
    - Problem: complexity

*global main*

*extern printf*  
*extern pthread\_create*  
*extern pthread\_exit*  
*extern pthread\_join*

*section .data*

*align 4*  
sLock: dd 0 ; The lock, values are:  
; 0 unlocked  
; 1 locked

tID1: dd 0

tID2: dd 0

fmtStr1: db "In thread %d with ID: %02x", 0x0A, 0

fmtStr2: db "Result %d", 0x0A, 0

*section .bss*

*align 4*  
result: resd 1

section .text

main:

; Using main since we are using gcc to link

; Call pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr,  
; void \*(\*start\_routine) (void \*), void \*arg);

push dword 0 ; Arg Four: argument pointer  
push thread1 ; Arg Three: Address of routine  
push dword 0 ; Arg Two: Attributes  
push tID1 ; Arg One: pointer to the thread ID  
call pthread\_create

push dword 0 ; Arg Four: argument pointer  
push thread2 ; Arg Three: Address of routine  
push dword 0 ; Arg Two: Attributes  
push tID2 ; Arg One: pointer to the thread ID  
call pthread\_create

; Call int pthread\_join(pthread\_t thread, void \*\*retval) ;  
;

push dword 0 ; Arg Two: retval  
push dword [tID1] ; Arg One: Thread ID to wait on  
call pthread\_join  
push dword 0 ; Arg Two: retval  
push dword [tID2] ; Arg One: Thread ID to wait on  
call pthread\_join

push dword [result]  
push dword fmtStr2  
call printf  
add esp, 8 ; Pop stack 2 times 4 bytes

call exit

*thread1:*

```
pause
push      dword [tID1]
push      dword 1
push      dword fmtStr1
call      printf
add      esp, 12           ; Pop stack 3 times 4 bytes

call      spinLock

mov      [result], dword 1
call      spinUnlock

push      dword 0           ; Arg one: retval
call      pthread_exit
```

*thread2:*

```
pause
push      dword [tID2]
push      dword 2
push      dword fmtStr1
call      printf
add      esp, 12           ; Pop stack 3 times 4 bytes

call      spinLock

mov      [result], dword 2
call      spinUnlock

push      dword 0           ; Arg one: retval
call      pthread_exit
```

*spinLock:*

```
push    ebp
mov     ebp, esp
mov     edx, 1          ; Value to set sLock to
spin:  mov     eax, [sLock] ; Check sLock
       test    eax, eax   ; If it was zero, maybe we have the lock
       jnz    spin      ; If not try again
       ;
       ; Attempt atomic compare and exchange:
       ; if (sLock == eax):
       ;     sLock           <- edx
       ;     zero flag      <- 1
       ; else:
       ;     eax            <- edx
       ;     zero flag      <- 0
       ;
       ; If sLock is still zero then it will have the same value as eax and
       ; sLock will be set to edx which is one and therefore we acquire the
       ; lock. If the lock was acquire between the first test and the
       ; cmpxchg then eax will not be zero and we will spin again.
       ;
       lock    cmpxchg [sLock], edx ;eax is implicit operand
       test    eax, eax
       jnz    spin
       pop    ebp
       ret
```

*spinUnlock:*

```
push    ebp
mov     ebp, esp
mov     eax, 0
xchg    eax, [sLock]
pop    ebp
ret
```

*exit:*

```
; ; Call exit(3) syscall  
; void exit(int status)  
;  
mov    ebx, 0      ; Arg one: the status  
mov    eax, 1      ; Syscall number:  
int    0x80
```

# OpenMP: a Standard for Directive-Based Parallel Programming

- OpenMP is a directive-based API that can be used with FORTRAN, C, and C++ for programming shared address space machines.
- OpenMP directives provide support for concurrency, synchronization, and data handling while obviating the need for explicitly setting up mutexes, condition variables, data scope, and initialization.

# OpenMP Programming Model

- OpenMP directives in C and C++ are based on the `#pragma` compiler directives.
- A directive consists of a directive name followed by clauses.

```
#pragma omp directive [clause list]
```
- OpenMP programs execute serially until they encounter the `parallel` directive, which creates a group of threads.

```
#pragma omp parallel [clause list]
/* structured block */
```
- The main thread that encounters the `parallel` directive becomes the *master* of this group of threads and is assigned the thread id 0 within the group.

# OpenMP Programming Model

- The clause list is used to specify conditional parallelization, number of threads, and data handling.
  - **Conditional Parallelization:** The clause `if (scalar expression)` determines whether the parallel construct results in creation of threads.
  - **Degree of Concurrency:** The clause `num_threads(integer expression)` specifies the number of threads that are created.
  - **Data Handling:** The clause `private (variable list)` indicates variables local to each thread. The clause `firstprivate (variable list)` is similar to the `private`, except values of variables are initialized to corresponding values before the parallel directive. The clause `shared (variable list)` indicates that variables are shared across all the threads.

# OpenMP Programming Model

```
int a, b;
main() {
    // serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    {
        // parallel segment
    }
    // rest of serial segment
```

Sample OpenMP program

|                                      |                                                                                                                                                                                                                                               |                                    |
|--------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------|
|                                      | <pre>int a, b; main() {     // serial segment     for (i = 0; i &lt; 8; i++)         pthread_create (....., internal_thread_fn_name, ...);     for (i = 0; i &lt; 8; i++)         pthread_join (.....);     // rest of serial segment }</pre> |                                    |
| Code inserted by the OpenMP compiler |                                                                                                                                                                                                                                               | Corresponding Pthreads translation |

- A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

# OpenMP Programming Model

```
#pragma omp parallel if (is_parallel== 1) num_threads(8) \
    private (a) shared (b) firstprivate(c) {
    /* structured block */
}
```

- If the value of the variable `is_parallel` equals one, eight threads are created.
- Each of these threads gets private copies of variables `a` and `c`, and shares a single value of variable `b`.
- The value of each copy of `c` is initialized to the value of `c` before the parallel directive.
- The default state of a variable is specified by the clause `default (shared)` or `default (none)`.

# Reduction Clause in OpenMP

- The reduction clause specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit.
- The usage of the reduction clause is `reduction(operator: variable list)`.
- The variables in the list are implicitly specified as being private to threads.
- The operator can be one of `+`, `*`, `-`, `&`, `|`, `^`, `&&`, and `||`.

```
#pragma omp parallel reduction(+: sum) num_threads(8) {  
/* compute local sums here */  
}  
/*sum here contains sum of all local instances of sums */
```

# OpenMP Programming: Example

```
/* ****
An OpenMP version of a threaded program to compute PI.
**** */
#pragma omp parallel default(private) shared (npoints) \
    reduction(+: sum) num_threads(8)
{
    num_threads = omp_get_num_threads();
    sample_points_per_thread = npoints / num_threads;
    sum = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            sum++;
    }
}
```

# Specifying Concurrent Tasks in OpenMP

- The `parallel` directive can be used in conjunction with other directives to specify concurrency across iterations and tasks.
- OpenMP provides two directives - `for` and `sections` - to specify concurrent iterations and tasks.
- The `for` directive is used to split parallel iteration spaces across threads. The general form of a `for` directive is as follows:

```
#pragma omp for [clause list]  
/* for loop */
```

- The clauses that can be used in this context are: `private`, `firstprivate`, `lastprivate`, `reduction`, `schedule`, `nowait`, and `ordered`.

# Specifying Concurrent Tasks in OpenMP: Example

```
#pragma omp parallel default(private) shared (npoints) \
    reduction(+: sum) num_threads(8)
{
    sum = 0;
    #pragma omp for
    for (i = 0; i < npoints; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            sum++;
    }
}
```

# Assigning Iterations to Threads

- The `schedule` clause of the `for` directive deals with the assignment of iterations to threads.
- The general form of the `schedule` directive is `schedule(scheduling_class[, parameter])`.
- OpenMP supports four scheduling classes: `static`, `dynamic`, `guided`, and `runtime`.

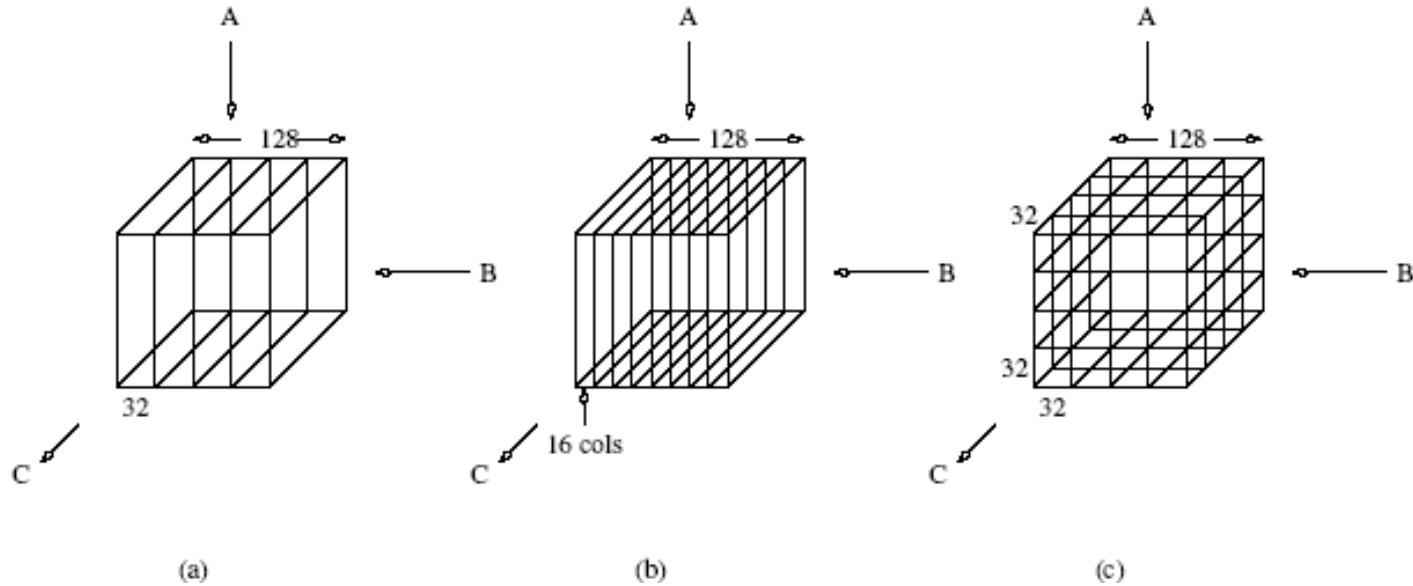
# Assigning Iterations to Threads:

## Example

```
/* static scheduling of matrix multiplication loops */
#pragma omp parallel default(private) shared (a, b, c, dim) \
num_threads(4)
#pragma omp for schedule(static)
for (i = 0; i < dim; i++) {
    for (j = 0; j < dim; j++) {
        c(i,j) = 0;
        for (k = 0; k < dim; k++) {
            c(i,j) += a(i, k) * b(k, j);
        }
    }
}
```

# Assigning Iterations to Threads:

## Example



- Three different schedules using the static scheduling class of OpenMP.

# Parallel For Loops

- Often, it is desirable to have a sequence of `for`-directives within a parallel construct that do not execute an implicit barrier at the end of each `for` directive.
- OpenMP provides a clause - `nowait`, which can be used with a `for` directive.

# Parallel For Loops: Example

```
#pragma omp parallel
{
    #pragma omp for nowait
    for (i = 0; i < nmax; i++)
        if (isEqual(name, current_list[i])
            processCurrentName(name);

    #pragma omp for
    for (i = 0; i < mmax; i++)
        if (isEqual(name, past_list[i])
            processPastName(name);
}
```

# The sections Directive

- OpenMP supports non-iterative parallel task assignment using the sections directive.
- The general form of the sections directive is as follows:

```
#pragma omp sections [clause list]
{
    [#pragma omp section
        /* structured block */
    ]
    [#pragma omp section
        /* structured block */
    ]
    ...
}
```

# The sections Directive: Example

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA( );
        }
        #pragma omp section
        {
            taskB( );
        }
        #pragma omp section
        {
            taskC( );
        }
    }
}
```

# Nesting parallel Directives

- Nested parallelism can be enabled using the OMP\_NESTED environment variable.
- If the OMP\_NESTED environment variable is set to TRUE, nested parallelism is enabled.
- In this case, each parallel directive creates a new team of threads.

# Synchronization Constructs in OpenMP

- OpenMP provides a variety of synchronization constructs:

```
#pragma omp barrier
```

```
#pragma omp single [clause list]  
    structured block
```

```
#pragma omp master  
    structured block
```

```
#pragma omp critical [(name)]  
    structured block
```

```
#pragma omp ordered  
    structured block
```

# OpenMP Library Functions

- In addition to directives, OpenMP also supports a number of functions that allow a programmer to control the execution of threaded programs.

```
/* thread and processor count */
void omp_set_num_threads (int
    num_threads);
int omp_get_num_threads ( );
int omp_get_max_threads ( );
int omp_get_thread_num ( );
int omp_get_num_procs ( );
int omp_in_parallel( );
```

# OpenMP Library Functions

```
/* controlling and monitoring thread creation */
void omp_set_dynamic (int dynamic_threads);
int omp_get_dynamic ();
void omp_set_nested (int nested);
int omp_get_nested ();
/* mutual exclusion */
void omp_init_lock (omp_lock_t *lock);
void omp_destroy_lock (omp_lock_t *lock);
void omp_set_lock (omp_lock_t *lock);
void omp_unset_lock (omp_lock_t *lock);
int omp_test_lock (omp_lock_t *lock);
```

- In addition, all lock routines also have a nested lock counterpart
- for recursive mutexes.

# Environment Variables in OpenMP

- OMP\_NUM\_THREADS: This environment variable specifies the default number of threads created upon entering a parallel region.
- OMP\_SET\_DYNAMIC: Determines if the number of threads can be dynamically changed.
- OMP\_NESTED: Turns on nested parallelism.
- OMP\_SCHEDULE: Scheduling of for-loops if the clause specifies runtime

# Explicit Threads versus Directive Based Programming

- Directives layered on top of threads facilitate a variety of thread-related tasks.
- A programmer is rid of the tasks of initializing attributes objects, setting up arguments to threads, partitioning iteration spaces, etc.
- There are some drawbacks to using directives as well.
- An artifact of explicit threading is that data exchange is more apparent. This helps in alleviating some of the overheads from data movement, false sharing, and contention.
- Explicit threading also provides a richer API in the form of condition waits, locks of different types, and increased flexibility for building composite synchronization operations.
- Finally, since explicit threading is used more widely than OpenMP, tools and support for Pthreads programs are easier to find.

# Memory Consistency Models

# Outline

- Need for memory consistency models
- Sequential consistency model
- Relaxed memory models
  - weak consistency model
  - release consistency model
- Conclusions

# Recall: uniprocessor execution

- Processors reorder operations to improve performance
- Constraint on reordering: must respect dependences
  - data dependences must be respected: in particular, loads/stores to a given memory address must be executed in program order
  - control dependences must be respected
- Reorderings can be performed either by compiler or processor

# Permitted memory-op reorderings

- Stores to different memory locations can be performed out of program order

store v1, data  
store b1, flag

↔

store b1, flag  
store v1, data

- Loads from different memory locations can be performed out of program order

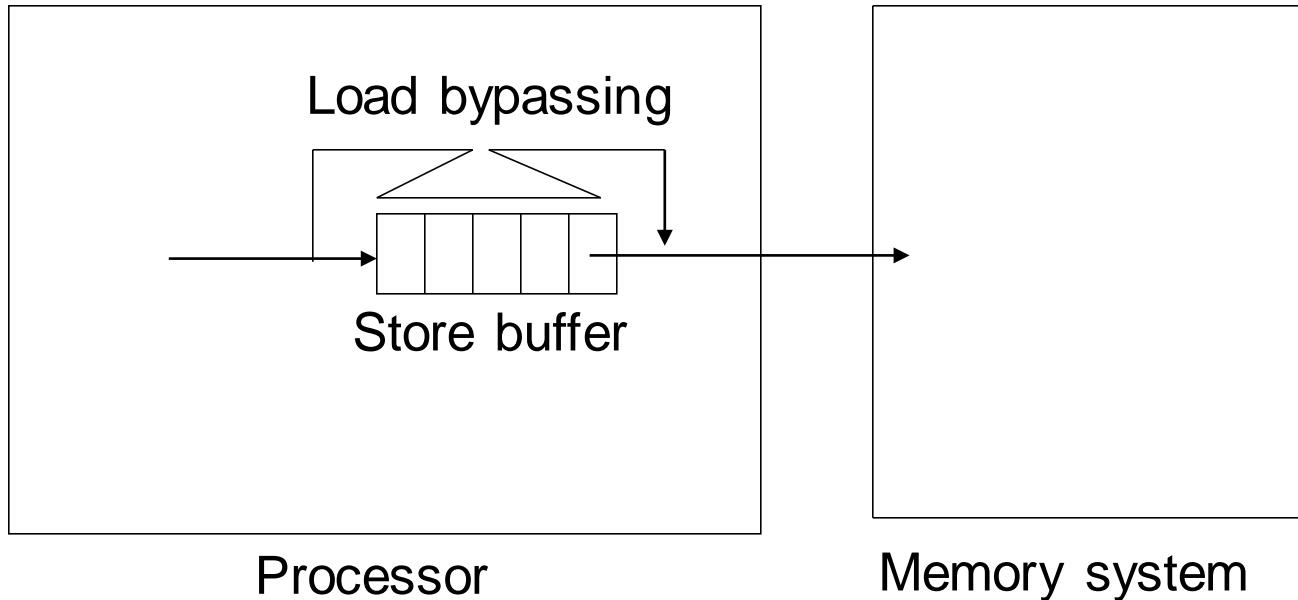
load flag, r1  
load data, r2

↔

load data,r2  
load flag, r1

- Load and store to different memory locations can be performed out of program order

# Example of hardware reordering



- Store buffer holds store operations that need to be sent to memory
- Loads are higher priority operations than stores since their results are needed to keep processor busy, so they bypass the store buffer
- Load address is checked against addresses in store buffer, so store buffer satisfies load if there is an address match
- Result: load can bypass stores to other addresses

# Key issue

- In single-threaded programs, reordering of instructions does not affect the output of the program
  - may improve performance but does not change the semantics
- In shared-memory programs, reordering of instructions executed by a thread may change the output of the program

# Example (I)

*Code:*

*Initially A = Flag = 0*

P1

A = 23;  
Flag = 1;

P2

while (Flag != 1) {}  
... = A;

Idea:

- P1 writes data into A and sets Flag to tell P2 that data value can be read from A.
- P2 waits till Flag is set and then reads data from A.

# Execution Sequence for (I)

Code:

*Initially A = Flag = 0*

P1  
A = 23;  
Flag = 1;

P2  
while (Flag != 1) {}  
... = A;

*Possible execution sequence on each processor:*

P1  
Write A 23  
Write Flag 1

P2  
Read Flag //get 0  
.....  
Read Flag //get 1  
Read A //what do you get?



Problem: If the two writes on processor P1 can be reordered, it is possible for processor P2 to read 0 from variable A.  
Can happen on most modern processors.

# Example II

Code: (*like Dekker's algorithm*)

*Initially Flag1 = Flag2 = 0*

P1

Flag1 = 1;

If (Flag2 == 0)

*critical section*

P2

Flag2 = 1;

If (Flag1 == 0)

*critical section*

*Possible execution sequence on each processor:*

P1

Write Flag1, 1

Read Flag2 //get 0

P2

Write Flag2, 1

Read Flag1 //what do you get?

# Execution sequence for (II)

Code: (like Dekker's algorithm)

Initially  $\text{Flag1} = \text{Flag2} = 0$

P1                            P2  
 $\text{Flag1} = 1;$                      $\text{Flag2} = 1;$   
If ( $\text{Flag2} == 0$ )               If ( $\text{Flag1} == 0$ )  
    *critical section*                *critical section*

Possible execution sequence on each processor:

P1                            P2  
Write  $\text{Flag1}, 1$                Write  $\text{Flag2}, 1$   
Read  $\text{Flag2} //\text{get } 0$             Read  $\text{Flag1}, ??$



Most people would say that P2 will read 1 as the value of Flag1.

Since P1 reads 0 as the value of Flag2, P1's read of Flag2 must happen before P2 writes to Flag2. Intuitively, we would expect P1's write of Flag to happen before P2's read of Flag1.

However, this is true only if reads and writes on the same processor to different locations are not reordered by the compiler or the hardware.

Unfortunately, this is very common on most processors (store-buffers with load-bypassing).

# Lessons

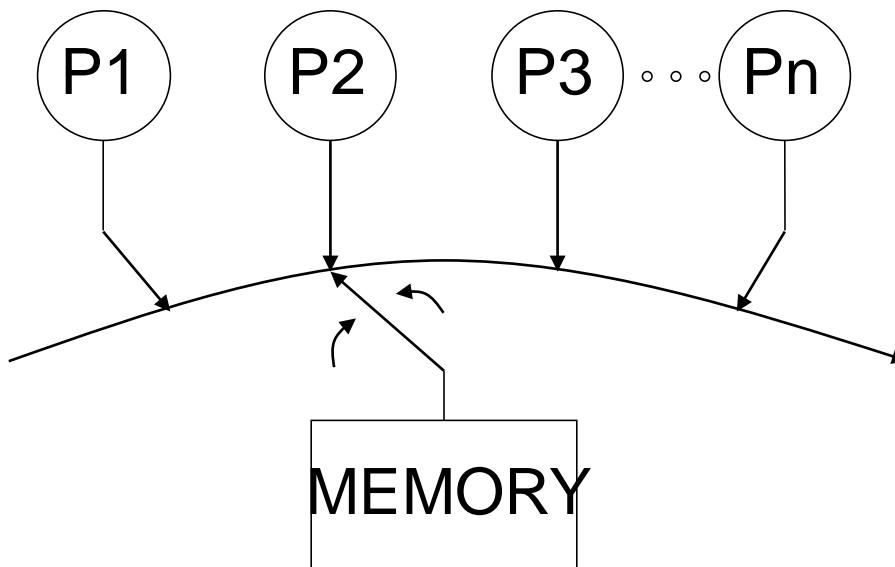
- Uniprocessors can reorder instructions subject only to control and data dependence constraints
- These constraints are not sufficient in shared-memory context
  - simple parallel programs may produce counter-intuitive results
- Question: what constraints must we put on uniprocessor instruction reordering so that
  - shared-memory programming is intuitive
  - but we do not lose uniprocessor performance?
- Many answers to this question
  - answer is called **memory consistency model** supported by the processor

# Consistency models

- Consistency models are **not** about memory operations from different processors.
- Consistency models are **not** about dependent memory operations in a single processor's instruction stream (these are respected even by processors that reorder instructions).
- Consistency models are all about ordering constraints on independent memory operations in a single processor's instruction stream that have **some high-level dependence** (such as flags guarding data) that should be respected to obtain intuitively reasonable results.

# Simplest Memory Consistency Model

- Sequential consistency (SC) [Lamport]
  - processor is not allowed to reorder reads and writes to global memory



# Sequential Consistency

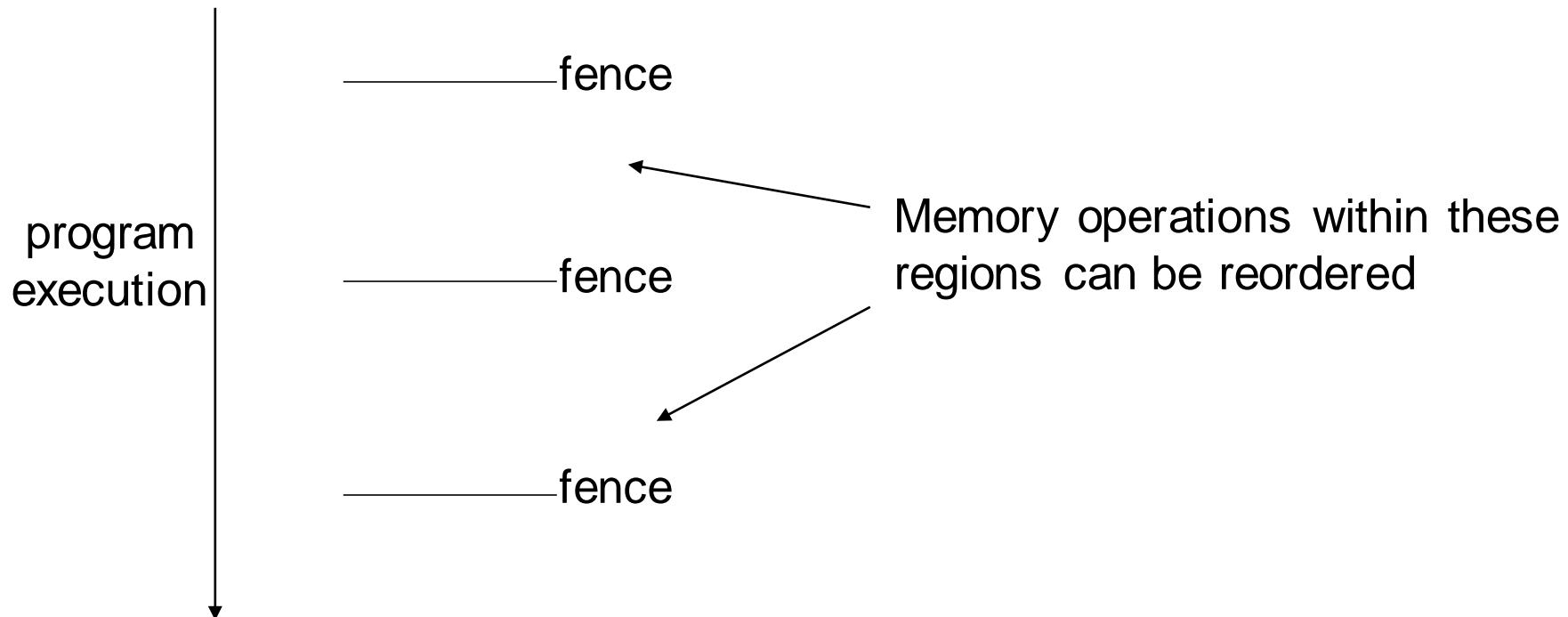
- SC constrains all memory operations:
  - Write → Read
  - Write → Write
  - Read → Read, Write
- Simple model for reasoning about parallel programs
  - You can verify that the examples considered earlier work correctly under sequential consistency.
  - However, this simplicity comes at the cost of uniprocessor performance.
  - Question: how do we reconcile sequential consistency model with the demands of performance?

# Relaxed consistency model:

## Weak consistency

- Programmer specifies regions within which global memory operations can be reordered
- Processor has **fence instruction**:
  - all data operations **before** fence in program order must complete before fence is executed
  - all data operations **after** fence in program order must wait for fence to complete
  - fences are performed in program order
- Implementation of fence:
  - processor has counter that is incremented when data op is issued, and decremented when data op is completed
- Example: PowerPC has **SYNC** instruction
- Language constructs:
  - OpenMP: flush
  - All synchronization operations like lock and unlock act like a fence

# Weak ordering picture



# Example (I) revisited

Code:

*Initially A = Flag = 0*

P1

```
A = 23;  
flush;  
Flag = 1;
```

P2

```
while (Flag != 1) {}  
... = A;
```

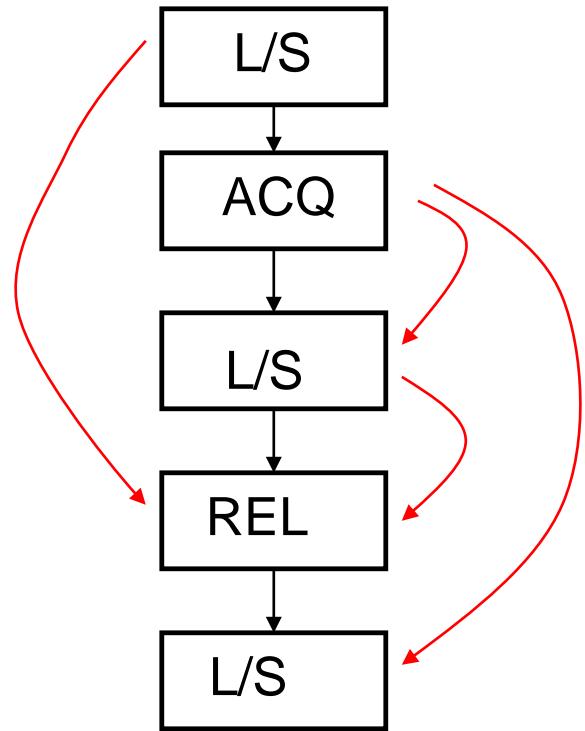
Execution:

- P1 writes data into A
- Flush waits till write to A is completed
- P1 then writes data to Flag
- Therefore, if P2 sees Flag = 1, it is guaranteed that it will read the correct value of A even if memory operations in P1 before flush and memory operations after flush are reordered by the hardware or compiler.
- Question: does P2 need a flush between the two statements?

# Another relaxed model: release consistency

- Further relaxation of weak consistency
- Synchronization accesses are divided into
  - **Acquires**: operations like lock
  - **Release**: operations like unlock
- Semantics of acquire:
  - Acquire must complete before all following memory accesses
- Semantics of release:
  - all memory operations before release are complete
- However,
  - acquire does not wait for accesses preceding it
  - accesses after release in program order do not have to wait for release
    - operations which follow release and which need to wait must be protected by an acquire

# Example



Which operations can be overlapped?

# Implementations on Current Processors

|                | Loads Reordered After Loads? | Loads Reordered After Stores? | Stores Reordered After Stores? | Stores Reordered After Loads? | Atomic Instructions Reordered With Loads? | Atomic Instructions Reordered With Stores? | Dependent Loads Reordered? | Incoherent Instruction Cache Pipeline? |
|----------------|------------------------------|-------------------------------|--------------------------------|-------------------------------|-------------------------------------------|--------------------------------------------|----------------------------|----------------------------------------|
| Alpha          | Y                            | Y                             | Y                              | Y                             | Y                                         | Y                                          | Y                          | Y                                      |
| AMD64          | Y                            |                               |                                | Y                             |                                           |                                            |                            |                                        |
| IA-64          | Y                            | Y                             | Y                              | Y                             | Y                                         | Y                                          |                            | Y                                      |
| (PA-RISC)      | Y                            | Y                             | Y                              | Y                             |                                           |                                            |                            |                                        |
| PA-RISC CPUs   |                              |                               |                                |                               |                                           |                                            |                            |                                        |
| POWER          | Y                            | Y                             | Y                              | Y                             | Y                                         | Y                                          |                            | Y                                      |
| SPARC RMO      | Y                            | Y                             | Y                              | Y                             | Y                                         | Y                                          |                            | Y                                      |
| (SPARC PSO)    |                              |                               | Y                              | Y                             |                                           | Y                                          |                            | Y                                      |
| SPARC TSO      |                              |                               |                                | Y                             |                                           |                                            |                            | Y                                      |
| x86            | Y                            | Y                             |                                | Y                             |                                           |                                            |                            | Y                                      |
| (x86 OO Store) | Y                            | Y                             | Y                              | Y                             |                                           |                                            |                            | Y                                      |
| zSeries        |                              |                               |                                | Y                             |                                           |                                            |                            | Y                                      |

# Comments

- In the literature, there are a large number of other consistency models
  - processor consistency
  - total store order (TSO)
  - ....
- It is important to remember that these are concerned with reordering of independent memory operations **within** a processor.
- Easy to come up with shared-memory programs that behave differently for each consistency model.
- Emerging consensus that weak/release consistency is adequate.

# Summary

- Two problems: memory consistency and memory coherence
- Memory consistency model
  - what instructions is compiler or hardware allowed to reorder?
  - nothing really to do with memory operations from different processors/threads
  - sequential consistency: perform global memory operations in program order
  - relaxed consistency models: all of them rely on some notion of a fence operation that demarcates regions within which reordering is permissible
- Memory coherence
  - Preserve the illusion that there is a single logical memory location corresponding to each program variable even though there may be lots of physical memory locations where the variable is stored

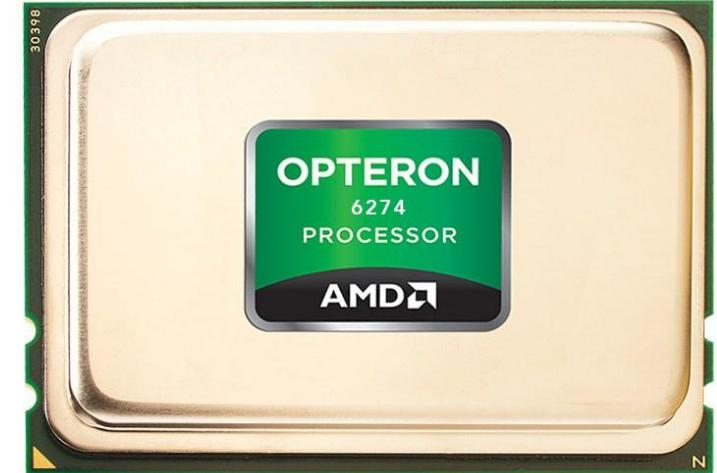
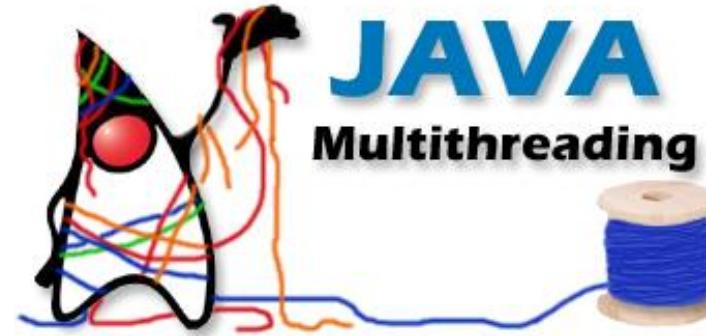
# Memory Consistency Model

---

Swarnendu Biswas  
UT Austin

# Outline

- Data races
- Memory consistency models
- Sequential Consistency
- Hardware memory models
  - TSO, PSO, Relaxed consistency
- Language memory models
  - C++, Java



# Today's Trends

# Data Race: Primary Source of Concurrency Errors

```
Object X = null;  
boolean done= false;
```

Thread T1

```
X = new Object();  
done = true;
```

Thread T2

```
while (!done) {}  
X.compute();
```

```
Object x = null;  
boolean done= false;
```

Thread T1

write

```
x = new Object();  
done = true;
```

Thread T2

read

```
while (!done) {}  
x.compute();
```

---

## Data race

**Conflicting accesses** – two threads access the same shared variable where at least one access is a write

---

**Concurrent accesses** – accesses are not ordered by synchronization operations

---

## Thread T1

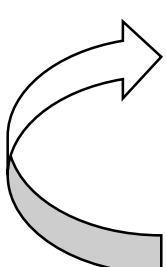
```
x = new Object();  
  
done = true;
```

## Thread T2

```
temp = done;  
  
while (!temp) {}
```

Infinite loop

## Thread T1



```
done = true;  
  
x = new Object();
```

## Thread T2

```
while (!done) {}  
x.compute();
```

NPE

# Data Races are Bad

Therac-25 accident & Northeast US Blackout &  
NASDAQ Facebook glitch

---

research highlights

## Technical Perspective **Data Races are Evil with No Exceptions**

By Sarita Adve

EXPLOITING PARALLELISM HAS become the primary means to higher performance. | racy code. Java's safety requiremer

How to miscompile programs with “benign” data races

Hans-J. Boehm  
*HP Laboratories*

# Memory Consistency Model: What Value Can a Read Return?

**TABLE 3.1:** Should r2 Always be Set to NEW?

| Core C1                                        | Core C2                                                                  | Comments                                                                      |
|------------------------------------------------|--------------------------------------------------------------------------|-------------------------------------------------------------------------------|
| S1: Store data = NEW;<br>S2: Store flag = SET; | L1: Load r1 = flag;<br>B1: if (r1 ≠ SET) goto L1;<br>L2: Load r2 = data; | /* Initially, data = 0 & flag ≠ SET */<br>/* L1 & B1 may repeat many times */ |

## How a Core Might Reorder Accesses?

- Store-store
- Load-load
- Store-load
- Load-store

# Memory Consistency Model

- Specifies the allowed behaviors of multithreaded programs executing with shared memory
  - Both at the hardware-level and at the programming-language-level
- “What values can a load return?”
  - Return the “last” write
  - Uniprocessor: program order
  - Multiprocessor: ?
- There can be multiple correct behaviors

# Memory Consistency Model

- Visibility:
  - “When does a value update become visible to others?”
- Ordering:
  - When can operations of any given thread appear out of order to another thread?

# Dekker's Algorithm

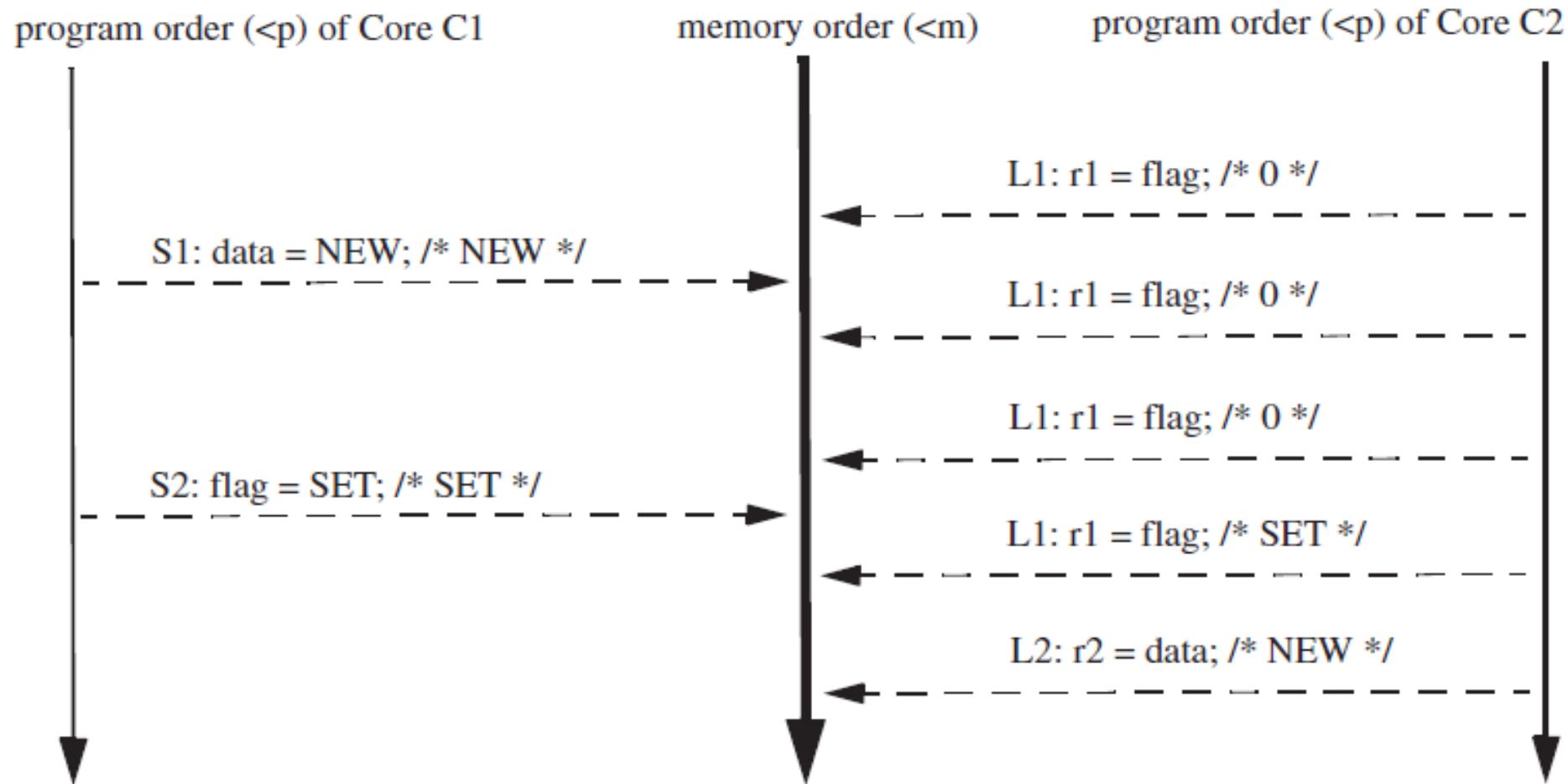
**TABLE 3.3:** Can Both r1 and r2 be Set to 0?

| Core C1                                | Core C2                                | Comments                          |
|----------------------------------------|----------------------------------------|-----------------------------------|
| S1: $x = \text{NEW};$<br>L1: $r1 = y;$ | S2: $y = \text{NEW};$<br>L2: $r2 = x;$ | /* Initially, $x = 0$ & $y = 0*/$ |

## Sequential Consistency (SC)

- Uniprocessor - operations executed in order specified by the program
- Multiprocessor - all operations executed in order, and the operations of each individual core appear in program order

# Earlier Example Under SC



## SC Rules

- **$a = b$  or  $a \neq b$** 
  - if  $L(a) <_p L(b) \Rightarrow L(a) <_m L(b)$
  - If  $L(a) <_p S(b) \Rightarrow L(a) <_m S(b)$
  - If  $S(a) <_p S(b) \Rightarrow S(a) <_m S(b)$
  - If  $S(a) <_p L(b) \Rightarrow S(a) <_m L(b)$
- Every load gets its value from the last store before it (in global memory order) to the same address

# SC Provides Write Atomicity

Initially  $A = B = 0$

P1

P2

P3

$A = 1$

if ( $A == 1$ )

$B = 1$

if ( $B == 1$ )

register1 =  $A$

# Write Atomicity

- Relaxing write atomicity violates SC

| Initially X=Y=0 |           |                             |                             |
|-----------------|-----------|-----------------------------|-----------------------------|
| T1<br>X=1       | T2<br>Y=1 | T3<br>r1=X<br>fence<br>r2=Y | T4<br>r3=Y<br>fence<br>r4=X |

r1=1, r2=0, r3=1, r4=0 violates write atomicity

## End-to-end SC

- Simple memory model that can be implemented both in hardware and in languages
- Performance
  - Naive hardware
    - Maintain program order - expensive for a write
      - E.g., write buffer can break Dekker's algorithm
    - Write atomicity
- Program semantics
  - SC does not guarantee data race freedom
  - Not a strong memory model

a++;

buffer[index]++;

# Cache Coherence

- Single writer multiple readers
  - Memory updates are passed correctly, cached copies always contain the most recent data
  - Virtually a synonym for SC
- 
- Alternate definition based on relaxed ordering
    - A write is eventually made visible to all processors
    - Writes to the **same** location appear to be seen in the same order by all processors (serialization)
    - SC - \*all\*

# Maintaining the Illusion of Write Atomicity

*Initially A = B = C = 0*

| P1     | P2     | P3                                                               | P4                                                               |
|--------|--------|------------------------------------------------------------------|------------------------------------------------------------------|
| A = 1; | A = 2; | while (B != 1) {}<br>while (C != 1) {}<br>tmp1 = A; <del>X</del> | while (B != 1) {}<br>while (C != 1) {}<br>tmp2 = A; <del>X</del> |
| B = 1; | C = 1; |                                                                  |                                                                  |

## Memory Consistency vs Cache Coherence

- Cache Coherence does not define shared memory behavior
  - Goal is to make caches invisible
- Memory consistency can use cache coherence as a “black box”

## Characterizing Hardware Memory Models

- Relax program order
  - Store → Load, Store → Store, etc.
  - Applicable to pairs of operations with different addresses
- Relax write atomicity
  - Read own write early
  - Read other's write early
    - Applicable to only cache-based systems

## Read Other's Write Early Can Violate Write Atomicity

*Initially A = B = 0*

P1

A = 1

P2

while (A != 1) ; while (B != 1) ;

B = 1;

P3

tmp = A

P1

Write, A, 1

P2

Read, A, 1

Write, B, 1

P3

Read, B, 1

Read, A, ~~1~~

# Possible Interleavings Under SC

TABLE 3.3: Can Both r1 and r2 be Set to 0?

| Core C1                     | Core C2                     | Comments                       |
|-----------------------------|-----------------------------|--------------------------------|
| S1: x = NEW;<br>L1: r1 = y; | S2: y = NEW;<br>L2: r2 = x; | /* Initially, x = 0 & y = 0 */ |

## Total Store Order (TSO)

- Allows reordering stores to loads
- Can read own write early, not other's writes

- Conjecture: widely-used x86 memory model is equivalent to TSO

## TSO Rules

- **a == b or a != b**
  - If L(a) < p L(b)  $\Rightarrow$  L(a) < m L(b)
  - If L(a) < p S(b)  $\Rightarrow$  L(a) < m S(b)
  - If S(a) < p S(b)  $\Rightarrow$  S(a) < m S(b)
  - ~~If S(a) < p L(b)  $\Rightarrow$  S(a) < m L(b) /\* Enables FIFO Write Buffer \*/~~
- Every load gets its value from the last store before it to the same address
- Needs a notion of a FENCE

## TSO Rules (...contd)

- If  $L(a) <_p FENCE \Rightarrow L(a) <_m FENCE$
- If  $S(a) <_p FENCE \Rightarrow S(a) <_m FENCE$
- If  $FENCE <_p FENCE \Rightarrow FENCE <_m FENCE$
- If  $FENCE <_p L(a) \Rightarrow FENCE <_m L(a)$
- If  $FENCE <_p S(a) \Rightarrow FENCE <_m S(a)$

- If  $S(a) <_p FENCE \Rightarrow S(a) <_m FENCE$
- If  $FENCE <_p L(a) \Rightarrow FENCE <_m L(a)$

## RMW in TSO

- Load of a RMW cannot be performed until earlier stores are performed (i.e., exited the write buffer)
- Load requires read–write coherence permissions, not just read permissions

- To guarantee atomicity, the cache controller may not relinquish coherence permission to the block between the load and the store

## Partial Store Order (PSO)

- Allows reordering of store to loads and stores to stores
- Writes to **different** locations from the same processor can be pipelined or overlapped and are allowed to reach memory or other cached copies out of program order
- Can read own write early, not other's writes

# Opportunities to Reorder Memory Operations

| TABLE 5.1: What Order Ensures r2 & r3 Always Get NEW?   |                                                                                                |                                                                                                       |
|---------------------------------------------------------|------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------|
| Core C1                                                 | Core C2                                                                                        | Comments                                                                                              |
| S1: data1 = NEW;<br>S2: data2 = NEW;<br>S3: flag = SET; | L1: r1 = flag;<br><br>B1: if (r1 ≠ SET) goto L1;<br><br>L2: r2 = data1;<br><br>L3: r3 = data2; | /* Initially, data1 & data2 = 0 & flag ≠ SET */<br><br>/* spin loop: L1 & B1 may repeat many times */ |

# Reorder Operations Within a Synchronization Block

TABLE 5.2: What Order Ensures Correct Handoff from Critical Section 1 to 2?

| Core C1                                                                                                                                                                    | Core C2                                                                                                                                                                    | Comments                                                                                                                                                                       |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| A1: acquire(lock)<br>/* Begin Critical Section 1 */<br><br>Some loads L1i interleaved<br>with some stores S1j<br><br>/* End Critical Section 1 */<br><br>R1: release(lock) | A2: acquire(lock)<br>/* Begin Critical Section 2 */<br><br>Some loads L2i interleaved<br>with some stores S2j<br><br>/* End Critical Section 2 */<br><br>R2: release(lock) | /* Arbitrary interleaving of L1i's & S1j's */<br><br>/* Handoff from critical section 1 */<br>/* To critical section 2 */<br><br>/* Arbitrary interleaving of L2i's & S2j's */ |

## Optimization Opportunities

- Non-FIFO coalescing write buffer
- Support non-blocking reads
  - Hide latency of reads
  - Use lockup-free caches and speculative execution
- Simpler support for speculation
  - Need not compare addresses of loads to coherence requests
  - For SC, need support to check whether the speculation is correct

## Relaxed Consistency Rules

- If  $L(a) <_p FENCE \Rightarrow L(a) <_m FENCE$
- If  $S(a) <_p FENCE \Rightarrow S(a) <_m FENCE$
- If  $FENCE <_p FENCE \Rightarrow FENCE <_m FENCE$
- If  $FENCE <_p L(a) \Rightarrow FENCE <_m L(a)$
- If  $FENCE <_p S(a) \Rightarrow FENCE <_m S(a)$

Maintain TSO rules for ordering two accesses to the same address only

- If  $L(a) <_p L'(a) \Rightarrow L(a) <_m L'(a)$
  - If  $L(a) <_p S(a) \Rightarrow L(a) <_m S(a)$
  - If  $S(a) <_p S'(a) \Rightarrow S(a) <_m S'(a)$
- 
- Every load gets its value from the last store before it to the same address

# Correct Implementation Under Relaxed Consistency

**TABLE 5.3:** Adding FENCEs for XC to Table 5.1's Program.

| Core C1                                                                     | Core C2                                                                                                | Comments                                                                               |
|-----------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| S1: data1 = NEW;<br>S2: data2 = NEW;<br><b>F1: FENCE</b><br>S3: flag = SET; | L1: r1 = flag;<br>B1: if (r1 ≠ SET) goto L1;<br><b>F2: FENCE</b><br>L2: r2 = data1;<br>L3: r3 = data2; | /* Initially, data1 & data2 = 0 & flag ≠ SET */<br>/* L1 & B1 may repeat many times */ |

# Correct Implementation Under Relaxed Consistency

| TABLE 5.4: Adding FENCEs for XC to Table 5.2's Critical Section Program.    |                                                                                                              |                                               |
|-----------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|-----------------------------------------------|
| Core C1                                                                     | Core C2                                                                                                      | Comments                                      |
| <b>F11: FENCE</b><br><br><b>A11: acquire(lock)</b>                          |                                                                                                              |                                               |
| <b>F12: FENCE</b><br><br>Some loads L1i interleaved<br>with some stores S1j |                                                                                                              | /* Arbitrary interleaving of L1i's & S1j's */ |
| <b>F13: FENCE</b><br><br>R11: release(lock)                                 | <b>F21: FENCE</b>                                                                                            | /* Handoff from critical section 1 */         |
| <b>F14: FENCE</b>                                                           | <b>A21: acquire(lock)</b><br><br><b>F22: FENCE</b><br><br>Some loads L2i interleaved<br>with some stores S2j | /* To critical section 2 */                   |
|                                                                             | <b>F23: FENCE</b><br><br>R22: release(lock)                                                                  | /* Arbitrary interleaving of L2i's & S2j's */ |
|                                                                             | <b>F24: FENCE</b>                                                                                            |                                               |

## Relaxed Consistency Memory Models

- Weak ordering
  - Distinguishes between data and synchronization operations
  - A synchronization operation is not issued until all previous operations are complete
  - No operations are issued until the previous synchronization operation completes
- Release consistency
  - Distinguishes between acquire and release synchronization operations
  - RCsc - maintains SC between synchronization operations
  - Acquire → all, all → release, and sync → sync

# Relaxed Consistency Memory Models

- Why should we use them?
  - Performance
- Why should we not use them?
  - Complexity

# Hardware Memory Models: One Slide Summary

| Relaxation      | $W \rightarrow R$<br>Order | $W \rightarrow W$<br>Order | $R \rightarrow RW$<br>Order | Read Others'<br>Write Early | Read Own<br>Write Early | Safety net                      |
|-----------------|----------------------------|----------------------------|-----------------------------|-----------------------------|-------------------------|---------------------------------|
| SC [16]         |                            |                            |                             |                             | ✓                       |                                 |
| IBM 370 [14]    | ✓                          |                            |                             |                             |                         | serialization instructions      |
| TSO [20]        | ✓                          |                            |                             |                             | ✓                       | RMW                             |
| PC [13, 12]     | ✓                          |                            |                             | ✓                           | ✓                       | RMW                             |
| PSO [20]        | ✓                          | ✓                          |                             |                             | ✓                       | RMW, STBAR                      |
| WO [5]          | ✓                          | ✓                          | ✓                           |                             | ✓                       | synchronization                 |
| RCsc [13, 12]   | ✓                          | ✓                          | ✓                           |                             | ✓                       | release, acquire, nsync,<br>RMW |
| RCpc [13, 12]   | ✓                          | ✓                          | ✓                           | ✓                           | ✓                       | release, acquire, nsync,<br>RMW |
| Alpha [19]      | ✓                          | ✓                          | ✓                           |                             | ✓                       | MB, WMB                         |
| RMO [21]        | ✓                          | ✓                          | ✓                           |                             | ✓                       | various MEMBAR's                |
| PowerPC [17, 4] | ✓                          | ✓                          | ✓                           | ✓                           | ✓                       | SYNC                            |

# DRFO Model

- Conceptually similar to WO
- Assumes no data races
- Allows many optimizations in the compiler and hardware

## Language Memory Models

- Developed much later
- Most are based on the data-race-free-0 (DRF0) model

Why do we need one?

- Isn't the hardware memory model enough?

## C++ Memory Model

- Adaptation of the DRFO memory model
  - SC for data race free programs
- C/C++ simply ignore data races
  - No safety guarantees in the language
- Memory operation
  - Synchronization: lock, unlock, atomic load, atomic store, atomic RMW
  - Data: Load, Store

## C++ Memory Model

- Compiler reordering **allowed** for memory operations M1 and M2 when:
  - M1 is a data operation and M2 is a read synchronization operation
  - M1 is write synchronization and M2 is data
  - M1 and M2 are both data with no synchronization between them
  - M1 is data and M2 is the write of a lock operation
  - M1 is unlock and M2 is either a read or write of a lock

# Write Correct C++ Code

- Mutually exclusive execution of critical code blocks

```
std::mutex mtx;  
{  
    mtx.lock();  
    // access shared data here  
    mtx.unlock();  
}
```

- Mutex provides inter-thread synchronization
  - Unlock() synchronizes with calls to lock() on the same mutex object

# Synchronize Using Locks

```
std::mutex mtx;
bool dataReady = false;

{

    mtx.lock();
    prepareData();
    dataReady = true;
    mtx.unlock();
}

{
    mtx.lock();
    if (dataReady) {
        consumeData();
    }
    mtx.unlock();
}
```

# Synchronize Using Locks

```
std::mutex mtx;
bool dataReady = false;

prepareData();
{
    mtx.lock();
    dataReady = true;
    mtx.unlock();
}

bool b;
{
    mtx.lock();
    b = dataReady;
    mtx.unlock();
}
if (b) {
    consumeData();
}
```

# Using Atomics

- “Data race free” variable by definition: `std::atomic<int>`
- A store synchronizes with operations that load the stored value
- Similar to `volatile` in Java
- C++ `volatile` is different!
  - Does not establish inter-thread synchronization, not atomic (can be part of a data race)

```
std::mutex mtx;
std::atomic<bool> dataReady(false);

prepareData();
dataReady.store(true);           if (dataReady.load()) {
                                consumeData();
                            }
```

# Memory Order of Atomics

- Specifies how regular, non-atomic memory accesses are to be ordered around an atomic operation
  - Default is sequential consistency

**atomic.h**

```
enum memory_order {  
    memory_order_relaxed,  
    memory_order_consume,  
    memory_order_acquire,  
    memory_order_release,  
    memory_order_acq_rel,  
    memory_order_seq_cst  
};
```

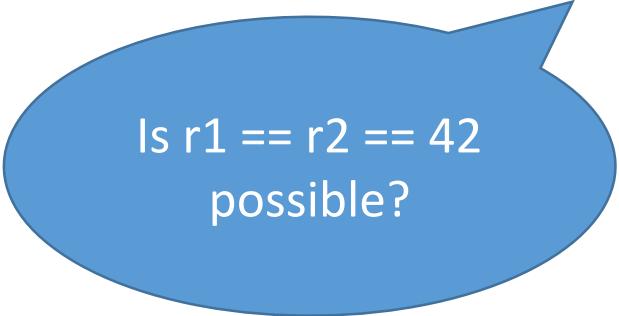
## Visibility and Ordering

- Visibility: When are the effects of one thread visible to another?
- Ordering: When can operations of any given thread appear out of order to another thread?

# Relaxed Ordering

```
// Thread 1:  
r1 = y.load(memory_order_relaxed);  
x.store(r1, memory_order_relaxed);
```

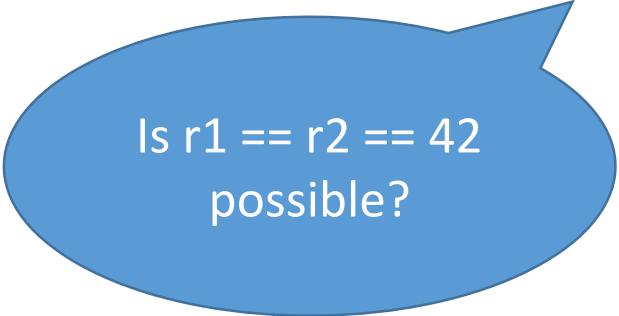
```
// Thread 2:  
r2 = x.load(memory_order_relaxed); // C  
y.store(42, memory_order_relaxed); // D
```



Is  $r1 == r2 == 42$   
possible?

# Relaxed Ordering

```
// Thread 1:  
r1 = x.load(memory_order_relaxed);  
If (r1 == 42) {  
    y.store(r1, memory_order_relaxed);  
}
```



Is  $r1 == r2 == 42$   
possible?

```
// Thread 2:  
r2 = y.load(memory_order_relaxed);  
If (r2 == 42) {  
    x.store(42, memory_order_relaxed);  
}
```

## Ensuring Visibility

- Writer thread releases a lock
  - Flushes all writes from the thread's working memory
- Reader thread acquires a lock
  - Forces a (re)load of the values of the affected variables
- Atomic (C++)/ volatile (Java)
  - Values written are made visible immediately before any further memory operations
  - Readers reload the value upon each access
- Thread join
  - Parent thread is guaranteed to see the effects made by the child thread

# Java Memory Model (JMM)

- First high-level language to incorporate a memory model
- Provides memory- and type-safety, so has to define some semantics for data races

Initially  $x = y = 0$

Thread 1:

$y = 1;$   
 $r1 = x;$

Thread 2:

$x = 1;$   
 $r2 = y;$

assert  $r1 \neq 0 \parallel r2 \neq 0$

## JMM (...contd)

Initially  $x = y = 0$

Thread 1:

```
r1 = x;  
y = 1;
```

Thread 2:

```
r2 = y;  
x = 1;
```

```
assert r1 == 0 || r2 == 0
```

# Happens-before Memory Model (HBMM)

Initially  $x = 0$

Thread 1:

$x = 7;$

Thread 2:

**if** ( $x \neq 0$ )  
 $r2 = r1 / x;$

## HBMM (...contd)

Initially  $x = y = 0$

Thread 1:

```
r1 = x;  
if (r1 == 1)  
    y = 1;
```

Thread 2:

```
r2 = y;  
if (r2 == 1)  
    x = 1;
```

assert r1 == 0 && r2 == 0

# JMM is Stronger than DRFO and HBMM

Initially  $x = y = 0$

Thread 1:

```
r1 = x;  
y = r1;
```

```
assert r1 != 42
```

Thread 2:

```
r2 = y;  
x = r2;
```

# JVMs Do Not Comply with the JMM

Initially  $x = y = 0$

Thread 1:

```
1 r1 = x;  
2 y = r1;
```

Thread 2:

```
3 r2 = y;  
4 if (r2 == 1) {  
5   r3 = y;  
6   x = r3;  
7 } else x = 1;
```

assert  $r2 == 0$

## What Constitutes a Good Memory Model?

- Programmability
- Performance
- Portability
- Precision

## Lessons Learnt

- SC for DRF is the minimal baseline
  - Make sure the program is free of data races
  - System guarantees SC execution
- Specifying semantics for racy programs is hard
- Simple optimizations may introduce unintended consequences

# Memory Consistency Model

---

Swarnendu Biswas  
UT Austin

# **CS377P Programming for Performance**

## **Introduction to Accelerators**

---

Sreepathi Pai

April 20, 2017

ICES

# Outline

---

Introduction to Accelerators

GPU Architectures

GPU Programming Models

# Outline

---

Introduction to Accelerators

GPU Architectures

GPU Programming Models

# Accelerators

- Single-core processors
- Multi-core processors
- What if these aren't enough?
- Accelerators, specifically GPUs
  - what they are
  - when you should use them

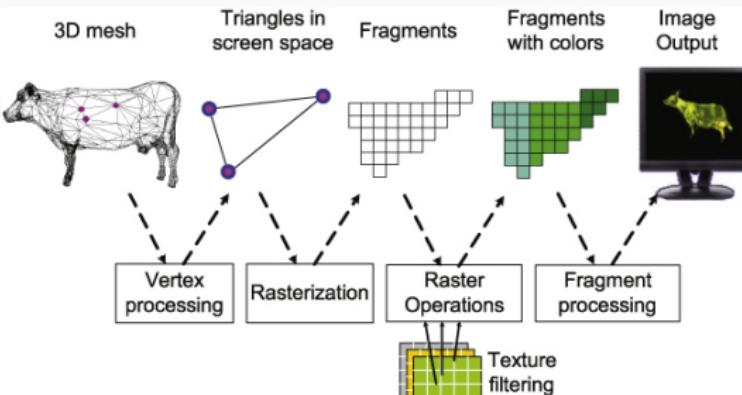
# Timeline

---

- 1980s
  - Geometry Engines
- 1990s
  - Consumer GPUs
  - Out-of-order Superscalars
- 2000s
  - General-purpose GPUs
  - Multicore CPUs
  - Cell BE (Playstation 3)
  - Lots of specialized accelerators in phones

# The Graphics Processing Unit (1980s)

- SGI Geometry Engine
- Implemented the *Geometry Pipeline*
  - Hardwired logic
- Embarrassingly Parallel
  - $O(Pixels)$
  - Large number of logic elements
  - High memory bandwidth
- From Kaufman et al. (2009):



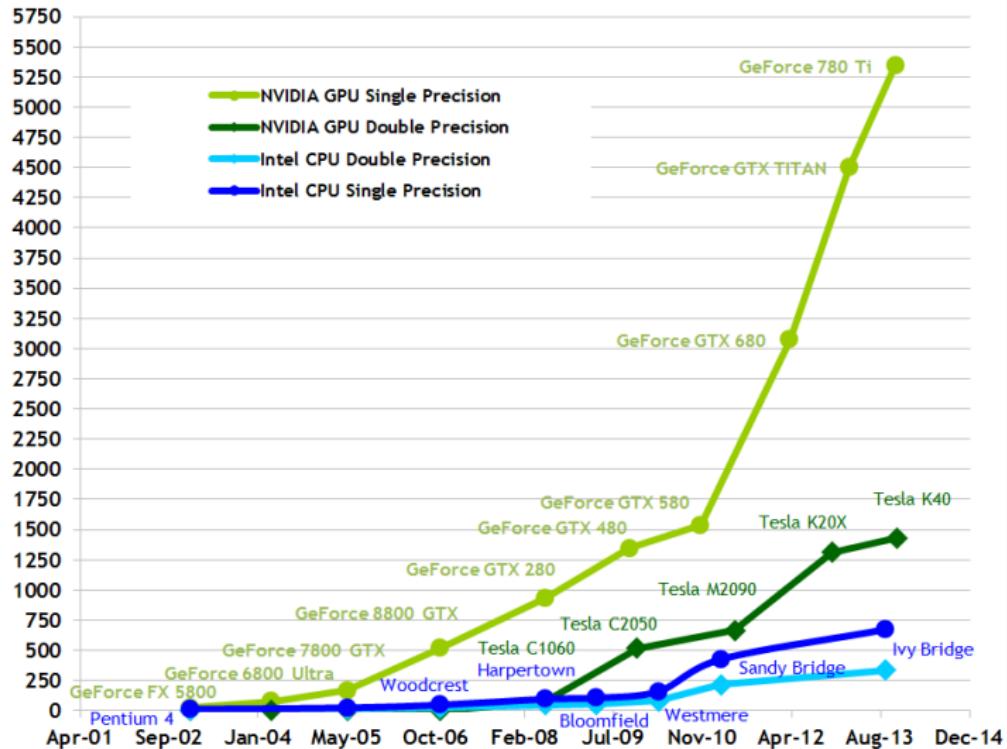
## GPU 2.0 (circa 2004)

- Like CPUs, GPUs benefited from Moore's Law
- Evolved from fixed-function hardwired logic to flexible, programmable ALUs
- Around 2004, GPUs were programmable “enough” to do some non-graphics computations
  - Severely limited by graphics programming model (shader programming)
- In 2006, GPUs became “fully” programmable
  - GPGPU: General-Purpose GPU
  - NVIDIA releases “CUDA” language to write non-graphics programs that will run on GPUs

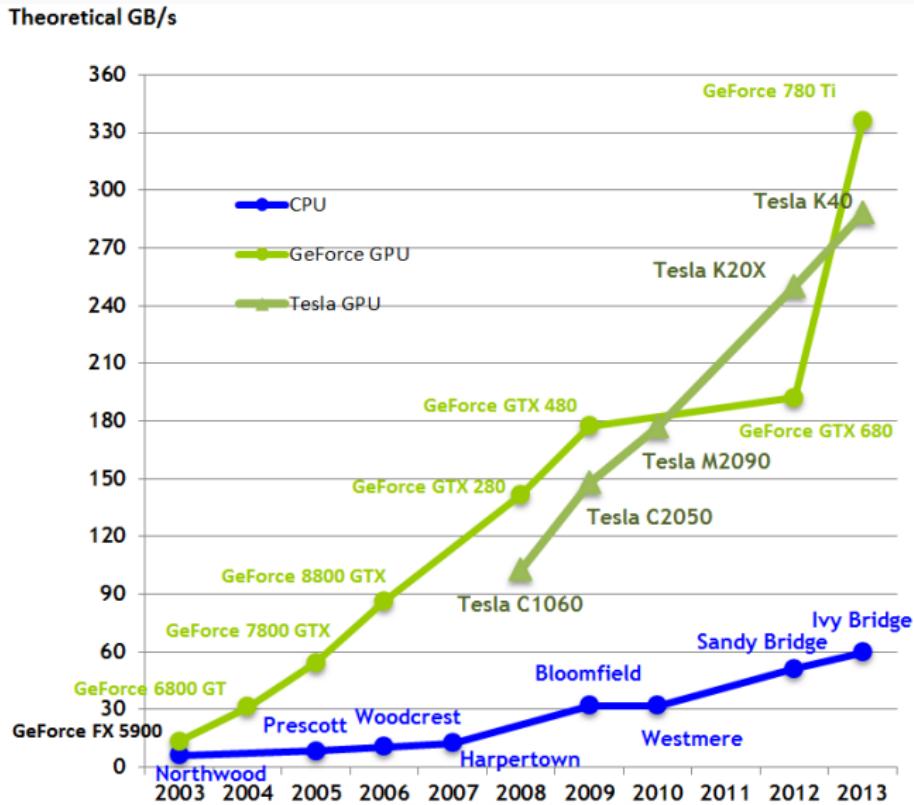


# FLOPS/s

Theoretical GFLOP/s



# Memory Bandwidth



# GPGPU Today

- GPUs are widely deployed as accelerators
- Intel Paper
  - 10x vs 100x Myth
- GPUs so successful that other accelerators are dead
  - Sony/IBM Cell BE
  - Clearspeed RSX
- Kepler K40 GPUs from NVIDIA have performance of 4TFlops (peak)
  - CM-5, #1 system in 1993 was 60 Gflops (Linpack)
  - ASCI White (#1 2001) was 4.9 Tflops (Linpack)



# Accelerator Programming Models

- CPUs have always depended on co-processors
  - I/O co-processors to handle slow I/O
  - Math co-processors to speed up computation
  - H.264 co-processor to play video (Phones)
  - DSPs to handle audio (Phones)
- Many have been transparent
  - Drop in the co-processor and everything sped up
- Or used a function-based model
  - Call a function and it is sped up (e.g. “decode video”)
- The GPU is not a transparent accelerator for general purpose computations
  - Only graphics code is sped up transparently
- Code must be rewritten to target GPUs

# Using a GPU

- You must retarget code for the GPU
  - Rewrite, recompile, translate, etc.

# Outline

---

Introduction to Accelerators

GPU Architectures

GPU Programming Models

# The Two (Three?) Kinds of GPUs

- Type 1: Discrete GPUs
  - More computational power
  - More memory bandwidth
  - Separate memory

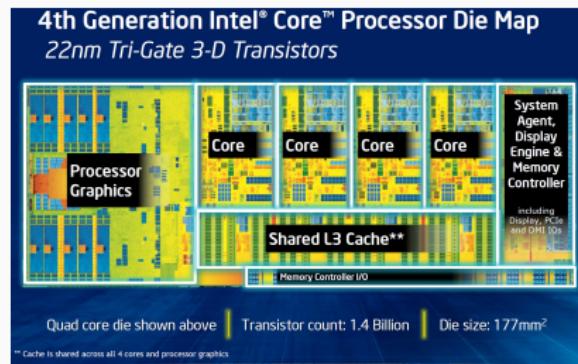
NVIDIA



# The Two (Three?) Kinds of GPUs #2

- Type 2: Integrated GPUs
  - Share memory with processor
  - Share bandwidth with processor
  - Consume Less power
  - Can participate in cache coherence

Intel



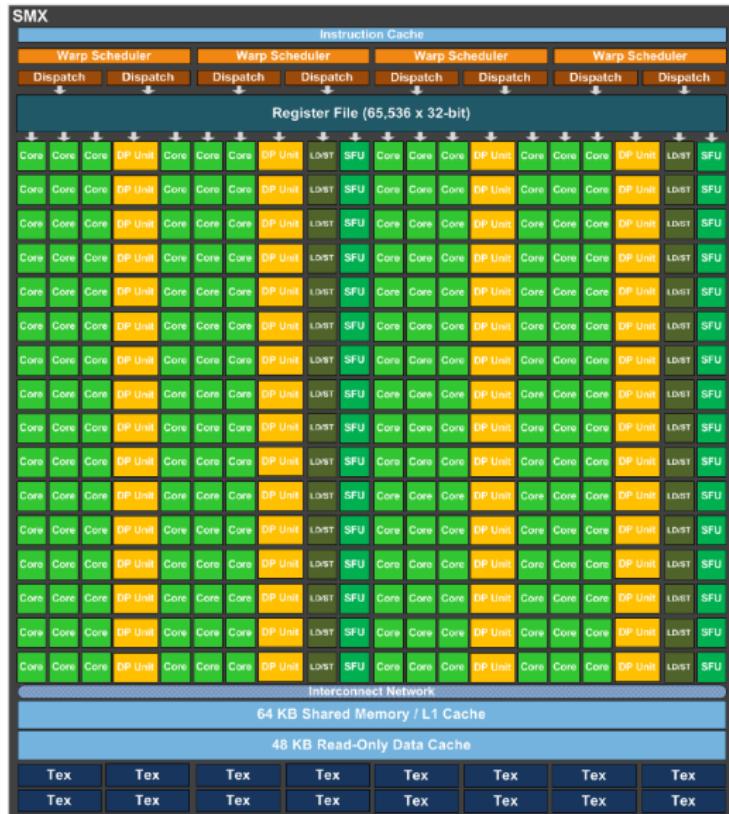
# The NVIDIA Kepler



# Using a Discrete GPU

- You must retarget code for the GPU
  - Rewrite, recompile, translate, etc.
- Working set must fit in GPU RAM
- You must copy data to/from GPU RAM
  - “You”: Programmer, Compiler, Runtime, OS, etc.
  - Some recent hardware can do this for you (it’s slow)

# NVIDIA Kepler SMX (i.e. CPU core equivalent)



# NVIDIA Kepler SMX Details

- 2-wide Inorder
- 4-wide SMT
  - 2048 threads per core (64 warps)
  - 15 cores
  - Each thread runs the same code (hence SIMT)
- 65536 32-bit registers (256KBytes)
  - A thread can use upto 255 of these
  - *Partitioned* among threads (not shared!)
- 192 ALUs
- 64 Double-precision
- 32 Load/store
- 32 Special Functional Unit
- 64 KB L1/Shared Cache
  - Shared cache is software-managed cache

# CPU vs GPU

| Parameter              | CPU                                            | GPU                                          |
|------------------------|------------------------------------------------|----------------------------------------------|
| Clockspeed             | > 1 GHz                                        | 700 MHz                                      |
| RAM                    | GB to TB                                       | 12 GB (max)                                  |
| Memory B/W             | 60 GB/s                                        | > 300 GB/s                                   |
| Peak FP                | < 1 TFlop                                      | > 1 TFlop                                    |
| Concurrent Threads     | O(10)                                          | O(1000)<br>[O(10000)]                        |
| LLC cache size         | > 100MB (L3)<br>[eDRAM] O(10)<br>[traditional] | < 2MB (L2)                                   |
| Cache size per thread  | O(1 MB)                                        | O(10 bytes)                                  |
| Software-managed cache | None                                           | 48KB/SMX                                     |
| Type                   | OOO<br>scalar                                  | super-scalar<br>2-way Inorder<br>superscalar |

# Using a GPU

- You must retarget code for the GPU
  - Rewrite, recompile, translate, etc.
- Working set must fit in GPU RAM
- You must copy data to/from GPU RAM
  - “You”: Programmer, Compiler, Runtime, OS, etc.
  - Some recent hardware can do this for you
- Data accesses should be streaming
  - Or use scratchpad as user-managed cache
- Lots of parallelism preferred (throughput, not latency)
- SIMD-style parallelism best suited
- High arithmetic intensity (FLOPs/byte) preferred

# Showcase GPU Applications

- Image Processing
- Graphics Rendering
- Matrix Multiply
- FFT

See "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU" by V.W.Lee et al. for more examples and a comparison of CPU and GPU.

# Outline

---

Introduction to Accelerators

GPU Architectures

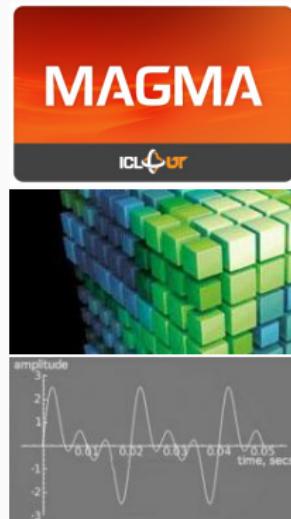
GPU Programming Models

# Hierarchy of GPU Programming Models

| Model                | GPU                        | CPU Equivalent        |
|----------------------|----------------------------|-----------------------|
| Vectorizing Compiler | PGI CUDA Fortran           | gcc, icc, etc.        |
| “Drop-in” Libraries  | cuBLAS                     | ATLAS                 |
| Directive-driven     | OpenACC,<br>OpenMP-to-CUDA | OpenMP                |
| High-level languages | pyCUDA                     | python                |
| Mid-level languages  | OpenCL, CUDA               | pthreads +<br>C/C++   |
| Low-level languages  | PTX, Shader                | -                     |
| Bare-metal           | SASS                       | Assembly/Machine code |

## “Drop-in” Libraries

- “Drop-in” replacements for popular CPU libraries,  
examples from NVIDIA:
  - CUBLAS/NVBLAS for BLAS (e.g. ATLAS)
  - CUFFT for FFTW
  - MAGMA for LAPACK and BLAS
- These libraries may still expect you to manage data transfers manually
- Libraries may support multiple accelerators (GPU + CPU + Xeon Phi)



# GPU Libraries

- NVIDIA Thrust
  - Like C++ STL, but executes on the GPU
- Modern GPU
  - At first glance:  
high-performance library routines for sorting, searching, reductions, etc.
  - A deeper look: Specific “hard” problems tackled in a different style
- NVIDIA CUB
  - Low-level primitives for use in CUDA kernels



# Directive-Driven Programming

- OpenACC, new standard for “offloading” parallel work to an accelerator
  - Currently supported only by PGI Accelerator compiler
  - gcc 5.0 support is ongoing
- OpenMPC, a research compiler, can compile OpenMP code + extra directives to CUDA
  - OpenMP 4.0 also supports offload to accelerators
  - Not for GPUs yet

```
int main(void) {
    double pi = 0.0f; long i;

    #pragma acc parallel loop reduction(+:pi)
    for (i=0; i<N; i++) {
        double t= (double)((i+0.5)/N);
        pi +=4.0/(1.0+t*t);
    }

    printf("pi=%16.15f\n",pi/N);
    return 0;
}
```

# Python-based Tools (pyCUDA)

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy
from pycuda.compiler import SourceModule

mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)

multiply_them(
    drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1), grid=(1,1))

print dest-a*b
```

# OpenCL

- C99-based dialect for programming heterogenous systems
  - Originally based on CUDA
  - nomenclature is different
- Supported by more than GPUs
  - Xeon Phi, FPGAs, CPUs, etc.
- Source code is portable (somewhat)
  - Performance may not be!
- Poorly supported by NVIDIA

# CUDA

- “Compute Unified Device Architecture”
- First language to allow general-purpose programming for GPUs
  - preceded by shader languages
- Promoted by NVIDIA for their GPUs
- Not supported by any other accelerator
  - though commercial CUDA-to-x86/64 compilers exist
- We will focus on CUDA programs

# CUDA Architecture

- From 10000 feet – CUDA is like pthreads
  - CUDA language – C++ dialect
- Host code (CPU) and GPU code in same file
- Special language extensions for GPU code
- CUDA Runtime API
  - Manages runtime GPU environment
  - Allocation of memory, data transfers, synchronization with GPU, etc.
  - Usually invoked by host code
- CUDA Device API
  - Lower-level API that CUDA Runtime API is built upon

# CUDA Limitations

- No standard library for GPU functions
- No parallel data structures
- No synchronization primitives (mutex, semaphores, queues, etc.)
  - you can roll your own
  - only atomic\*() functions provided
- Toolchain not as mature as CPU toolchain
  - Felt intensely in performance debugging
- It's only been a decade :)

# Conclusions

- GPUs are very interesting parallel machines
- They're not going away
  - Xeon Phi might pose a formidable challenge
- They're here and now
  - Your laptop probably already contains one
  - Your phone definitely has one

# **CS377P Programming for Performance**

## **Basic GPU Programming**

---

Sreepathi Pai

April 24, 2015

UTCS

# Outline

---

Introduction to CUDA

Basic Performance

Memory Performance

# Outline

---

Introduction to CUDA

Basic Performance

Memory Performance

## Background

- Discrete GPUs
- CUDA

# Basics

- CUDA is a C++ dialect
  - extra keywords
  - extra semantics
- CUDA code consists of:
  - *device* code (executes on the GPU)
  - *host* code (executes on the CPU)
  - execution always starts on the CPU
- CUDA compiler is nvcc

# First CUDA program: Vector addition

```
void vector_add(int *a, int *b, int *c, int N) {  
    for(int i = 0; i < N; i++) {  
        c[i] = a[i] + b[i];  
    }  
}  
  
int main(void) {  
    ...  
    vector_add(a, b, c, N);  
}
```

## Kernels: `--global__` keyword

```
--global__
void vector_add(int *a, int *b, int *c, int N) {
    ...
}

int main(void) {
    ...
    vector_add<<<...>>>(a, b, c, N);
}
```

- The `--global__` keyword indicates a GPU kernel
- GPU kernels must be called with a *configuration*

# Kernels: Configuration

- GPU kernels are SPMD kernels
  - All threads execute the same code
- Number of threads to execute is specified at launch time
  - As a *grid* of  $B$  *thread blocks* of  $T$  threads each
  - Total threads:  $B \times T$
- Reason: Only threads within the same thread block can communicate with each other (cheaply)
  - Other reasons too, but this is the only algorithm-specific reason

## Determining the Configuration: Work Size

- Determine a thread block size: say, 256 threads
- Divide work by thread block size
  - Round up
  - $\lceil N/256 \rceil$
- Configuration can be changed every call

```
int threads = 256;
int Nup = (N + threads - 1) / threads;
int blocks = Nup / threads;

vector_add<<<blocks, threads>>>(...)
```

## Distributing work in the kernel

```
--global--  
vector_add(int *a, int *b, int *c, int N) {  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
  
    if(tid < N) {  
        c[tid] = a[tid] + b[tid];  
    }  
}
```

- Maximum  $2^{32}$  threads supported
- $gridDim$ ,  $blockDim$ ,  $blockIdx$  and  $threadIdx$  are CUDA-provided variables

## CUDA vector\_add so far

```
--global--
vector_add(int *a, int *b, int *c, int N) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if(tid < N) {
        c[tid] = a[tid] + b[tid];
    }
}

int main(void) {
    int threads = 256;
    int Nup = (N + threads - 1) / threads;
    int blocks = Nup / threads;

    ...

    vector_add<<<blocks, threads>>>(a, b, c, N);
}
```

## CUDA vector\_add: GPU memory

- GPU can't read CPU memory directly by default
- Arrays `a`, `b` need to be copied to GPU memory
- The result `c` needs to be copied back to CPU memory
- Two CUDA functions:
  - `cudaMalloc` allocates GPU memory
  - `cudaMemcpy` copies memory between CPU and GPU

## CUDA vector\_add with GPU memory

```
int *g_a, *g_b, *g_c;

cudaMalloc(&g_a, sizeof(a[0]) * N);
cudaMalloc(&g_b, sizeof(b[0]) * N);
cudaMalloc(&g_c, sizeof(c[0]) * N);

cudaMemcpy(g_a, a, sizeof(a[0]) * N, cudaMemcpyHostToDevice);
cudaMemcpy(g_b, b, sizeof(b[0]) * N, cudaMemcpyHostToDevice);

vector_add<<<...>>>(g_a, g_b, g_c, N);

cudaMemcpy(c, g_c, sizeof(c[0]) * N, cudaMemcpyDeviceToHost);
```

## CUDA vector\_add: complete?

```
--global--
vector_add(int *a, int *b, int *c, int N) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if(tid < N) {
        c[tid] = a[tid] + b[tid];
    }
}

int main(void) {
    int threads = 256;
    int Nup = (N + threads - 1) / threads;
    int blocks = Nup / threads;

    cudaMalloc(&g_a, sizeof(a[0]) * N);
    cudaMalloc(&g_b, sizeof(b[0]) * N);
    cudaMalloc(&g_c, sizeof(c[0]) * N);

    cudaMemcpy(g_a, a, sizeof(a[0]) * N, cudaMemcpyHostToDevice);
    cudaMemcpy(g_b, b, sizeof(b[0]) * N, cudaMemcpyHostToDevice);

    vector_add<<<blocks, threads>>>(g_a, g_b, g_c, N);

    cudaMemcpy(c, g_c, sizeof(c[0]) * N, cudaMemcpyDeviceToHost);
}
```

# Outline

---

Introduction to CUDA

Basic Performance

Memory Performance

# Heterogeneous Systems

- GPU + CPU form a *heterogeneous* system
  - “A system with non-trivial choices of where to perform a computation”
- Parallel execution is possible
  - CPU and GPU can work independently in parallel
  - In fact, GPU allows data transfers in parallel to GPU execution
- Consider distributing work so that all execution units (CPU and GPU) are fully utilized
- Not easy to do manually, but no automatic solution widely accepted yet

# Measurement Pitfalls

Keep in mind:

- A GPU program is a parallel CPU program
  - i.e. GPU code sometimes runs on a separate thread
- A CPU + GPU system is a distributed system
  - i.e. clocks are unsynchronized
  - especially across GPU cores
- Use timelines not intervals to reason about performance
  - timelines capture overlap
  - timelines illustrate critical path
  - NVIDIA Profiler provides timelines

# How NOT to time a GPU kernel

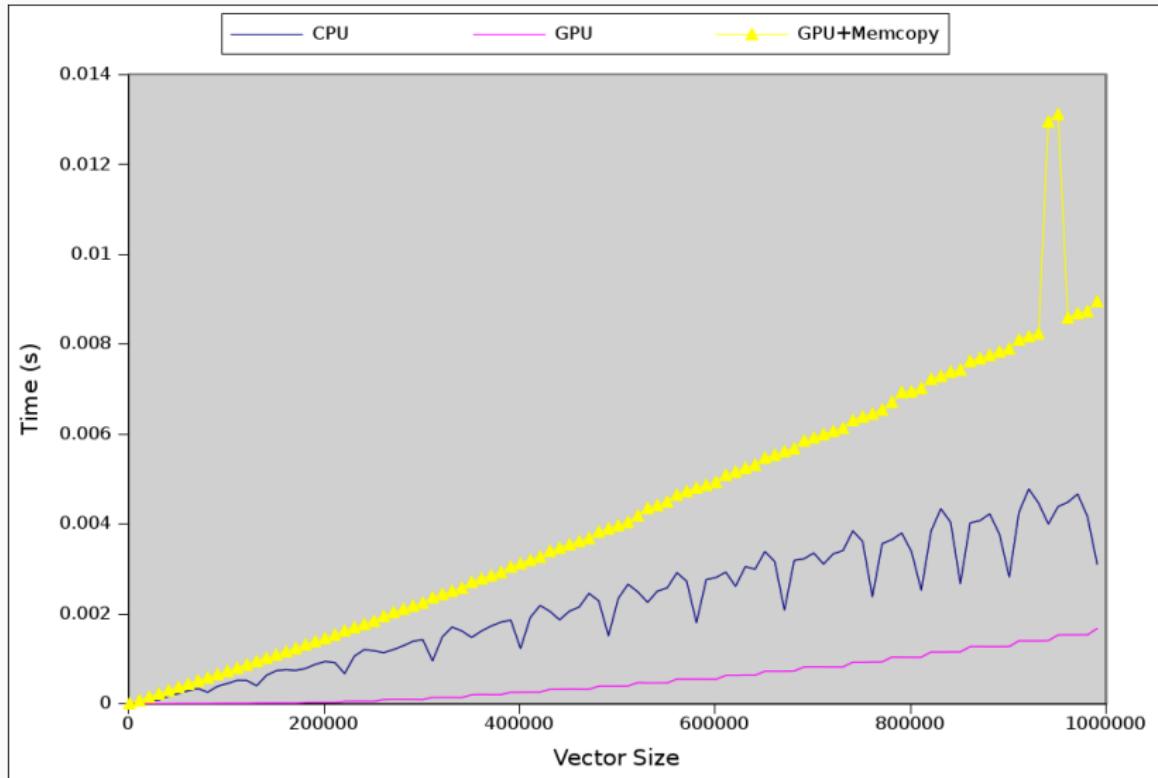
```
struct stopwatch va;  
  
clock_start (&va ) ;  
vector_add_1 <<<14*8, 384>>>(ga , gb , gc , N) ;  
clock_stop (&va) ;  
  
printf (TIMEFMT "s \n" , va.elapsed.tv_sec , va.elapsed.tv_nsec ) ;
```

- Output is approx.  $40\mu s$  on my machine
- NVIDIA Compute Profiler:
  - gputime=[ 14078.336 ] ( $\mu s$ )

# Vector Addition Performance



# Vector Addition Performance



## How many threads: GPU Occupancy

- CPU threads share resources by time multiplexing
  - One thread owns all CPU resources (registers, etc.) for its time slice
  - Context-switches are performed by OS
- GPU threads *do not share* resources
  - Own fixed partition of resources for entire lifetime of thread
  - Context-switches are performed by hardware every few cycles
- Changing number of threads changes *utilization* of resources

## GPU Resources per SM (NVIDIA Kepler)

| Resource      | Available | Maximum    |
|---------------|-----------|------------|
| Threads       | 2048      | 1024/block |
| Shared Memory | 48K (max) | 48K/block  |
| Registers     | 65536     | 255/thread |
| Thread Blocks | 16        | 16/SM      |

- Every block consumes:
  - $T$  threads
  - $T \times R$  registers where  $R$  is registers per thread
  - 1 block
  - $SM$  shared memory per block (optional)
- The resource that gets exhausted first determines occupancy and residency
  - *Occupancy*: number of hardware threads utilized
  - *Residency*: number of hardware blocks utilized

## GPU Occupancy: Example 1

```
kernel<<<2048, 32>>>()
```

- $T = 32$ 
  - thread limit  $2048/32 = 64$  thread blocks
- $R = 100$  ( $100 \times 32 = 3200$  per thread block)
  - register limit  $65536/3200 = 20$  thread blocks
- $SM = 1K$ 
  - SM limit  $48K/1K = 48$  thread blocks
- Limiting resource: thread blocks (16)
- Residency: 16
- Occupancy:  $(16 \times 32)/2048 = 25\%$

## GPU Occupancy: Example 2

```
kernel<<<2048, 64>>>()
```

- $T = 64$ 
  - thread limit  $2048/64 = 32$  thread blocks
- $R = 100$  ( $100 \times 64 = 6400$  per thread block)
  - register limit  $65536/6400 = ?$  thread blocks
- $SM = 1K$ 
  - SM limit  $48K/1K = 48$  thread blocks
- Limiting resource: ?
- Residency: ?
- Occupancy:  $(? \times 64)/2048 = ?\%$

## How many threads?

- Try to maximize utilization (NVIDIA Manual)
- Is there a better strategy?
  - See Volkov, V., "Better Performance at Lower Occupancy" , GTC 2010

# Outline

---

Introduction to CUDA

Basic Performance

Memory Performance

# Data Layout for GPU programs (AoS)

```
struct pt {  
    int x;  
    int y;  
};  
  
__global__  
void aos_kernel(int n_pts, struct pt *p) {  
    int tid = blockIdx.x * blockDim.x + threadIdx.x;  
    int nthreads = blockDim.x * gridDim.x;  
  
    for(int i = tid; i < n_pts; i += nthreads) {  
        p[i].x = i;  
        p[i].y = i * 10;  
    }  
}
```

In main():

```
struct pt *p;  
cudaMalloc(&p, ...)
```

# Data Layout for GPU programs (SoA)

```
struct pt {  
    int *x;  
    int *y;  
};  
  
__global__  
void soa_kernel(int n_pts, struct pt p) {  
    int tid = blockIdx.x * blockDim.x + threadIdx.x;  
    int nthreads = blockDim.x * gridDim.x;  
  
    for(int i = tid; i < n_pts; i += nthreads) {  
        p.x[i] = i;  
        p.y[i] = i * 10;  
    }  
}
```

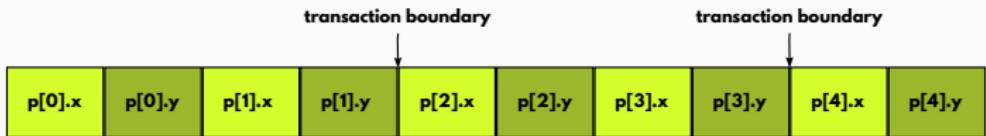
In main():

```
struct pt p;  
cudaMalloc(&p.x, ...)  
cudaMalloc(&p.y, ...)
```

# AoS vs SoA for GPU programs

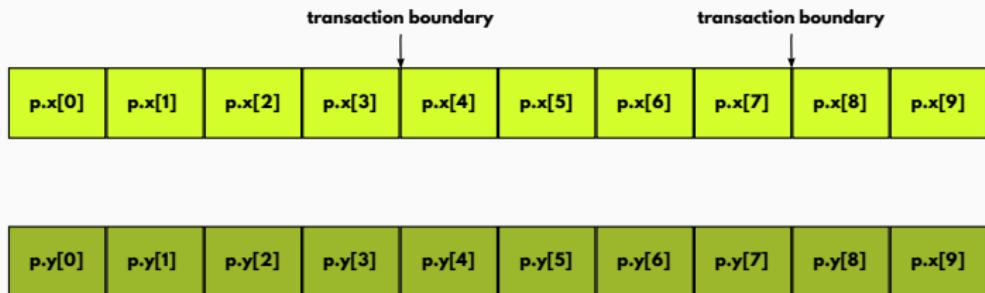
- Array of Structures
- Structure of Arrays
- Which is better for CPU?
- Which is better for GPU?

# AoS memory layout



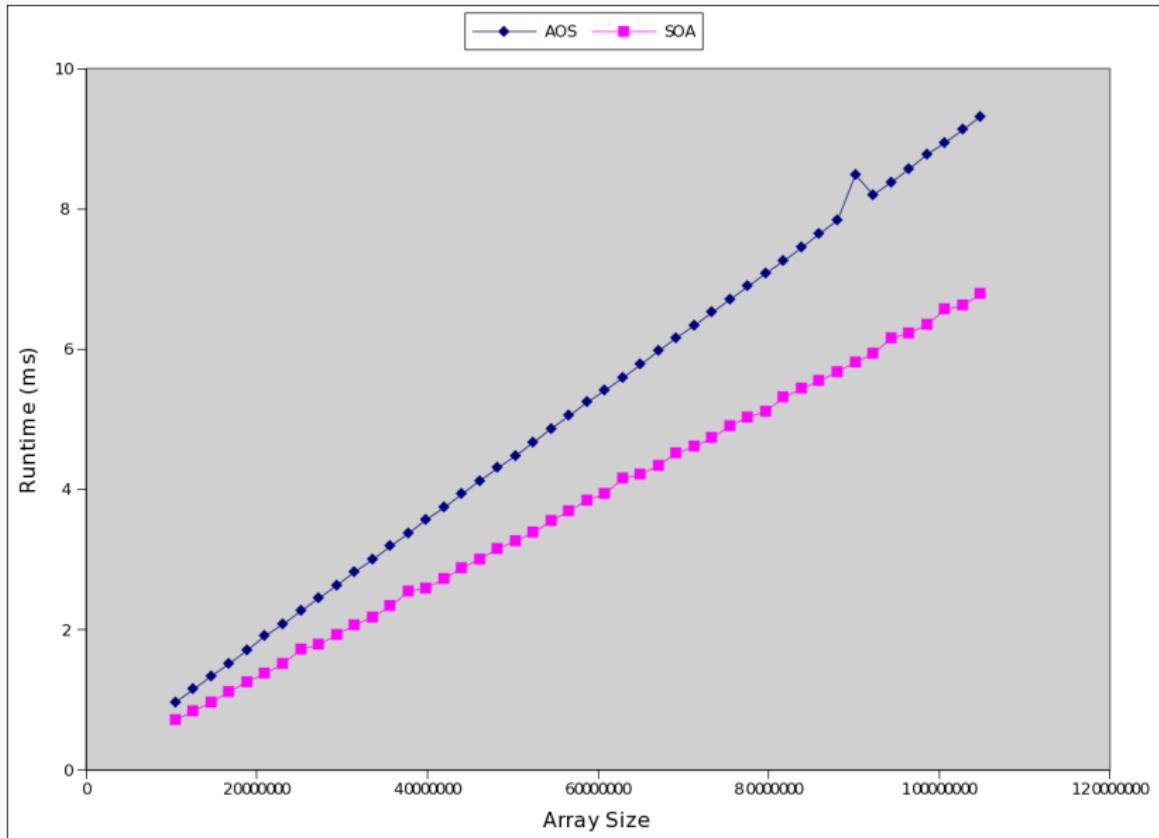
- $p[i].x$  memory bandwidth utilization?

# SoA memory layout

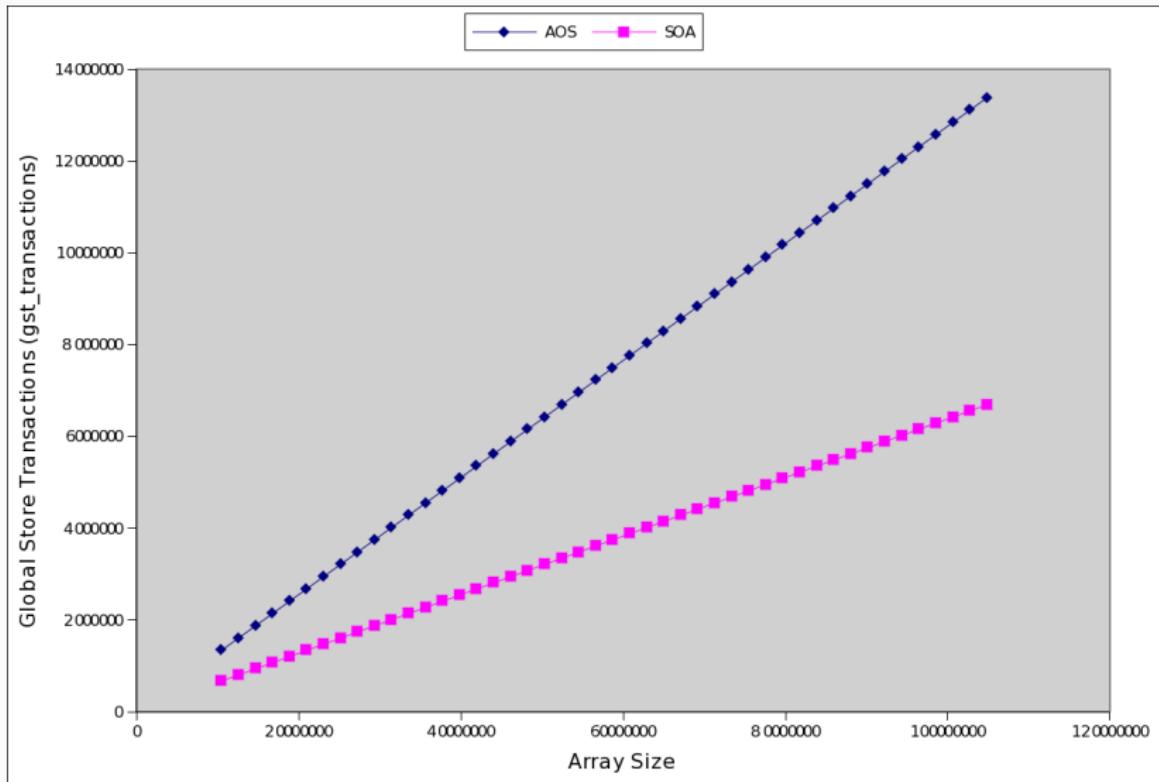


- `p.x[i]` memory bandwidth utilization?

# AoS vs SoA Performance



# AoS vs SoA: Number of Memory Transactions



# Assigning Work to Threads

Blocked:

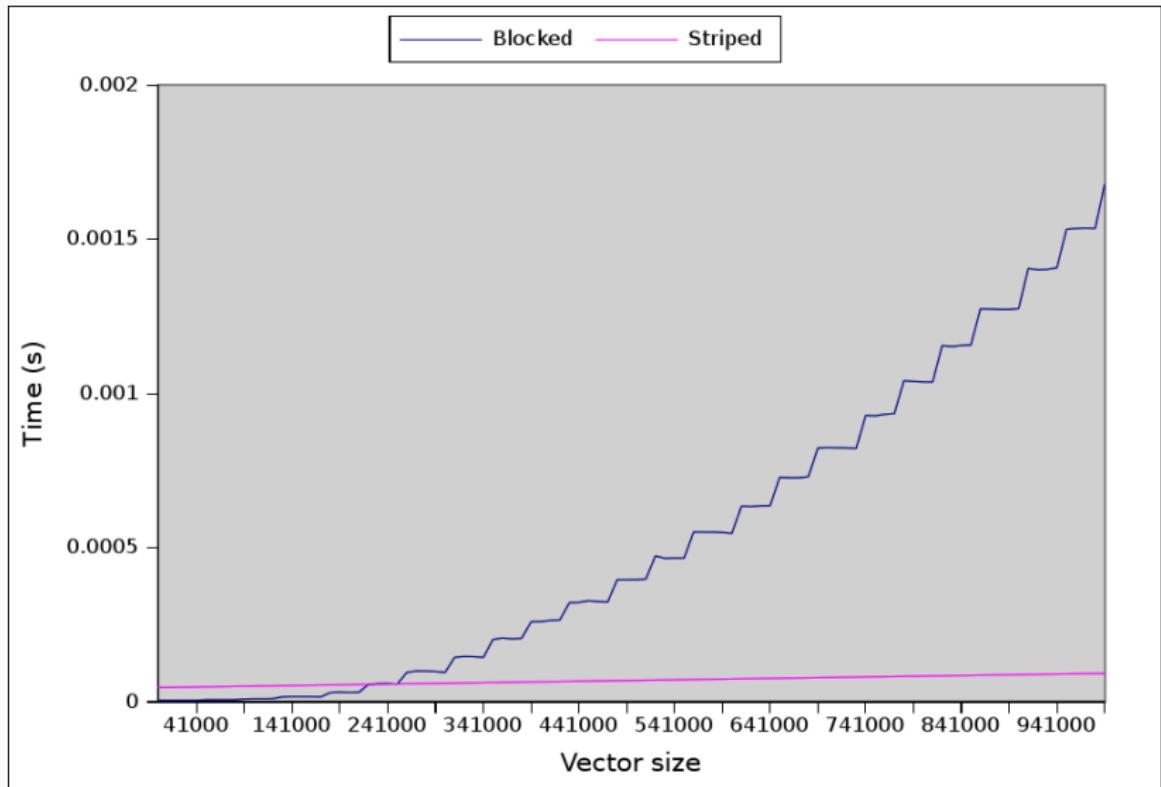
```
start = tid * blksize;  
end = start + blksize;  
  
for(i = start; i < N && i < end; i++)  
    a[i] = b[i] + c[i]
```

Interleaved:

```
start = tid;  
  
for(i = start; i < N; i+=nthreads)  
    a[i] = b[i] + c[i]
```

Which, if any, is faster?

# Blocking vs Interleaved



## Exploiting Spatial Locality: Texture Caches

- Textures are 2-D images that are “wrapped” around 3-D models
- Exhibit 2-D locality, so textures have a separate cache
- GPU contains a texture fetch unit that non-graphics programs can also use
  - Step 1: map arrays to textures
  - Step 2: replace array reads by `tex1Dfetch()`, `tex2Dfetch()`
- Catch: Only read-only data can be cached
  - you can write to the array, but it may not become visible through the texture in the same kernel call
  - i.e. texture caches are not coherent with GPU memory
- Easiest way to use textures:
  - `const __restrict__ *`
  - Compiler will automatically use texture cache for marked arrays

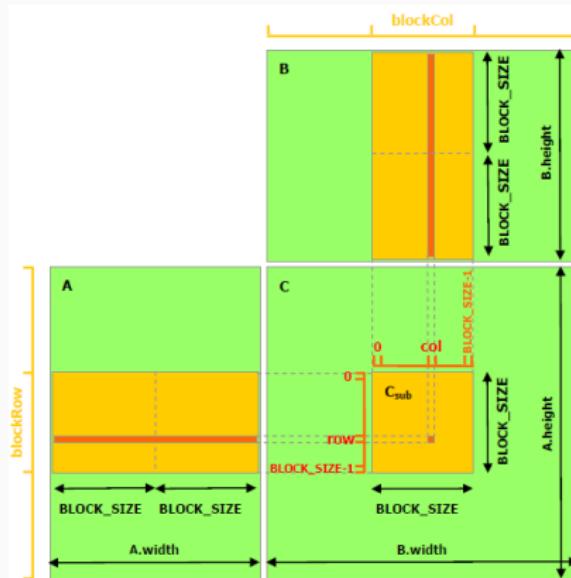
# Exploiting Locality: Shared Memory

- “Shared Memory” is on-chip software-managed cache, also known as a scratchpad
- 48K maximum size
- Partitioned among thread blocks
- `__shared__` qualifier places variables in shared memory
- Can be used for communicating between threads of the same thread block

```
__shared__ int x;  
  
if(threadIdx.x == 0)  
    x = 1;  
  
__syncthreads(); //required!  
  
printf("%d\n", x);
```

# Shared Memory (SGEMM)

```
--shared__ float c_sub[BLOCKSIZE] [BLOCKSIZE];  
  
// calculate c_sub  
  
__syncthreads();  
  
// write out c_sub to memory
```



# Constant Data Cache

- 64KB of “constant” data
  - not written by kernel
- Suitable for read-only, “broadcast” data
- All threads in a warp read the same constant data item at the same time
  - what type of locality is this?
- Uses: Filter coefficients
  - 2dconv: convolution matrix entries

# Summary of Memory Performance

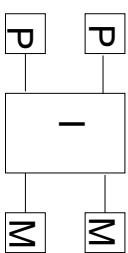
- Layout data structures in memory to maximize bandwidth utilization
- Assign work to threads to maximize bandwidth utilization
- Rethink caching strategies
  - identify readonly data
  - identify blocks that you can load into shared memory
  - identify tables of constants

## Lecture 2

# *Logical Abstractions of Multiprocessors*

## Physical Organization

- Uniform memory access (UMA) machines

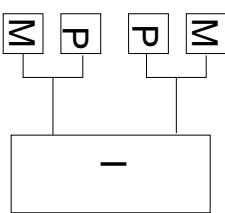


All memory is equally far away from all processors.

Early parallel processors like NYU Ultracomputer

Problem: why go across network for instructions? read-only data?  
what about caches?

- Non-uniform memory access (NUMA) machines:



Access to local memory is usually 10-1000 times  
faster than access to non-local memory

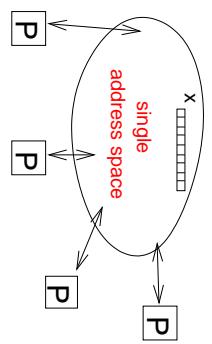
Static and dynamic locality of reference are critical for high performance.

Compiler support? Architectural support?

Bus-based symmetric multiprocessors (SMP's): combine both aspects

# Logical Organization

## - Shared Memory Model



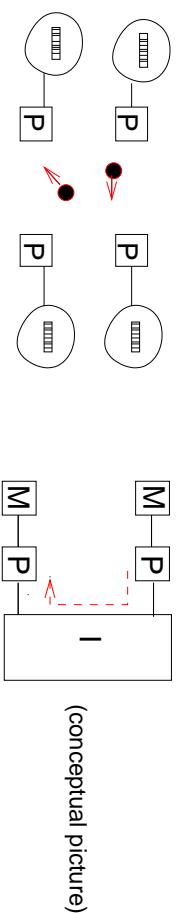
- hardware/systems software provide single address space model to applications programmer

- some systems: distinguish between local and remote references

- communication between processors: read/write shared memory locations:

**put get**

## - Distributed Memory Model (Message Passing)



- each processor has its own address space

- communication between processors: messages (like e-mail)

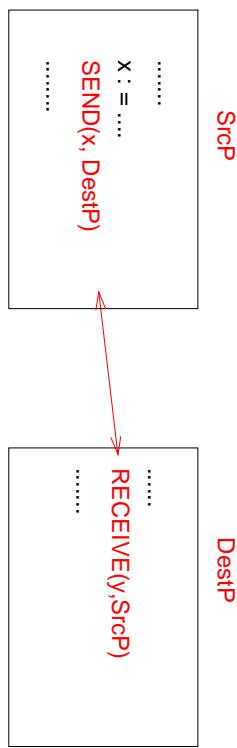
- basic message-passing commands: **send receive**

**Key difference: In SMM, P1 can access remote memory locations w/o prearranged participation of application program on remote processor**

# Message Passing

## Blocking SEND/RECEIVE : couple data transfer and synchronization

- Sender and receiver rendezvous to exchange data

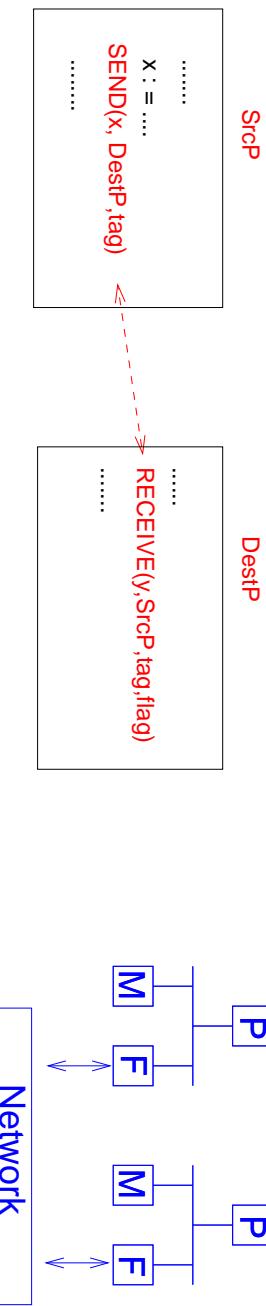


History: Caltech Cosmic Cube

- SrcP field in RECEIVE command permits DestP to select which processor it wants to receive data from
- Implementation:
  - SrcP sends token saying 'ready to send'
  - DestP returns token saying 'me too'
  - Data transfer takes place directly between application programs w/o buffering in O/S
- Motivation: Hardware 'channels' between processors in early multicomputers
- Problem:
  - sender cannot push data out and move on
  - receiver cannot do other work if data is not available yet
- one possibility: new command TEST(SrcP,flag): is there a message from SrcP?

**Overlapping of computation and communication is critical for performance**

## Non-blocking SEND/RECEIVE : decouple synchronization from data transfer



- SrcP can push data out and move on
- Many variation: return to application program when
  - data is out on network?
  - data has been copied into an O/S buffer?
- Tag field on messages permits receiver to receive messages in an order different from order that they were sent by SrcP
- RECEIVE does not block
  - flag is set to true by O/S if data was transferred/false otherwise
- Applications program can test flag and take the right action
- What if DestP has not done a RECEIVE when data arrives from SrcP?
- Data is buffered in O/S buffers at DestP till application program does a RECEIVE

Can we eliminate waiting at SrcP ?

Can we eliminate buffering of data at DestP ?

## Asynchronous SEND/RECEIVE



- SEND returns as soon as O/S knows about what needs to be sent
- 'Flag1' set by O/S when data in x has been shipped out
- Application program continues, but must test 'flag1' before overwriting x
- RECEIVE is non-blocking:
  - returns before data arrives
    - tells O/S to place data in 'y' and set 'flag' after data is received
    - 'posting' of information to O/S
- 'Flag2' is written by O/S and read by application program on DestP
- Eliminates buffering of data in DestP O/S area if IRECEIVE is posted before message arrives at DestP

So far, we have looked at **point-to-point** communication

### **Collective communication:**

- patterns of group communication that can be implemented more efficiently than through long sequences of send's and receive's
- important ones:

- one-to-all broadcast**

(eg.  $A^*x$  implemented by rowwise distribution: all processors need  $x$ )

- all-to-one reduction**

(eg. adding a set of numbers distributed across all processors)

- all-to-all broadcast**

every processor sends a piece of data to every other processor

- one-to-all personalized communication**

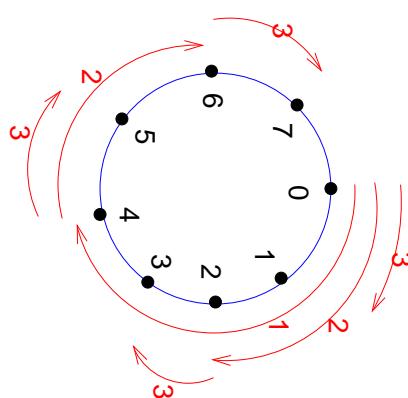
one processor sends a different piece of data to all other processors

- all-to-all personalized communication**

each processor does a one-to-all communication

## Example: One-to-all broadcast

(intuition: think ‘tree’)



Messages in each phase  
do not compete for links

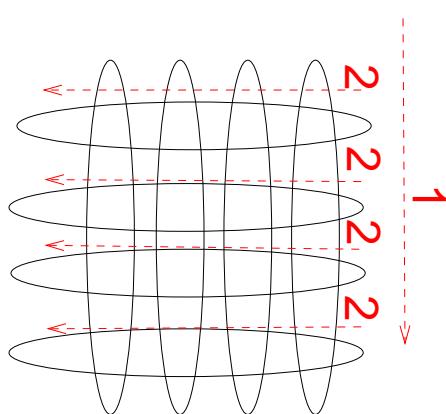
Assuming message size is small, time to send a message =  $T_s + h^*Th$   
where  $T_s$  = overhead at sender/receiver  
 $Th$  = time per hop

Total time for broadcast =  $T_s + Th^*P/2$

$$\begin{aligned} &+ T_s + Th^*P/4 \\ &+ \dots \\ &= T_s * \log P + Th^*(P-1) \end{aligned}$$

Reality check: Actually, a k-ary tree makes sense because processor 0 can send many messages by the time processor 4 is ready to participate in broadcast

Other topologies: use the same idea



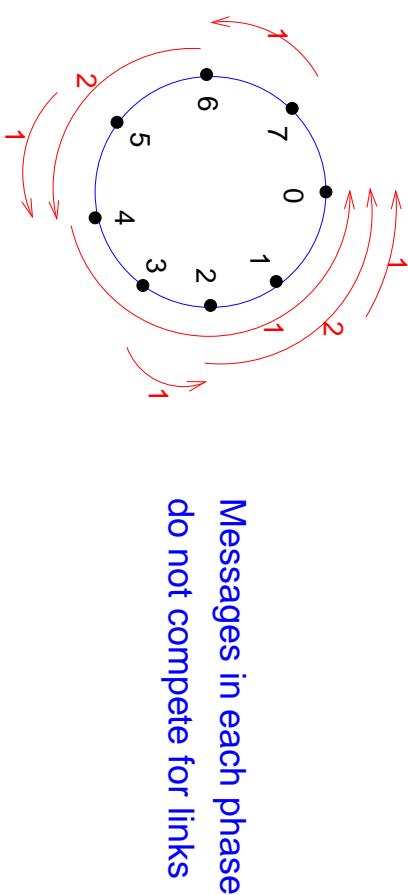
2-D Mesh

Step 1: Broadcast within row of originating processor

Step 2: Broadcast within each column in parallel

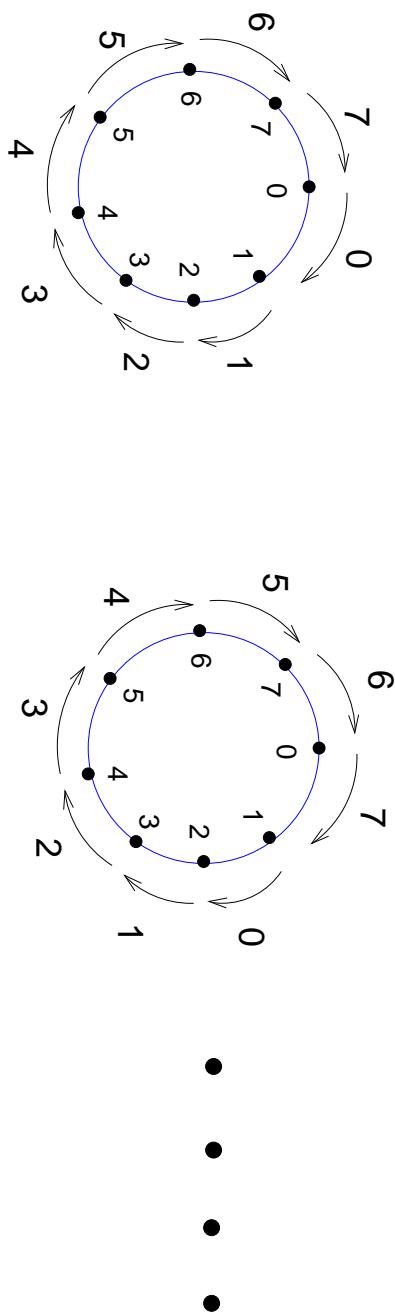
$$\text{Time} = T_s \log P + 2T_h^*(\sqrt{P} - 1)$$

## Example: All-to-one reduction



- Purpose: apply a commutative and associative operator (reduction operator) like  $+$ ,  $*$ , AND, OR etc to values contained in each node
- Can be viewed as inverse of one-to-all broadcast  
Same time as one-to-all broadcast
- Important use: determine when all processors are finished working  
(implementation of 'barrier')

## Example: All-to-all broadcast



- Intuition: cyclic shift register
- Each processor receives a value from one neighbor, stores it away, and sends it to next neighbor in the next phase.
- Total of  $(P-1)$  phases to complete all-to-all broadcast

Time =  $(T_s + Th) * (P-1)$  assuming message size is small

- Same idea can be applied to meshes as well:
  - first phase, all-to-all broadcast within each row
  - second phase, all-to-all broadcast within each column

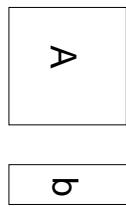
## A Message-passing Program

## **MPI: Message-Passing Interface**

### **Goal: Portable Parallel Programming for Distributed Memory Computers**

- Lots of vendors of Distributed Memory Computers:  
IBM, NCube, Intel, CM-5, .....
- Each vendor had its own communication constructs  
=> porting programs required changing parallel programs  
even to go from one distributed memory platform to another!
- MPI goal: standardize message passing constructs syntax and semantics
- Mid 1994: MPI-1 standard out and several implementations available (SP-2)

## Write an MPI program to perform matrix-vector multiply



- Style of programming: **Master-Slave**
- one master, several slaves
- master co-ordinates activities of slaves
- Master initially owns all rows of A and vector b
- Master broadcasts vector b to all slaves
- Slaves are **self-scheduled**
  - each slave comes to master for work
  - master sends a row of matrix to slave
  - slave performs product, returns result and asks for more work
- Very naive algorithm, but it's a start.

## Key MPI Routines we will use:

**MPI\_INIT** : Initialize the MPI System

**MPI\_COMM\_SIZE**: Find out how many processes there are

**MPI\_COMM\_RANK**: Who am I?

**MPI\_SEND**: Send a message

**MPI\_SEND(address,count,datatype,DestP,tag,comm)**

permits entire data structures  
to be sent with one command

↑  
identifies process group

**MPI\_RECV**: Receive a message (blocking receive)

**MPI\_FINALIZE**: Terminate MPI

**MPI\_BCAST**: Broadcast

```

c      COMMON PROGRAM EXECUTED BY BOTH MASTER AND SLAVES
c*****
c      matmul.f - matrix - vector multiply, simple self-scheduling version
c*****



program main

include 'mpif.h'

integer MAX_ROWS, MAX_COLS, rows, cols
parameter (MAX_ROWS = 1000, MAX_COLS = 1000)
double precision a(MAX_ROWS,MAX_COLS), b(MAX_COLS), c(MAX_COLS)
double precision buffer(MAX_COLS), ans

integer myid, master, numprocs, ierr, status(MPI_STATUS_SIZE)
integer i, j, nmsent, numrcvd, sender, job(MAX_ROWS)
integer rowtype, anstype, donetype


call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )

master    = 0
rows      = 100
cols      = 100

if ( myid .eq. master ) then
  c      master initializes and then dispatches
  .....
else
  c      slaves receive b, then compute dot products until done message
  .....
endif

200 call MPI_FINALIZE(ierr)
stop
end

```

```

c CODE EXECUTED BY MASTER
if ( myid .eq. master ) then
c
c   master initializes and then dispatches
c   initialize a and b to arbitrary values
do 20 i = 1,cols
b(i) = 1
do 10 j = 1,rows
a(i,j) = i
10    continue
20    continue

numsent = 0
numrcvd = 0

c   send b to each other process
call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
$           MPI_COMM_WORLD, ierr)

c   send a row to each other process
do 40 i = 1,numprocs-1
do 30 j = 1,cols
buffer(j) = a(i,j)
30    continue

call MPI_SEND(buffer, cols, MPI_DOUBLE_PRECISION, i,
$             rowtype, MPI_COMM_WORLD, ierr)

c Job(i) = NUMBER OF ROW CURRENTLY BEING PROCESSED BY PROCESS i.
        job(i) = i
        numsent = numsent+1
40    continue

do 70 i = 1,rows

```

```

call MPI_RECV(ans, 1, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,
$ anstype, MPI_COMM_WORLD, status, ierr)
sender = status(MPI_SOURCE)
c(job(sender)) = ans

if (numsent .lt. rows) then
do 50 j = 1,cols
  buffer(j) = a(numsent+1,j)
  continue
50
  call MPI_SEND(buffer, cols, MPI_DOUBLE_PRECISION, sender,
$ rowtype, MPI_COMM_WORLD, ierr)
  job(sender) = numsent+1
  numsent = numsent+1
else
  call MPI_SEND(1, 1, MPI_INTEGER, sender, donetype,
$ MPI_COMM_WORLD, ierr)
endif
70
  continue

c
  print out the answer
do 80 i = 1,cols
print *, "c(", i, ") = ", c(i)
80
  continue

```

c CODE EXECUTED BY SLAVES

```
c
slaves receive b, then compute dot products until done message
call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
$ MPI_COMM_WORLD, ierr)
90 call MPI_RECV(buffer, cols, MPI_DOUBLE_PRECISION, master,
$ MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
if (status(MPI_TAG) .eq. donetype) then
  go to 200
else
  ans = 0.0
  do 100 i = 1,cols
    ans = ans+buffer(i)*b(i)
  continue
100 call MPI_SEND(ans, 1, MPI_DOUBLE_PRECISION, master, ansstype,
$ MPI_COMM_WORLD, ierr)
  go to 90
endif
endif

200 call MPI_FINALIZE(ierr)
stop
end
```

```

C*****matrix-vector multiply, simple self-scheduling version
C*****
program main

include 'mpif.h'

integer MAX_ROWS, MAX_COLS, rows, cols
parameter (MAX_ROWS = 1000, MAX_COLS = 1000)
double precision a(MAX_ROWS,MAX_COLS), b(MAX_COLS), c(MAX_COLS)
double precision buffer(MAX_COLS), ans

integer myid, master, numprocs, ierr, status(MPI_STATUS_SIZE)
integer i, j, numsent, numrcvd, sender, job(MAX_ROWS)
integer rowtype, anstype, donetype

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
if (numprocs .lt. 2) then
  print *, "Must have at least 2 processes!"
  MPI_Abort( MPI_COMM_WORLD, 1 )
stop
endif
print *, "Process ", myid, " of ", numprocs, " is alive"

rowtype = 1
anstype = 2
donetype = 3

master = 0
rows = 100
cols = 100

```

```

if ( myid .eq. master ) then
c
master initializes and then dispatches
c
initialize a and b
do 20 i = 1,cols
b(i) = 1
do 10 j = 1,rows
a(i,j) = i
10
continue
20
continue

numsent = 0
numrcvd = 0

c
send b to each other process
call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
$ MPI_COMM_WORLD, ierr)

c
send a row to each other process
do 40 i = 1,numprocs-1
do 30 j = 1,cols
buffer(j) = a(i,j)
30
continue
call MPI_SEND(buffer, cols, MPI_DOUBLE_PRECISION, i,
$ rowtype, MPI_COMM_WORLD, ierr)
job(i) = i
nusent = numsent+1
40
continue

do 70 i = 1,rows
call MPI_RECV(ans, 1, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,
$ anstype, MPI_COMM_WORLD, status, ierr)
sender = status(MPI_SOURCE)

```

```

c(job(sender)) = ans

if (numsent .lt. rows) then
do 50 j = 1,cols
  buffer(j) = a(numsent+1,j)
  continue
50
  call MPI_SEND(buffer, cols, MPI_DOUBLE_PRECISION, sender,
                rowtype, MPI_COMM_WORLD, ierr)
  job(sender) = numsent+1
  numsent      = numsent+1

else
  call MPI_SEND(1, 1, MPI_INTEGER, sender, doneType,
                MPI_COMM_WORLD, ierr)
endif
70
  continue

c
  print out the answer
do 80 i = 1,cols
  print *, "c(", i, ") = ", c(i)
80
  continue

else
  slaves receive b, then compute dot products until done message
  call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
                 MPI_COMM_WORLD, ierr)
90
  call MPI_RECV(buffer, cols, MPI_DOUBLE_PRECISION, master,
                MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
  if (status(MPI_TAG) .eq. doneType) then
    go to 200
  else
    ans = 0.0
    do 100 i = 1,cols
      ans = ans+buffer(i)*b(i)
100

```

```
100      continue
        call MPI_SEND(ans, 1, MPI_DOUBLE_PRECISION, master, ansstype,
$          MPI_COMM_WORLD, ierr)
        go to 90
      endif
    endif

200  call MPI_FINALIZE(ierr)
stop
end
```

This style of parallel programming is called

Single Program Multiple Data (SPMD) programming

All processors execute the same code but branch on their IDs to perform disjoint activities.

In principle, each processor could run different programs, but this is not very common (cf. CSP, OCCAM, ...).