

CESE4130: Computer Engineering

2024-2025, lecture 8

Going Parallel

Computer Engineering Lab

Faculty of Electrical Engineering, Mathematics & Computer Science

2024-2025

Announcement

- Lab 2 went much better!
- Vivado is still the most challenging pcomponent

Course objectives

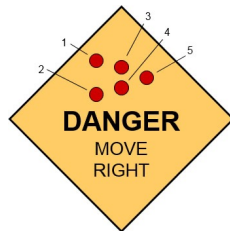
- Describe number representation systems and inter-conversion.
- Perform binary arithmetic operation such as addition and multiplication.
- Explain basic concepts of computer architecture.
- Use logic gates to implement simple combinational circuits.
- Explain system software and operating systems fundamentals, task management, synchronization, compilation, and interpretation.
- Use design and automation tools to perform synthesis and optimization.

Objectives

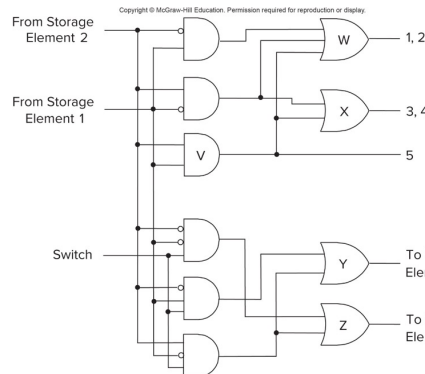
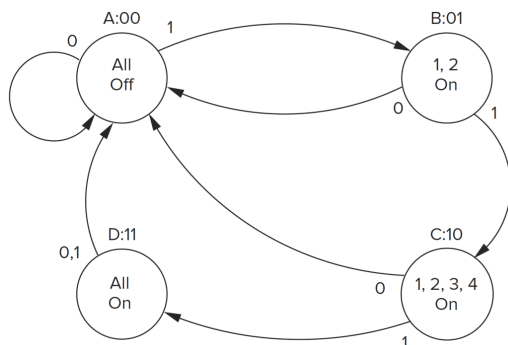
- Understand systems with multiple processors
- Explain the different types of parallel machines
- Get the basic of a widely used massively parallel platform

Recap

- From bits to gates to functional units to u-architecture to computer architecture (✓)
- Now also the main memory should be more or less clear
- Differences between SRAM and DRAM memory cells
- Anything else I miss (?)
 - our main goal is **"to remove magic"** as you remember

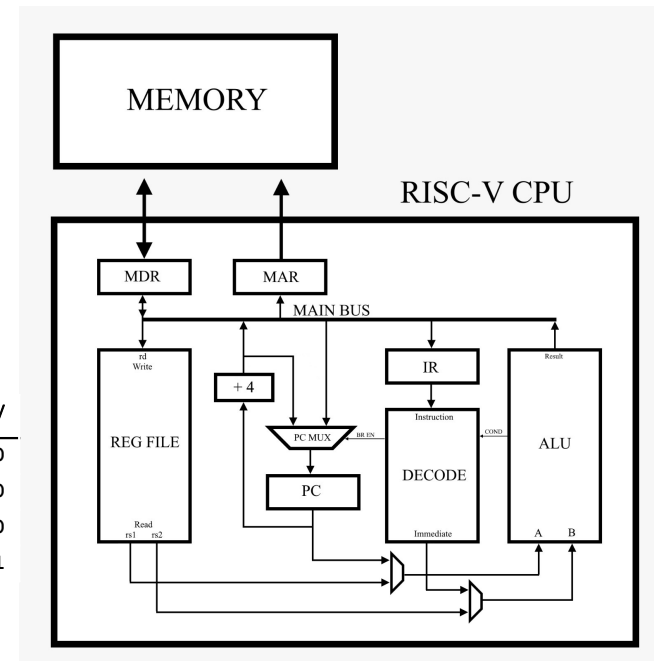


Equivalent representations



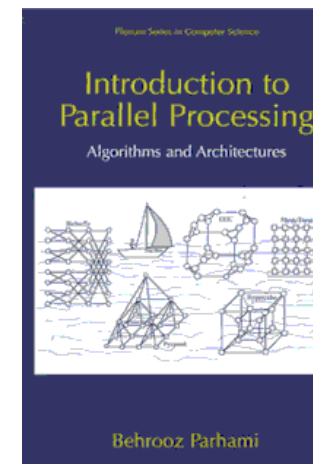
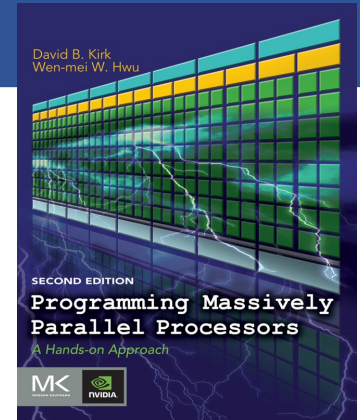
Switch	S[1:0]	S'[1:0]
0	00	00
0	01	00
0	10	00
0	11	00
1	00	01
1	01	10
1	10	11
1	11	00

S[1:0]	W	X	V
00	0	0	0
01	1	0	0
10	1	1	0
11	1	1	1

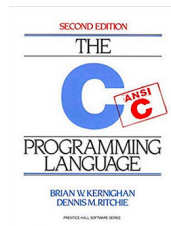
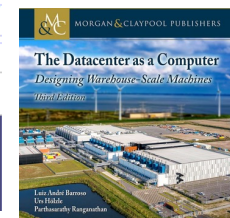


Overview

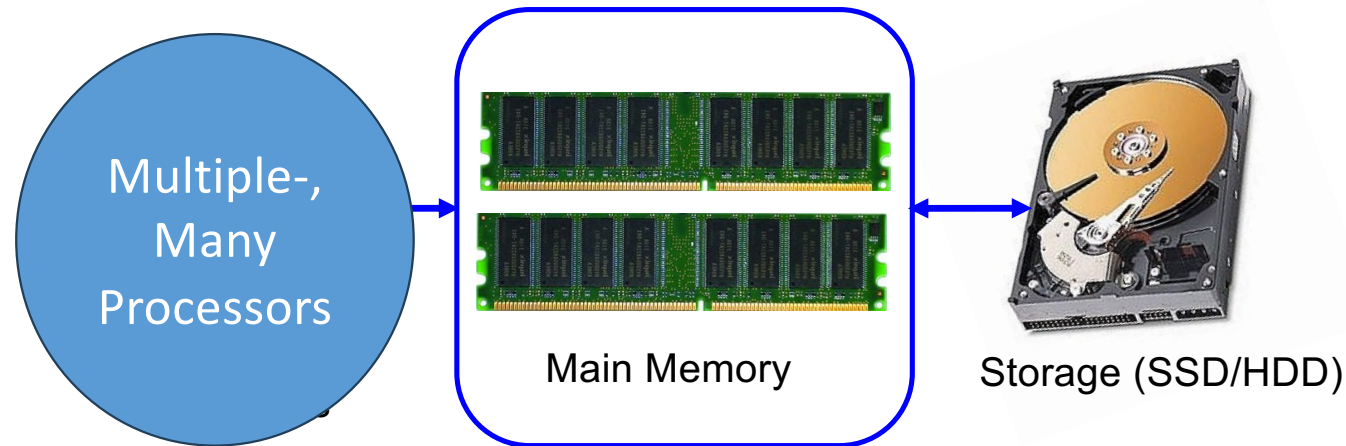
- The lecture material is collected from various sources
- About CUDA, please refer to Wen-Mei and David
 - <https://shop.elsevier.com/books/programming-massively-parallel-processors/hwu/978-0-323-91231-0>
- Also NVIDIA has a lot of tutorials and recorded lectures
 - <https://developer.nvidia.com/educators/existing-courses>
- Parallel Processing course, again Behrooz Parhami
 - https://web.ece.ucsb.edu/~parhami/text_par_proc.htm#slides



Maybe even

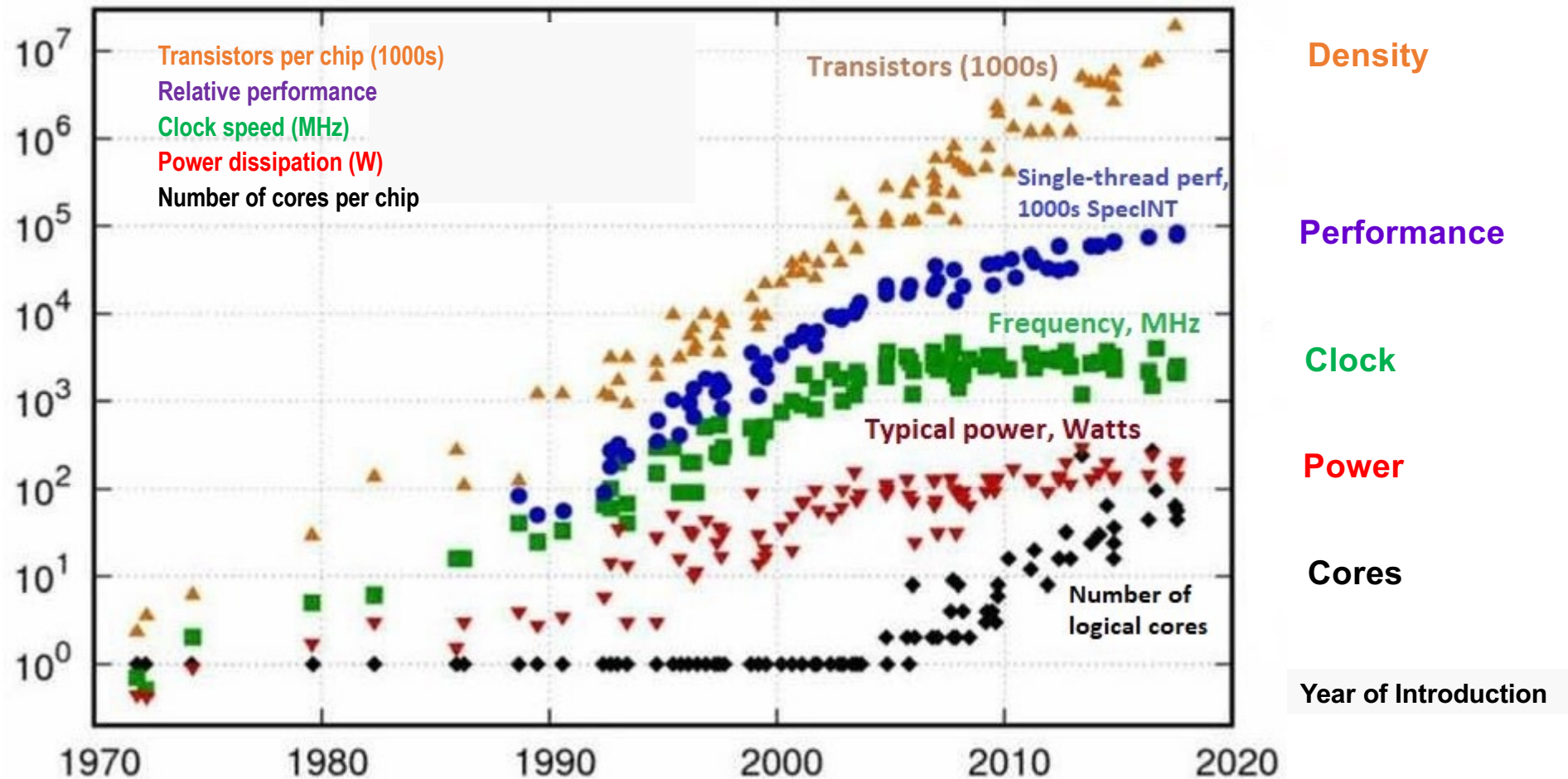


The Main Memory System (review)



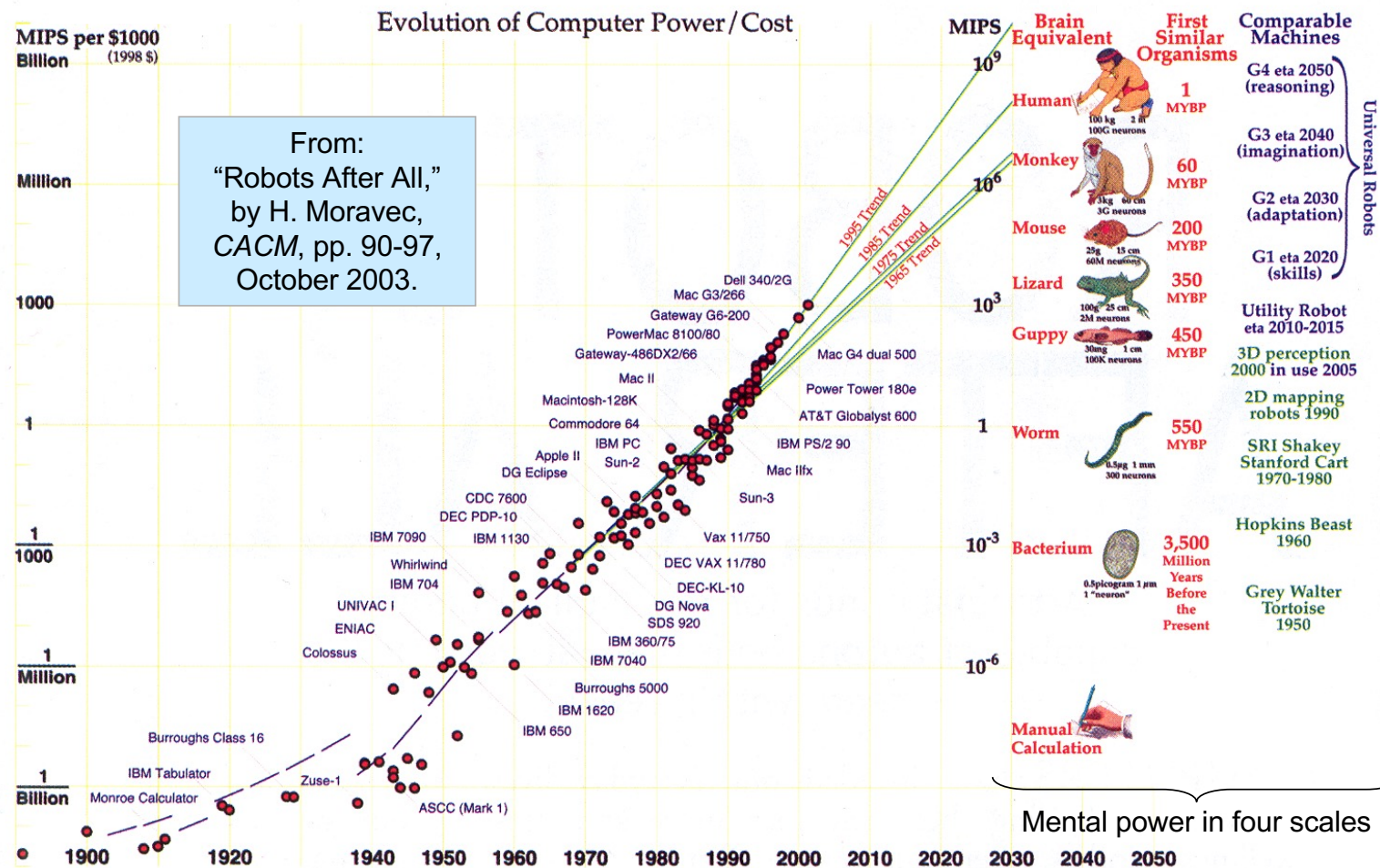
- Main memory is a critical component of all computing systems: server, mobile, embedded, desktop, sensor, etc
- Main memory system must scale (in *size, technology, efficiency, cost, and management algorithms*) to maintain performance growth and technology scaling benefits

Trends in Processor Chip Density, Performance, Clock Speed, Power, and #Cores



Original data up to 2010 collected/plotted by M. Horowitz et al.; Data for 2010-2017 extension collected by K. Rupp

Evolution of Computer Performance/Cost (??)



Everyday Parallelism

- Juggling -- event-based computation
- House construction -- parallel tasks, wiring and plumbing performed at once
- Assembly line manufacture -- pipelining, many instances in process at once
- Call center -- independent tasks executed simultaneously

How do we describe execution of tasks?

Parallel vs Distributed Computing

- Comparisons are often matters of degree

<i>Characteristic</i>	<i>Parallel</i>	<i>Distributed</i>
Overall Goal	Speed	Convenience
Interactions	Frequent	Infrequent
Granularity	Fine	Coarse
Reliable	Assumed	Not Assumed

Parallel vs Concurrent

- In OS and DB communities execution of multiple threads is **logically** simultaneous
- In Architecture and HPC communities execution of multiple threads is **physically** simultaneous
- Issues are often the same, say with respect to *rac*es
- Parallelism can achieve states that are impossible with concurrent execution because two events happen at once

Consider A Simple Task ...

- Adding a sequence of numbers $A[0], \dots, A[n-1]$
- Standard way to express it

```
sum = 0;
for (i=0; i<n; i++) {
    sum += A[i];
}
```

- Semantics require:
 $(\dots((\text{sum} + A[0]) + A[1]) + \dots) + A[n-1]$
 - That is, **sequential**
- Can it be executed in parallel?

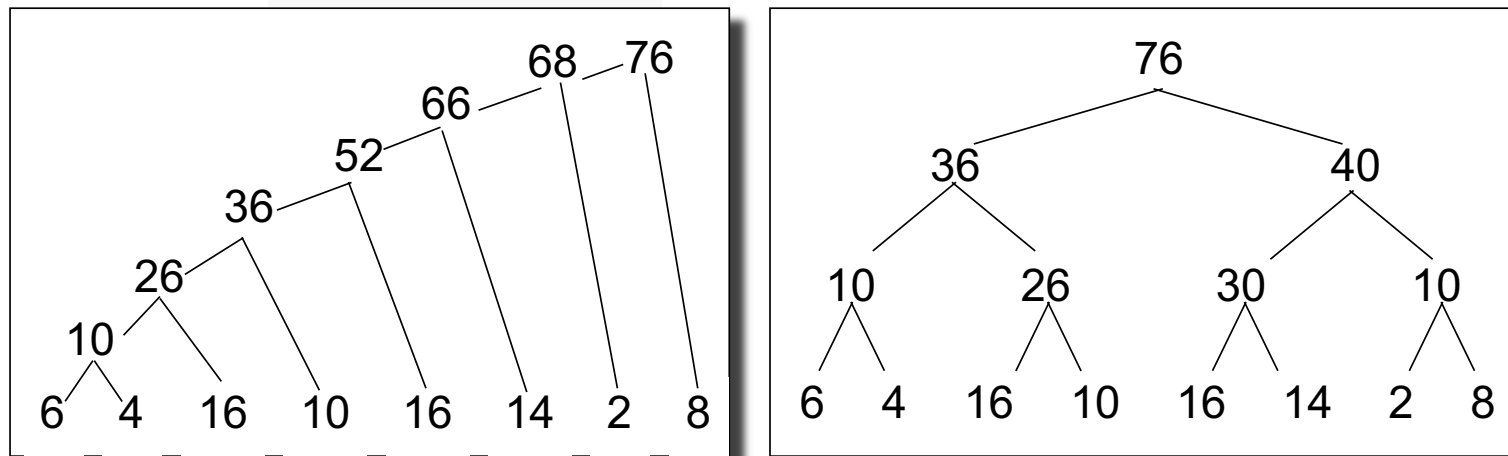
Parallel Summation

- To sum a sequence in parallel
 - add pairs of values producing 1st level results,
 - add pairs of 1st level results producing 2nd level results,
 - sum pairs of 2nd level results ...
- That is,

$$(\dots((A[0]+A[1]) + (A[2]+A[3])) + \dots + (A[n-2]+A[n-1]))\dots)$$

Express the Two Formulations

- Graphic representation makes difference clear



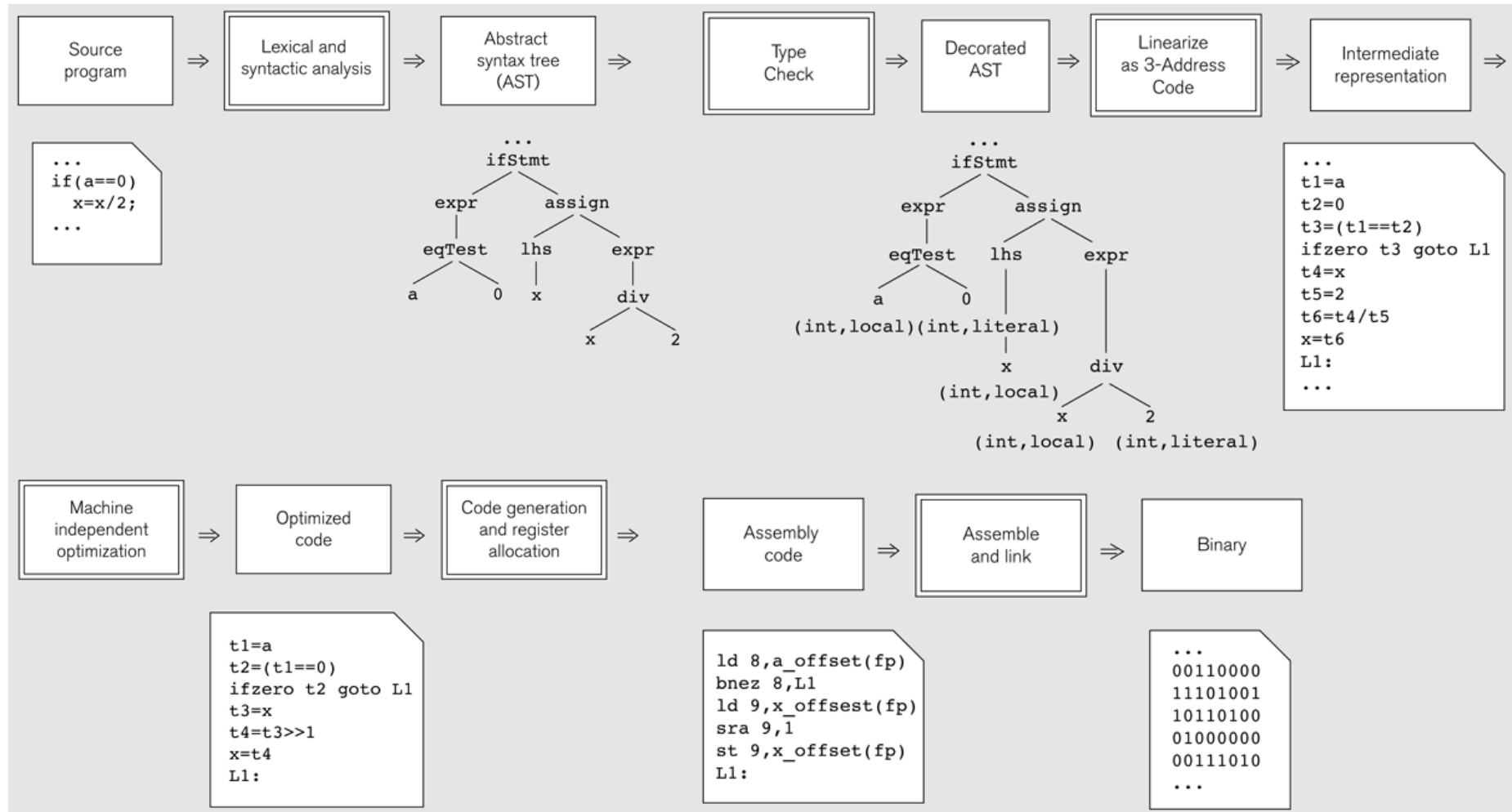
- Same number of operations; different order

Simple

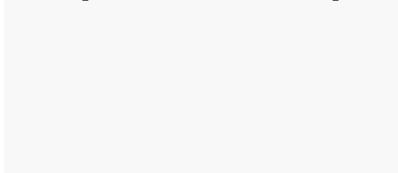
The Dream ...

- Since 70s (Illiac IV days) the dream has been to **compile sequential programs** into parallel object code
- Many decades of continual, well-funded research by smart people implies it's hopeless
- For a tight loop summing numbers, its doable
- For other computations it has proved **extremely challenging** to generate parallel code, even with pragmas or other assistance from programmers

Compilers



What's the Problem?

- It's not likely a compiler will produce parallel code from a C specification any time soon... 
- Fact: For most computations, a "best" sequential solution (practically, not theoretically) and a "best" parallel solution are usually fundamentally different ...
- Different solution paradigms imply computations are not "simply" related
- Compiler transformations generally preserve the solution paradigm

Therefore... the programmer must discover the || solution

A Related Computation

- Consider computing the prefix sums

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

$A[i]$ is the sum of the first $i + 1$ elements

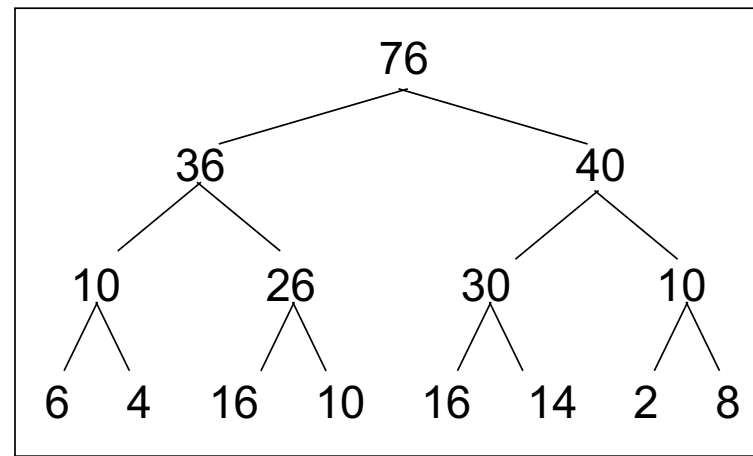
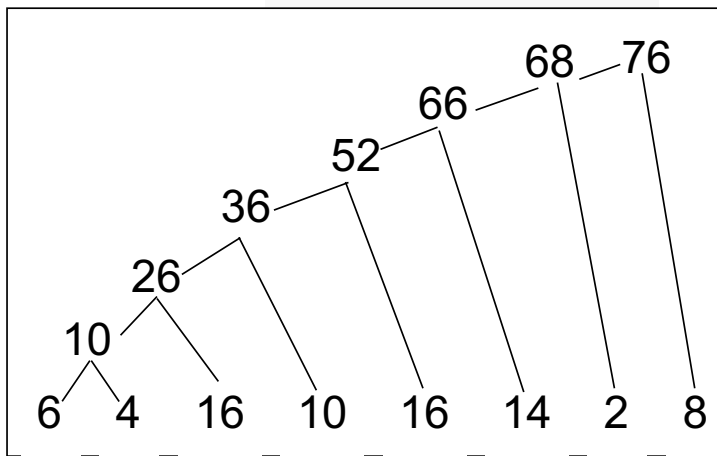
- Semantics ...

- $A[0]$ is unchanged
- $A[1] = A[1] + A[0]$
- $A[2] = A[2] + (A[1] + A[0])$
- ...
- $A[n-1] = A[n-1] + (A[n-2] + (\dots (A[1] + A[0]) \dots)$

What advantage can $| |$ ism give?

Comparison of Paradigms

- The sequential solution computes the prefixes ... the parallel solution computes only the last



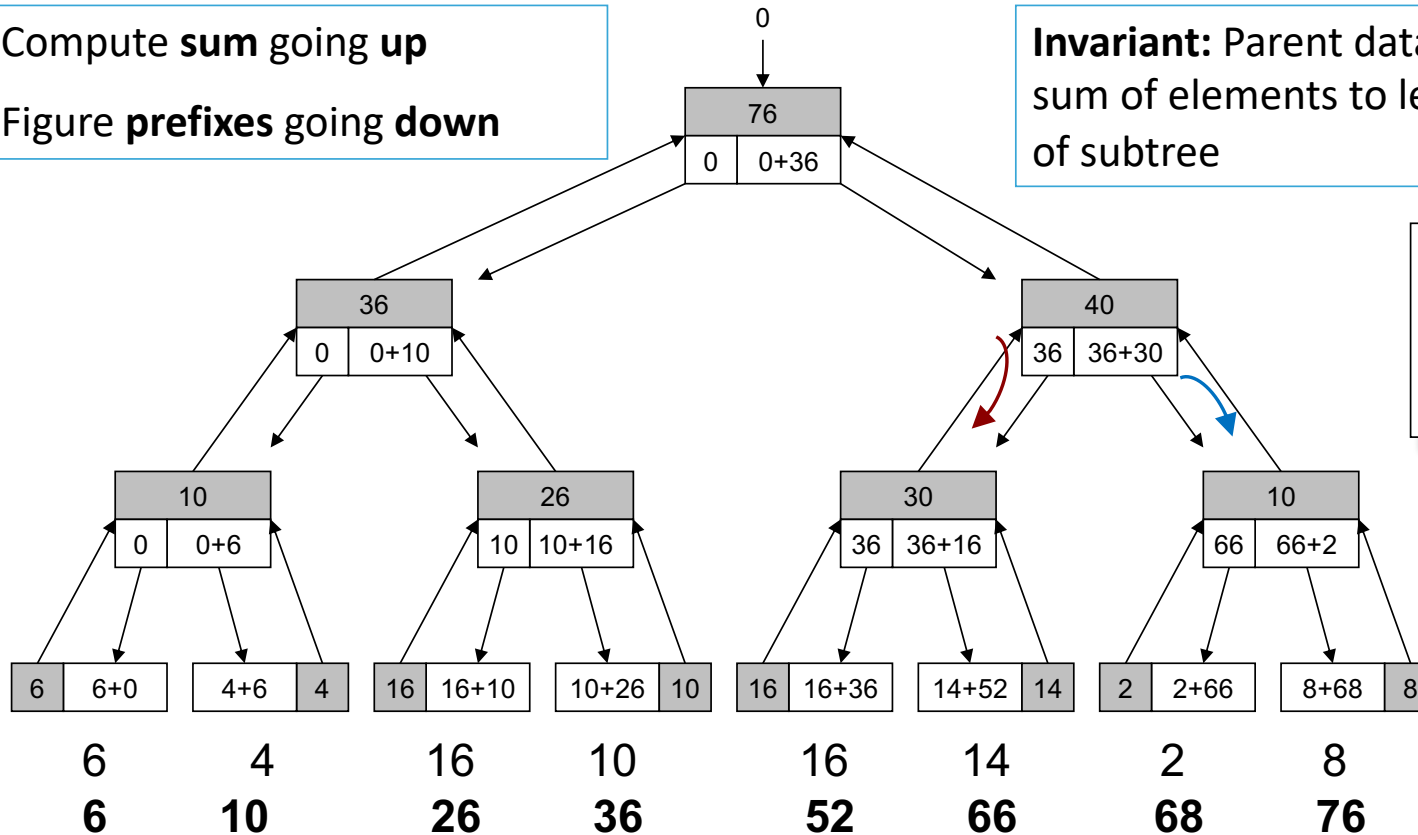
- or does it?

Parallel Prefix Algorithm

Compute **sum** going **up**
Figure **prefixes** going **down**

Invariant: Parent data is
sum of elements to left
of subtree

The Ladner-Fischer algorithm
requires $2\log n$ time, twice as
much as simple tournament
global sum, not linear time



Original paper: R.E. Ladner and M. J. Fischer, Parallel Prefix Computation, *Journal of the ACM*, 27(4):831-838, 1980

Fundamental Tool of || Programming (useful for wide class of II operations)

Parallel Compared to Sequential Programming

- Has different costs, different advantages
- Requires different, unfamiliar algorithms
- Must use different abstractions
- More complex to understand a program's behavior
- More difficult to control the interactions of the program's components
- Knowledge/tools/understanding more primitive

*The parallel approach to computing ... does require that some **original thinking** be done about **numerical analysis and data management** in order to secure efficient use. In an environment which has represented the absence of the need to think as the highest virtue, this is a decided disadvantage.*

-- Dan Slotnick, 1967

But also understand your II machine

Consider a Simple Problem

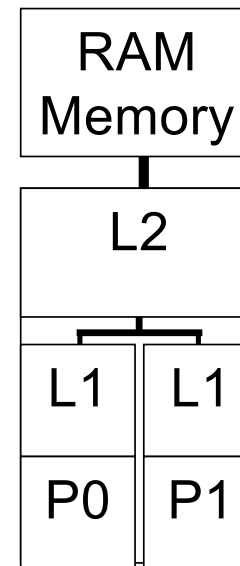
- Count the 3s in `array[]` of `length` values
- Definitional solution ...
 - Sequential program

```
count = 0;
for (i=0; i<length; i++)
{
    if (array[i] == 3)
        count += 1;
}
```

Write A Parallel Program

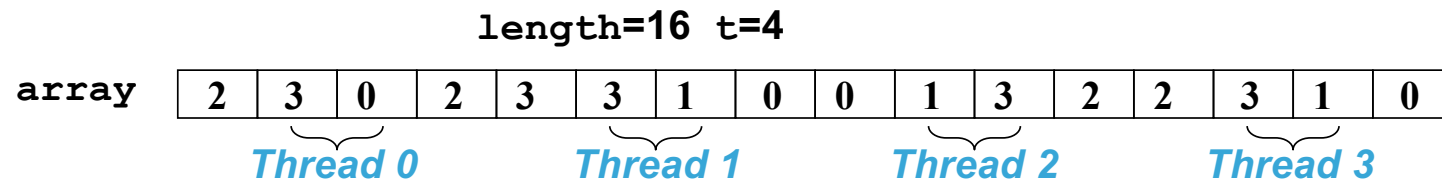
- Need to know something about the machine
... use multicore architecture

How would you solve it in parallel?



Divide Into Separate Parts (Divide and Conquer)

- Threading solution -- prepare for Multi-Threaded procs

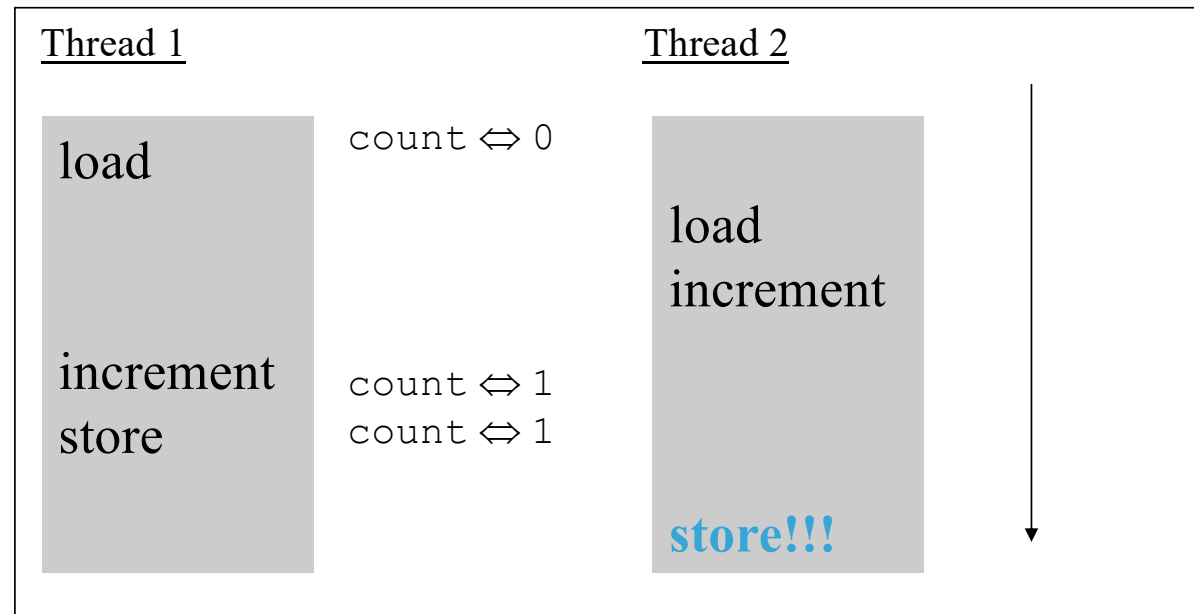


```
int length_per_thread = length/t;
int start = id * length_per_thread;
for (i=start; i<start+length_per_thread; i++)
{
    if (array[i] == 3)
        count += 1;
}
```

Doesn't actually get the right answer

Races!

- Two processes interfere on memory writes



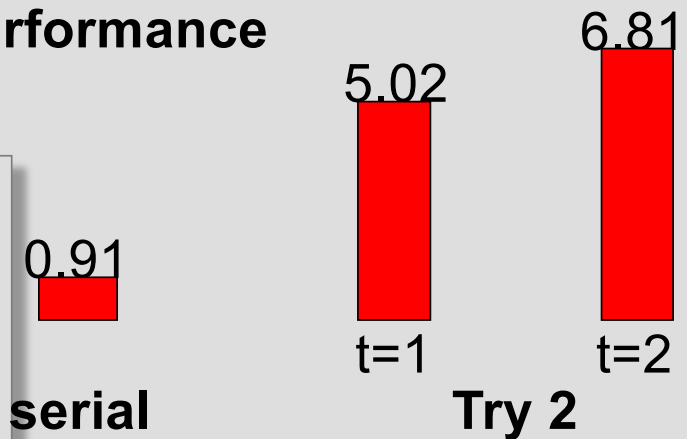
Try 1 (mmm)

Protect Memory References

- Protect Memory References

```
mutex m;  
for (i=start; i<start+length_per_thread; i++)  
{  
    if (array[i] == 3)  
    {  
        mutex_lock(m);  
        count += 1;  
        mutex_unlock(m);  
    }  
}
```

Performance



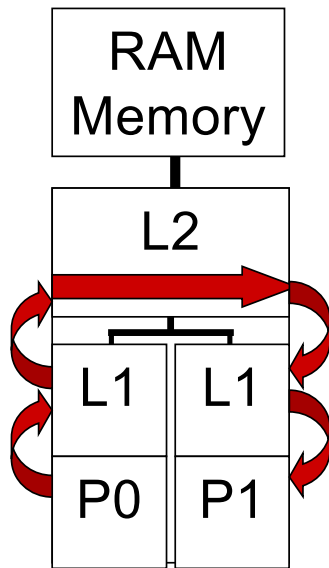
Correct Results but SLOW
Serializing at the mutex

- The processors wait on each other

Try 2 (?)

Closer Look: Motion of `count`, `m`

- Lock Reference and Contention



```
mutex m;  
for (i=start; i<start+length_per_thread; i++)  
{  
    if (array[i] == 3)  
    {  
        mutex_lock(m);  
        count += 1;  
        mutex_unlock(m);  
    }  
}
```


Accumulate Into Private Count

- Each processor adds into its own memory; combine at the end

```
for (i=start; i<start+length_per_thread; i++)  
{  
    if (array[i] == 3)  
    {  
        private_count[t] += 1;  
    }  
}  
mutex_lock(m);  
count += private_count[t];  
mutex_unlock(m);
```

Performance

0.91

0.91

1.15

serial

t=1

t=2

Try 3

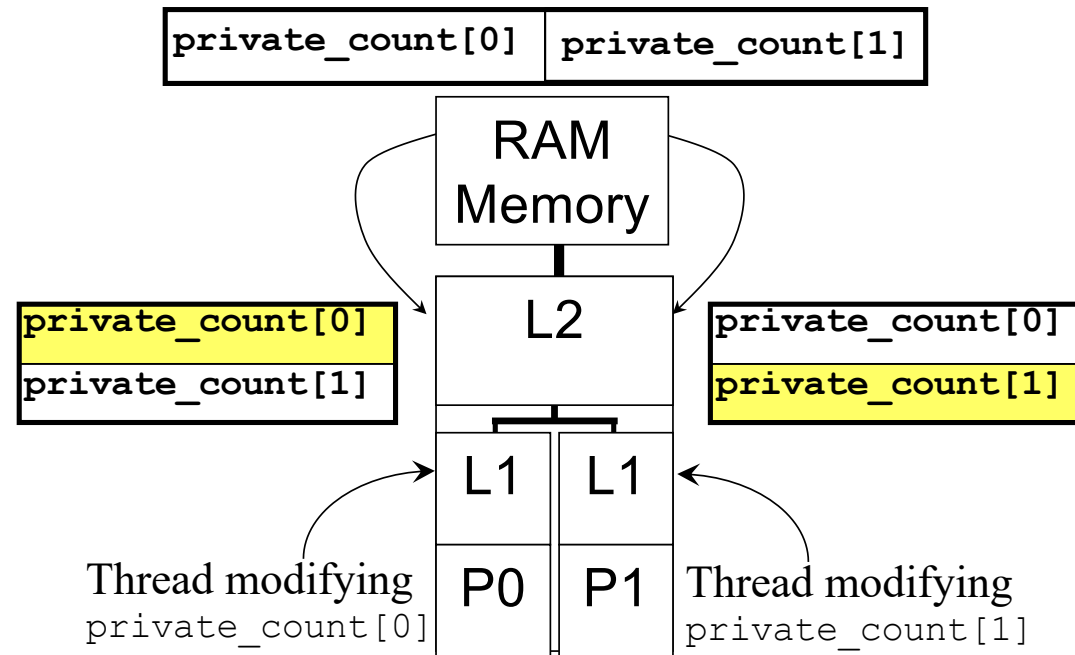
Keeping up but NOT gaining

- Sequential and one processor match, but it's a loss with two processors

Try 3

False Sharing

- Private var \neq private cache-line



Force Into Different Cache Lines

- Padding the private variables forces them into separate cache lines and removes false sharing

```
struct padded_int  
{ int value;  
  char padding[128];  
} private_count[MaxThreads];
```

Performance

0.91



serial

0.91



t=1

0.51



t=2

Try 4

Success!!!

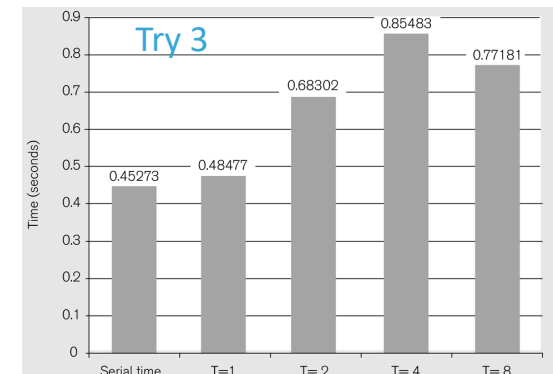
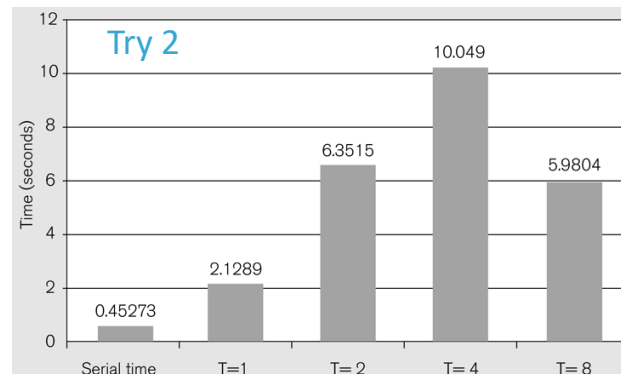
- Two processors are almost twice as fast
- Is this the best solution???

Try 4

31

Count 3s Summary

- Recapping the experience of writing the program, we
 - Wrote the obvious “break into blocks” program
 - We needed to protect the count variable
 - We got the right answer, but the program was slower ... lock congestion
 - Privatized memory and 1-process was fast enough, 2-processes slow ... false sharing
 - Separated private variables to own cache line
- What happens when more processors are available?
 - 4 processors
 - 8 processors (look in the book)
 - 256 processors
 - 32,768 processors



Von Neumann (RAM) Model

- Call the 'standard' model of a random access machine (RAM) the von Neumann model
 - A processor interpreting 3-address instructions
 - PC pointing to the next instruction of program in memory
 - "Flat," randomly accessed memory requires 1 time unit
 - Memory is composed of fixed-size addressable units
 - One instruction executes at a time, and is completed before the next instruction executes
- The model is not literally true, e.g., memory is hierarchical but made to "look flat"

C directly implements this model in a HLL

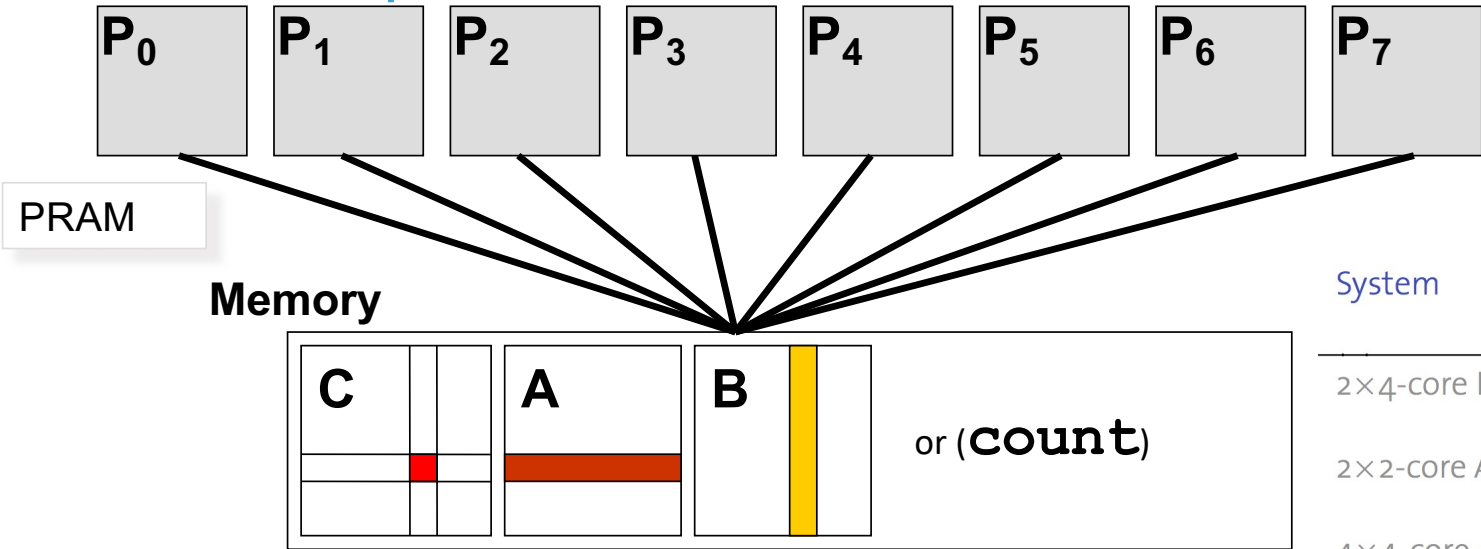
Parallel RAM (PRAM) Often Proposed As A Candidate

- PRAM ignores memory organization, collisions, latency, conflicts, etc.
- Ignoring these are claimed to have benefits ...
 - Portable everywhere since it is very general
 - It is a simple programming model ignoring only insignificant details -- off by “only log P”
 - Ignoring memory difficulties is OK because hardware can “fake” a shared memory
 - Good for getting started: Begin with PRAM then refine the program to a practical solution if needed

What is the best II programming language?

PRAM has any number of processors

- Every proc references any memory in “time 1”
- Memory read/write collisions must be resolved
- SMPs implement PRAMs for small P ... not scalable



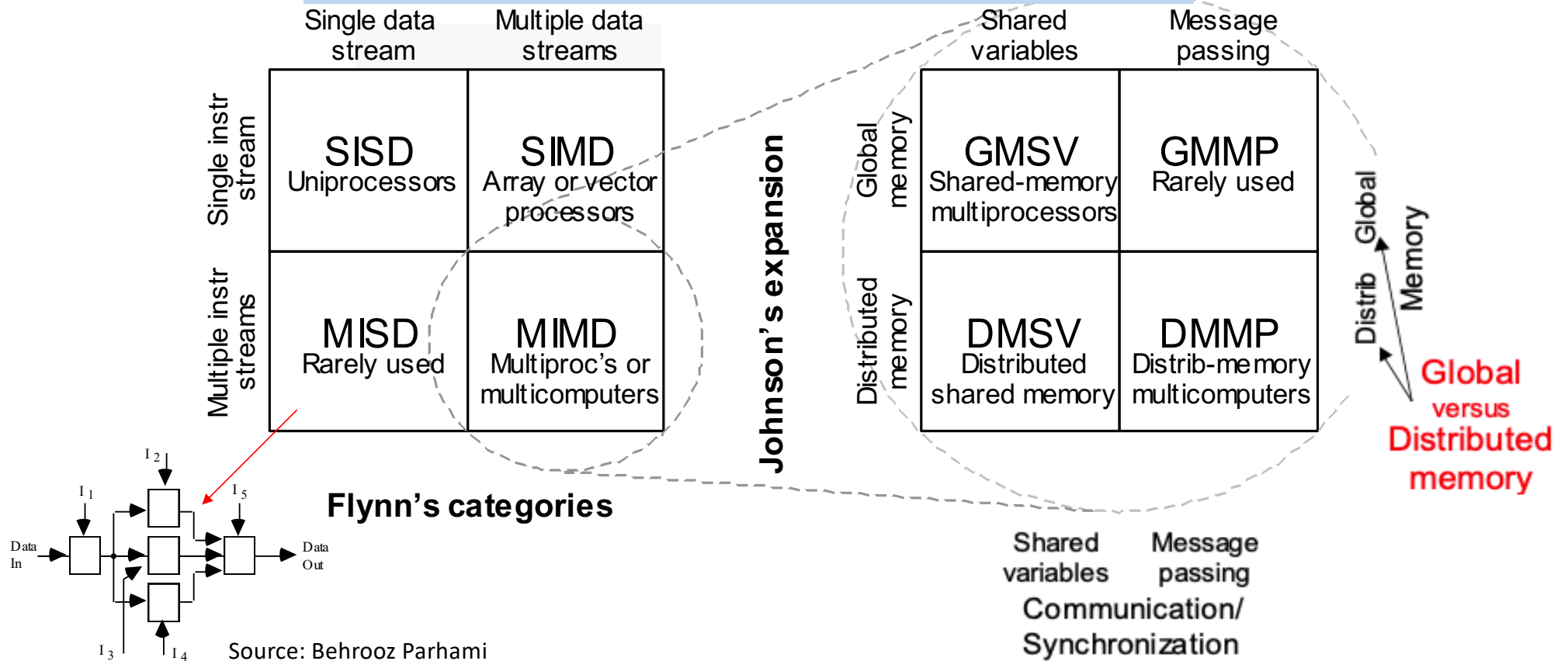
Accessing remote memory is slower!

CMP	AMD	100
SMP	Sun Fire E25K	400-660
Cluster	Itanium + Myrinet	4,100-5,100
Super	BlueGene/L	5,000

System	Cache	Latency cycles	Throughput msgs/kcycle
2×4-core Intel	shared	180	11.97
	non-shared	570	3.78
2×2-core AMD	same die	450	3.42
	one-hop	532	3.19
4×4-core AMD	shared	448	3.57
	one-hop	545	3.53
8×4-core AMD	two-hop	659	3.19
	shared	538	2.77
	one-hop	613	2.79
	two-hop	682	2.71

Types of Parallelism: A Taxonomy

The Flynn-Johnson classification of computer systems.



What is the II model?

SIMD

- Single Instruction, Multiple Data
- One instruction stream is broadcast to all processors
- Each processor, also called a *processing element* (PE), is usually simplistic and logically is essentially an ALU
 - PEs do not store a copy of the program nor have a program control unit
- Individual processors can remain idle during execution of segments of the program (based on a data test)

SIMD (cont)

- All active processors execute the same instruction **synchronously**, but on different data
- Technically, on a memory access, all active processors must access the *same location* in their local memory
 - This requirement is sometimes relaxed a bit
- The data items form an array (or vector) and an instruction can act on the complete array in one cycle
- Examples:
 - ILLIAC IV (1974) was the first SIMD computer
 - The STARAN and MPP
 - Connection Machine CM2 (by Thinking Machines)
 - MasPar MP-1 (for Massively Parallel) computers

Vector machines, VLIW, etc.

How to view an SMID machine ?

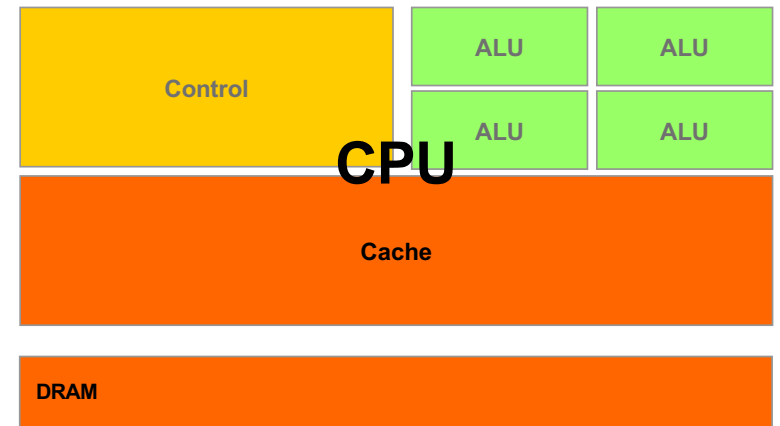
Think of all soldiers in a unit

- The commander selects certain soldiers as active, e.g., the first row
- The commander barks out an order to all the active soldiers, who execute the order synchronously
- The remaining soldiers do not execute orders until they are re-activated

Single Program, Multiple Data (SPMD), e.g. CUDA != SIMD

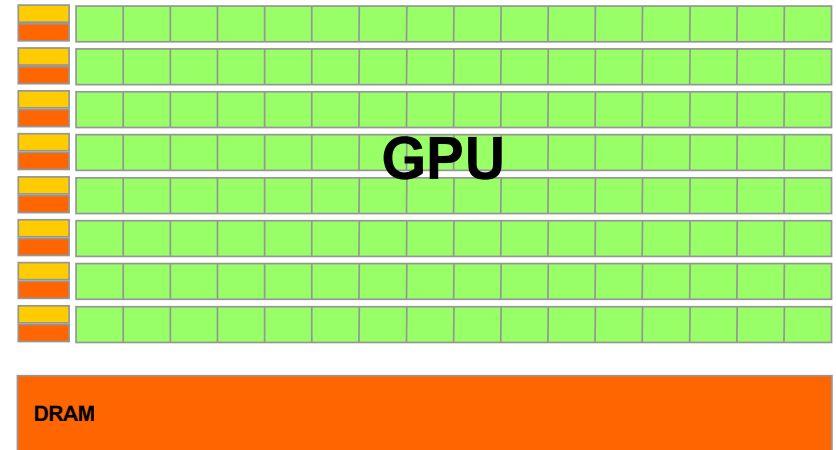
CPUs: Latency Oriented Design

- High clock frequency
- Large caches
 - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
 - Branch prediction for reduced branch latency
 - Data forwarding for reduced data latency
- Powerful ALU
 - Reduced operation latency

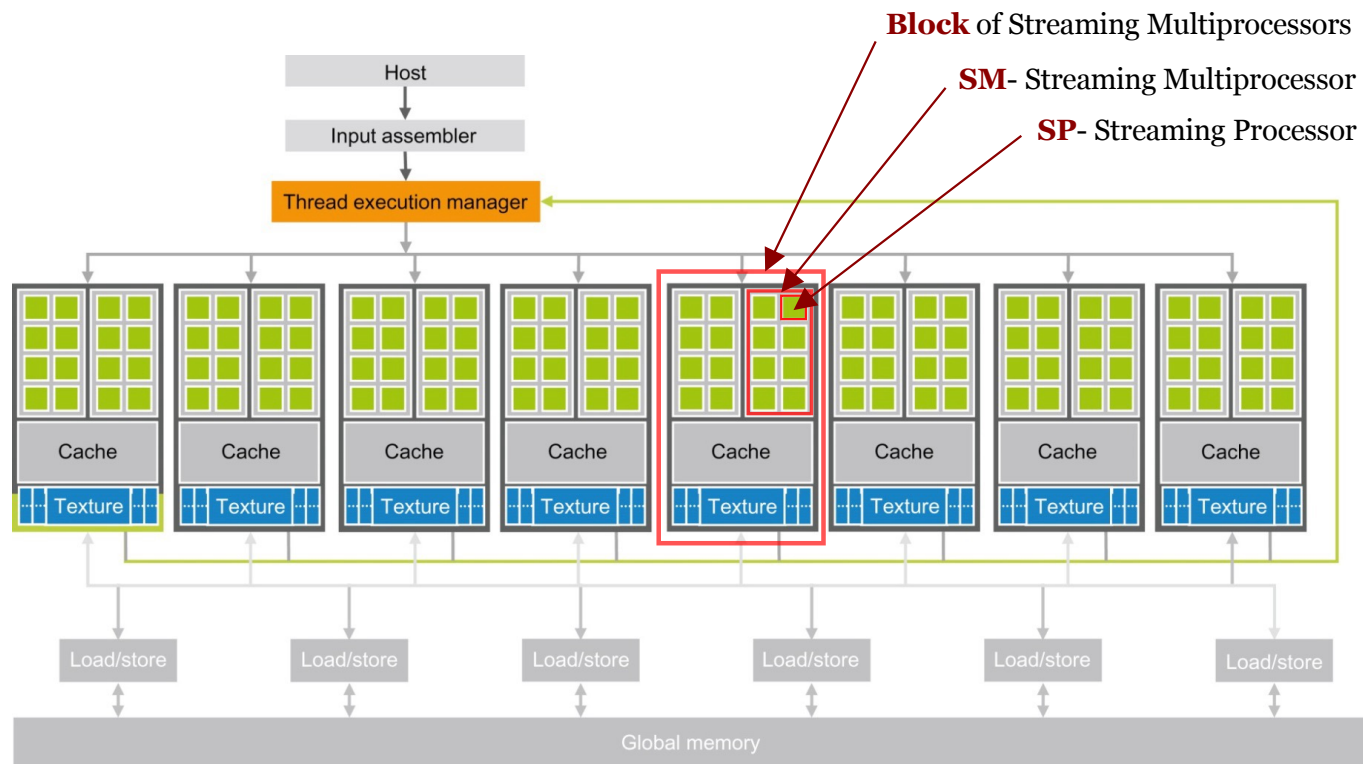


GPUs: Throughput Oriented Design

- Moderate clock frequency
- Small caches
 - To boost memory throughput
- Simple control
 - No branch prediction
 - No data forwarding
- Energy efficient ALUs
 - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies

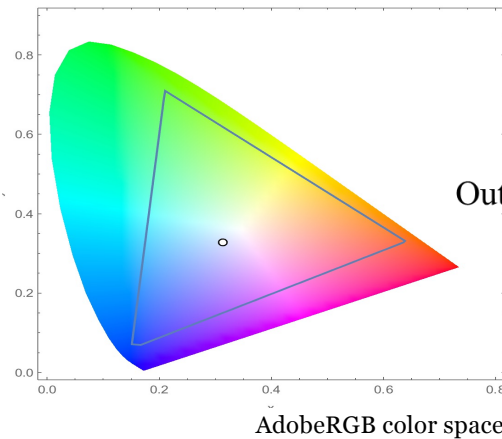
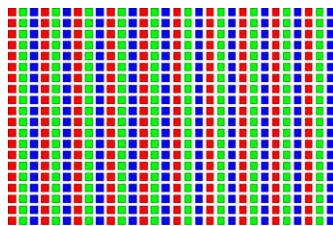


GPUs based system architecture

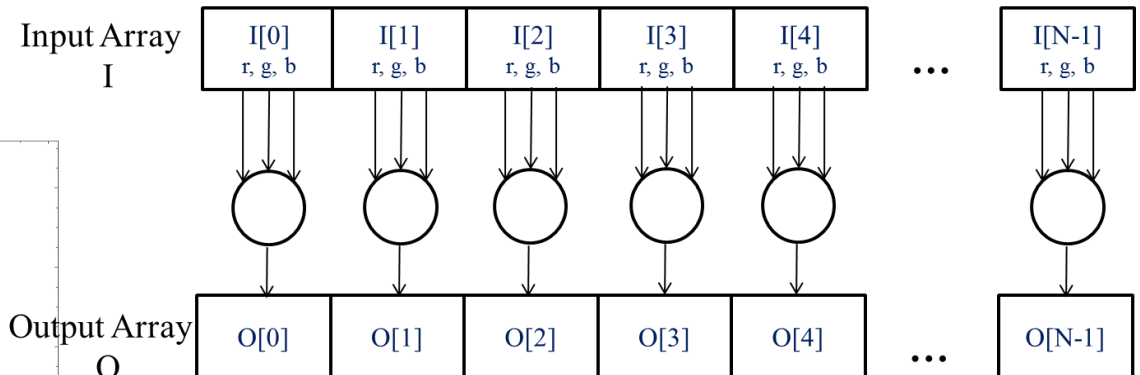


Architecture of a CUDA-capable GPU

Motivational example: Color image to grey-scale convert



$$L = r * 0.21 + g * 0.72 + b * 0.07$$



The pixels can be calculated independently of each other

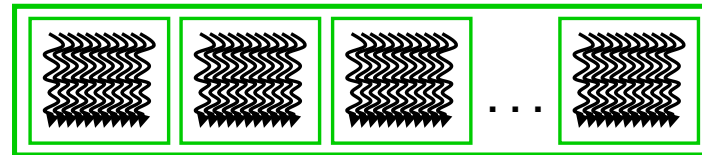
CUDA/OpenCL – Execution Model

- Integrated host+device app C program
 - Serial or modestly parallel parts in **host** C code
 - Highly parallel parts in **device** SPMD kernel C code

Serial Code (host)

Parallel Kernel (device)

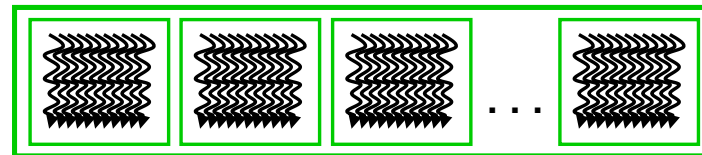
```
KernelA<<<nBlk, nTid>>>(args);
```



Serial Code (host)

Parallel Kernel (device)

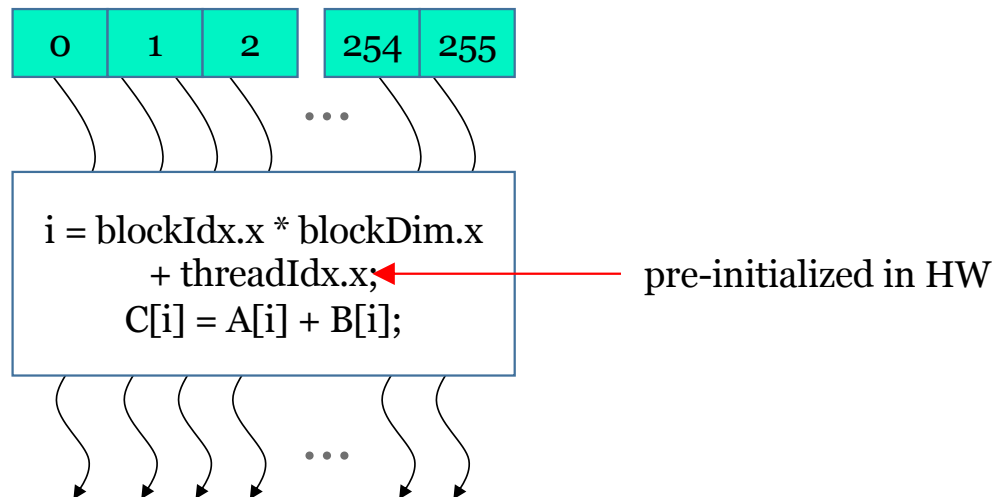
```
KernelA<<<nBlk, nTid>>>(args);
```



* SPMD – Single Program
Multiple Data

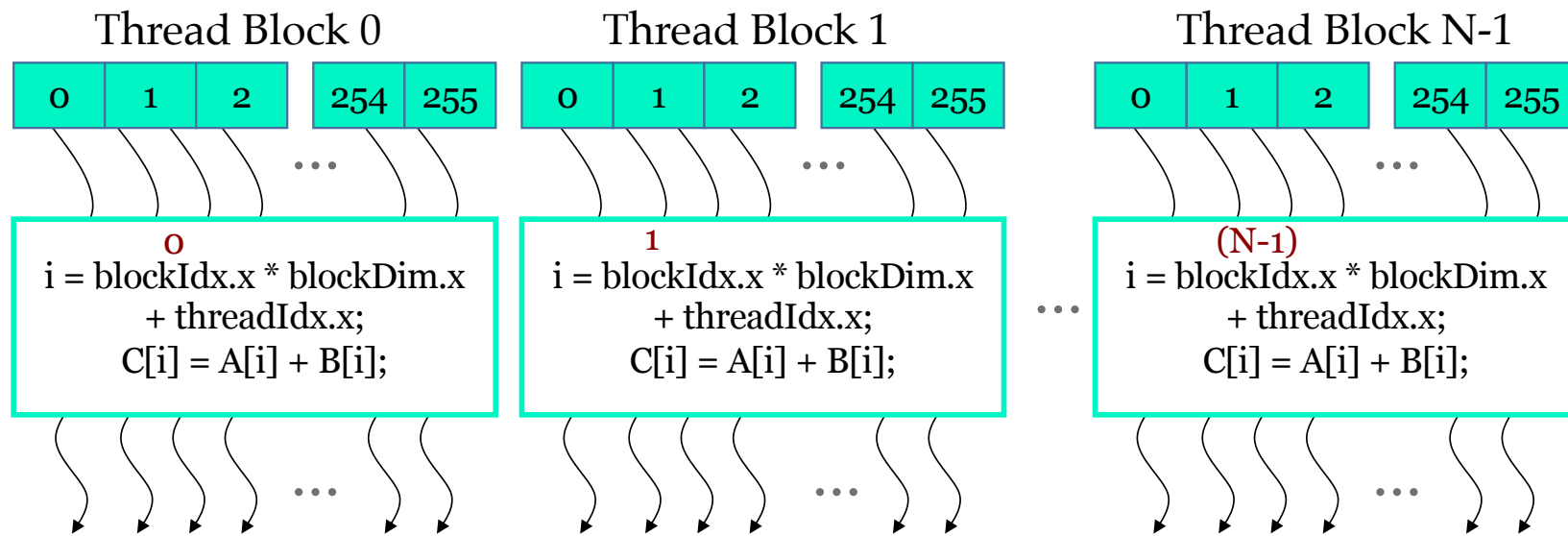
Arrays of Parallel Threads

- A CUDA kernel is executed by a **grid** (array) of threads
 - All threads in a grid run the same kernel code. Single Program Multiple Data (SPMD != SIMD)
 - Each thread has an index that it uses to compute memory addresses and make control decisions



Thread Blocks: Scalable Cooperation

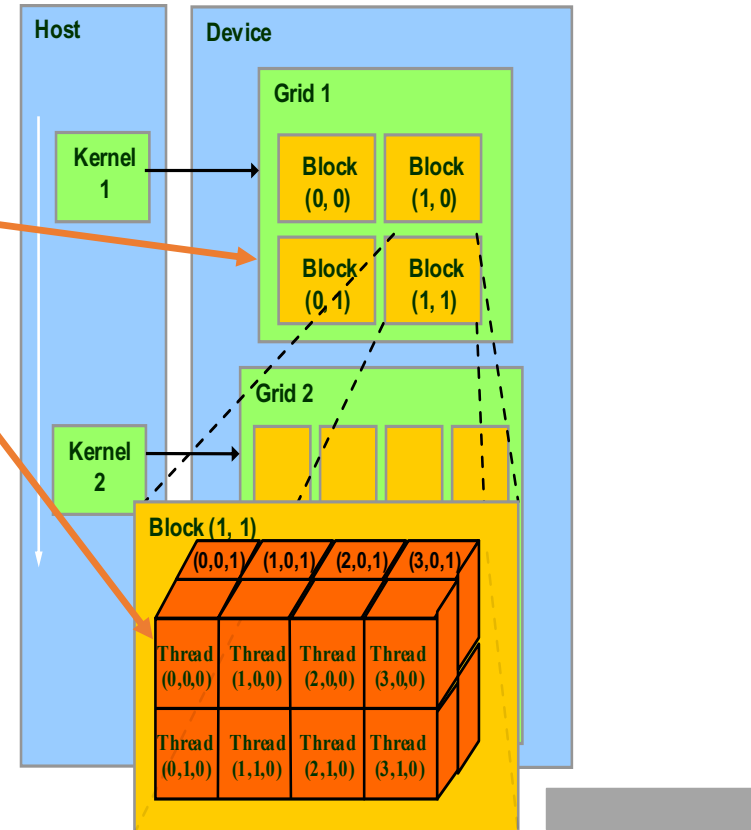
- Divide thread array into multiple **blocks**
 - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
 - Threads in different blocks cannot cooperate



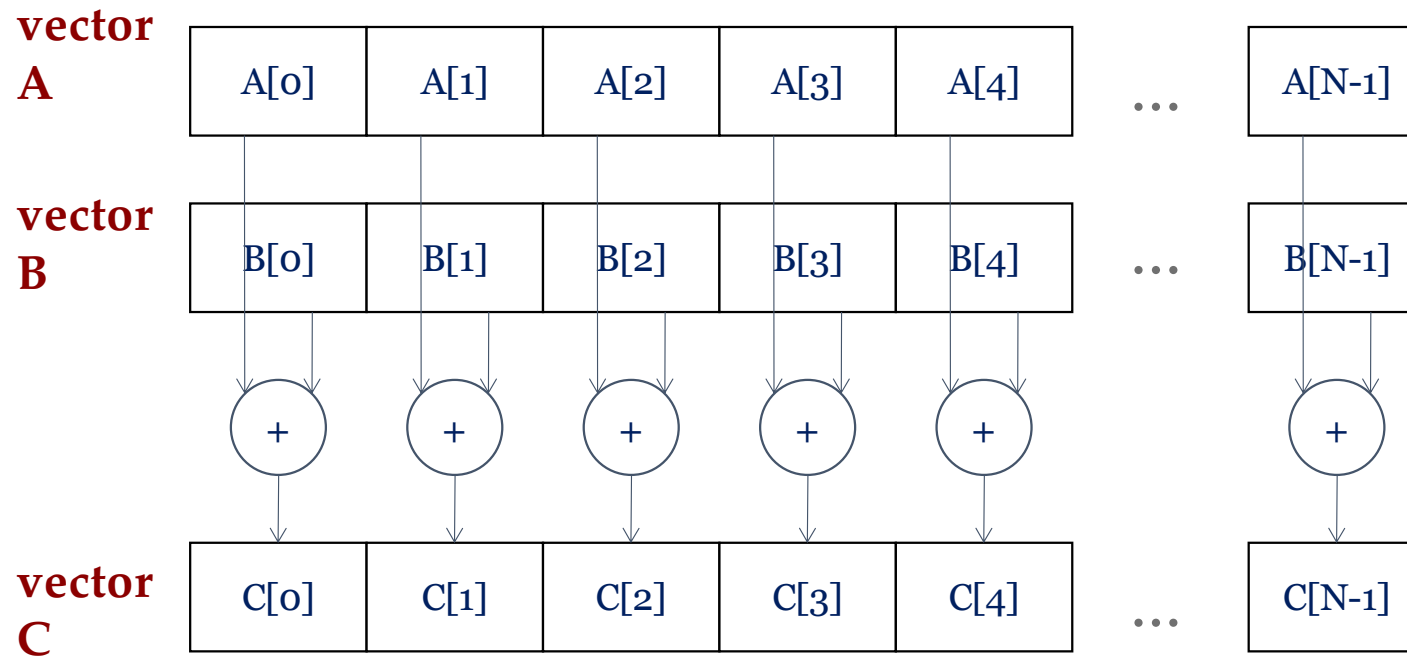
*There is a maximum number of threads in a thread block:
1,024 on CUDA 3.0 and up, 512 on earlier versions*

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
 - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
 - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
 - Image processing
 - Solving PDEs on volumes (*real 2D and 3D models*)
 - ...



Vector Addition – Conceptual View



Vector Addition – Traditional C Code

```
// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C, int n)
{
    for (i = 0, i < n, i++)
        C[i] = A[i] + B[i];
}

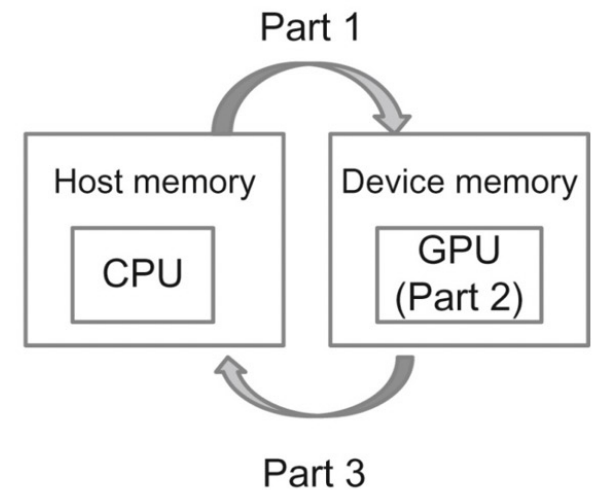
int main()
{
    // Memory allocation for A_h, B_h, and C_h
    // I/O to read A_h and B_h, N elements
    ...
    vecAdd(A_h, B_h, C_h, N);
}
```

Heterogeneous Computing vecAdd Host Code

```
#include <cuda.h>
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n* sizeof(float);
    float* A_d, B_d, C_d;
    ...
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code - to have the device
   // to perform the actual vector addition

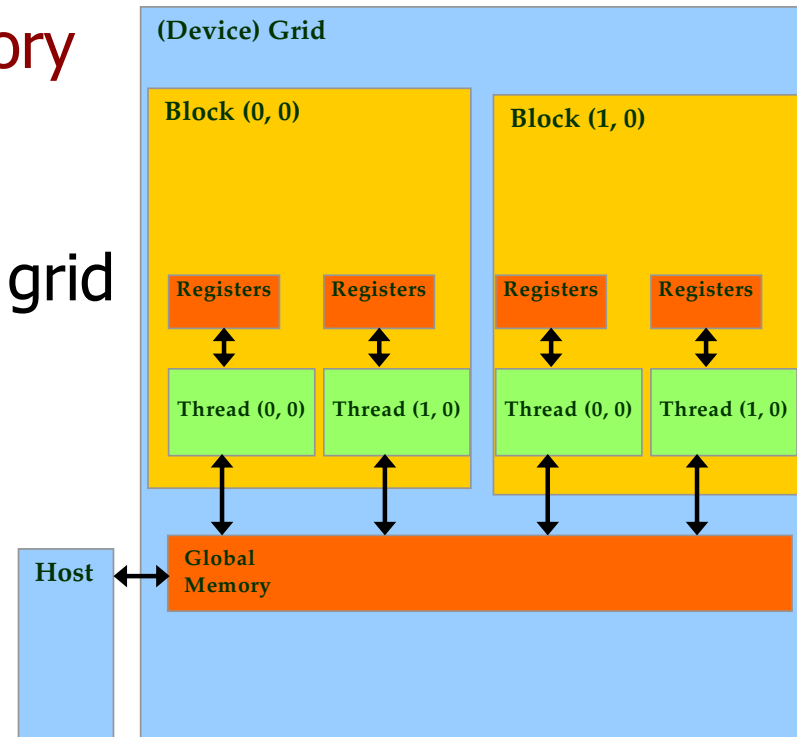
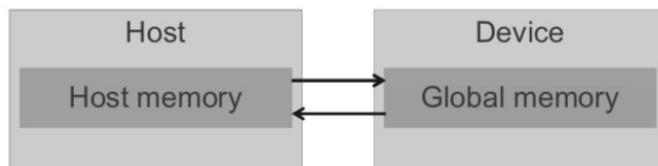
3. // copy C from the device memory
   // Free device vectors
}
```



Partial Overview of CUDA Memories

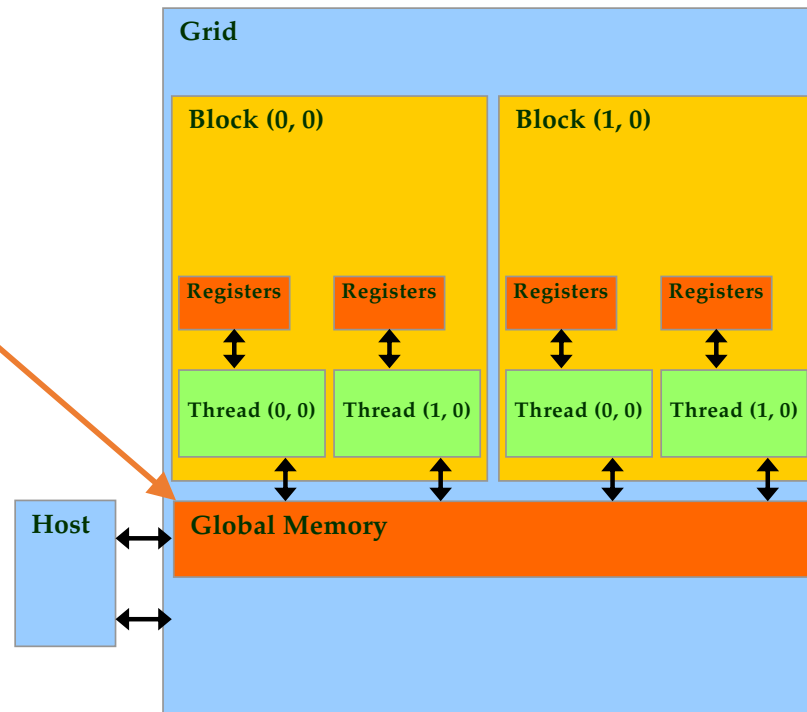
- Device code can:
 - R/W per-thread **registers**
 - R/W per-grid **global memory**
- Host code can
 - Transfer data to/from per grid **global memory**

more details later



CUDA Device Memory Management API functions

- `cudaMalloc()`
 - Allocates object in the **device global memory**
 - Two parameters
 - **Address of a pointer** to the allocated object
 - **Size of** allocated object in terms of bytes
- `cudaFree()`
 - Frees object from device global memory
 - **Pointer** to freed object

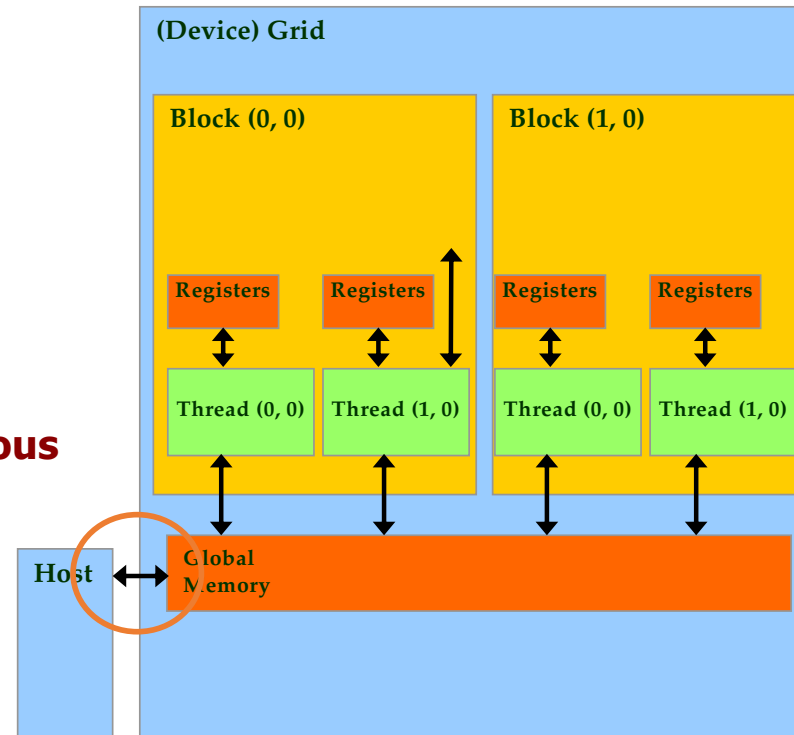


**cudaMalloc() returns a generic object making dynamic allocation more challenging, more later*

Host-Device Data Transfer API functions

- `cudaMemcpy()`
 - memory data transfer
 - Requires four parameters
 - Pointer to destination
 - Pointer to source
 - Number of bytes copied
 - Type/Direction of transfer
- Transfer to device is **asynchronous**

Immediate return
Overlapping opportunities



Some code ...

```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float* A_d, B_d, C_d;    “_d” stands for device (the latest book edition uses
                             d_<XYZ>)

1. // Transfer A and B to device memory
   cudaMalloc((void **) &A_d, size);
   cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
   cudaMalloc((void **) &B_d, size);
   cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

   // Allocate device memory for
   cudaMalloc((void **) &C_d, size);

2. // Kernel invocation code - to be shown later
   ...
3. // Transfer C from device to host
   cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
   // Free device memory for A, B, C
   cudaFree(A_d); cudaFree(B_d); cudaFree(C_d);
}
```

H --> D

D --> H

Dereferencing A_d, B_d and C_d from host is not advisable

In Practice, Check for API Errors in Host Code

```
cudaError_t err = cudaMalloc((void **) &d_A, size);

if (err != cudaSuccess) {
    printf("%s in %s at line %d\n",
        cudaGetErrorString(err), __FILE__, __LINE__);
    exit(EXIT_FAILURE);
}
```

All CUDA calls return error codes

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
```

(no return value)

```
__global__
```

```
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
```

```
{
```

```
    int i = threadIdx.x + blockDim.x * blockIdx.x;
```

```
    if(i<n) C_d[i] = A_d[i] + B_d[i];
```

```
}
```

```
int vectAdd(float* A, float* B, float* C, int n)
```

```
{
```

```
    // A_d, B_d, C_d allocations and copies omitted
```

```
    // Run ceil(n/256) blocks of 256 threads each
```

```
    vecAddKernel<<<ceil(n/256.0), 256>>>>(A_d, B_d, C_d, n);
```

```
}
```

There is a maximum number of threads in a thread block:
1,024 on CUDA 3.0 and 512 on earlier versions

use FP number to avoid truncation

Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddkernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vecAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0),256>>>>(A_d, B_d, C_d, n);
}
```

Host Code

More on Kernel Launch

```
int vecAdd(float* A, float* B, float* C, int n) Host Code
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    dim3 DimGrid(n/256, 1, 1);           avoid FP and ceil()
    if (n%256) DimGrid.x++;             alternative: (n-1)/256 + 1
    dim3 DimBlock(256, 1, 1);

    vecAddKernel<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);
}
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

Kernel execution in a nutshell

__host__
Void vecAdd()
{

dim3 DimGrid(ceil(n/256.0),1,1);
dim3 DimBlock(256,1,1);

vecAddKernel<<<DimGrid,DimBlock>>>
(A_d,B_d,C_d,n);
}

__global__

void vecAddKernel(float *A_d,
float *B_d, float *C_d, int n)

{
int i = blockIdx.x * blockDim.x
+ threadIdx.x;
if(i < n) C_d[i] = A_d[i] + B_d[i];
}



Schedule onto streaming multiprocessors

DimGrid and DimBlock
“live” in host and can be
called ... Maria and Mary



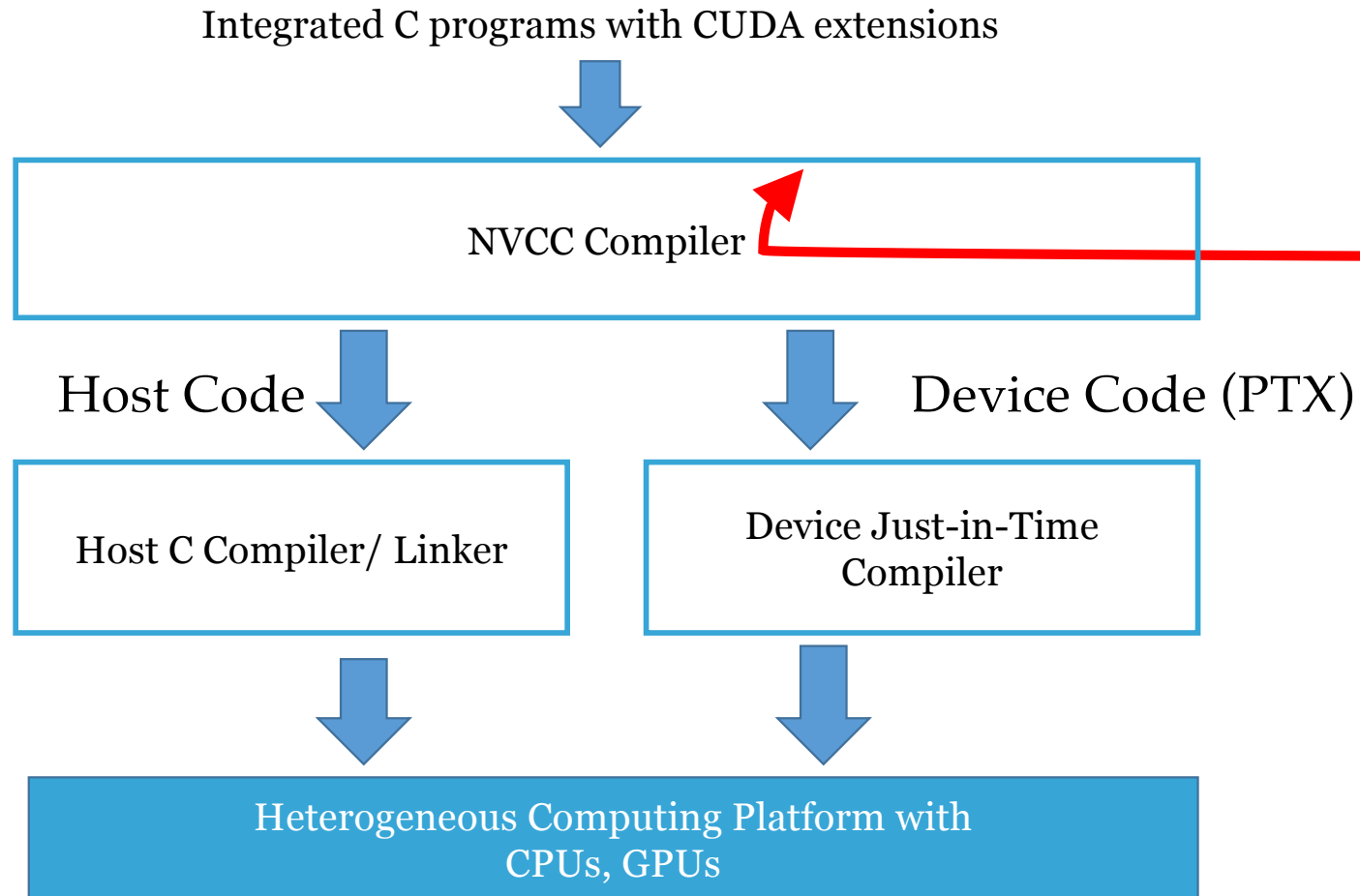
threadIdx, blockIdx,
blockDim and gridDim are
on the device and part of
the CUDA C specification

More on CUDA Function Declarations

	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

- `__global__` defines a kernel function
 - Each “__” consists of two underscore characters
 - A kernel function must return `void`
- `__device__` and `__host__` can be used together (e.g., user libraries)

Compiling a CUDA Program



Summary

- Parallel machines are here to stay
- We have to be able to build them but also model and program
- Fully automated parallelization compilers are still a dream
- Programming Massively Parallel accelerators requires tools

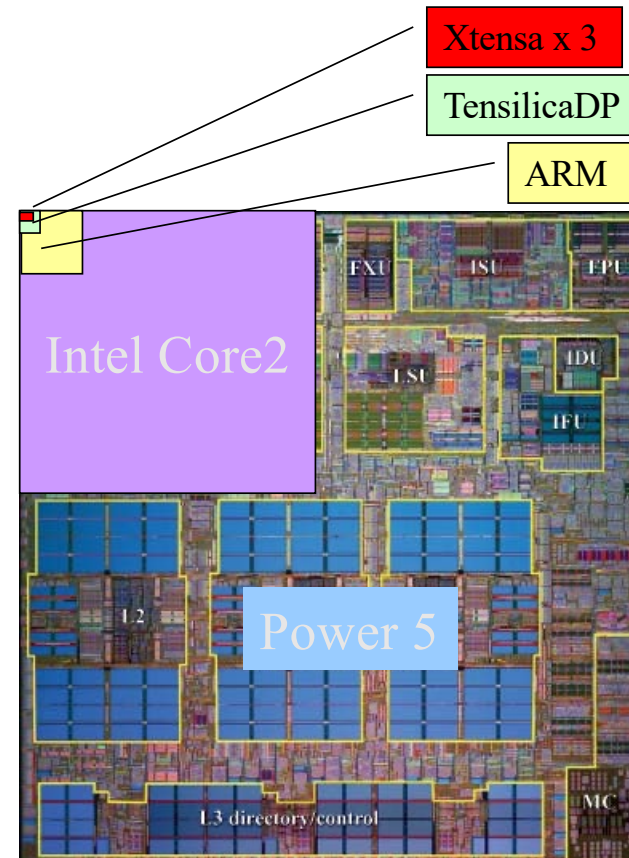
Thank you

Crash intro on Parallel Computing

Material from a BSc course on Parallel Computing

Size vs Power

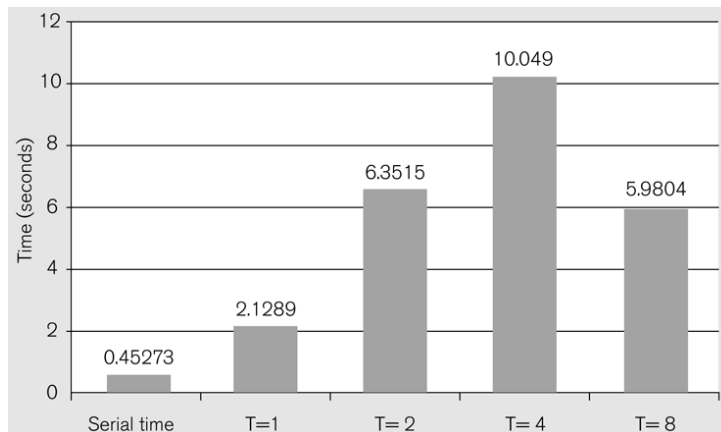
- Power5 (Server)
 - 389mm²
 - 120W@1900MHz
- Intel Core2 sc (laptop)
 - 130mm²
 - 15W@1,000MHz
- ARM Cortex A8 (automobiles)
 - 5mm²
 - 0.8W@800MHz
- Tensilica DP (cell phones / printers)
 - 0.8mm²
 - 0.09W@600MHz
- Tensilica Xtensa (Cisco router)
 - 0.32mm² for 3!
 - 0.05W@600MHz



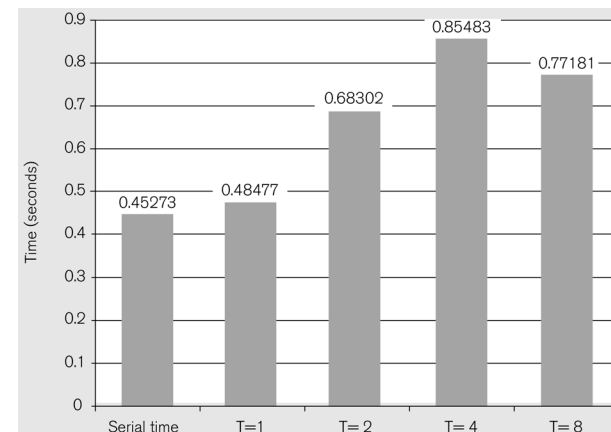
Each processor operates with 0.3-0.1 efficiency of the largest chip: more threads, lower power

Variations of Count 3s

- What happens when more processors are available?
 - 4 processors
 - 8 processors (look in the book)
 - 256 processors
 - 32,768 processors

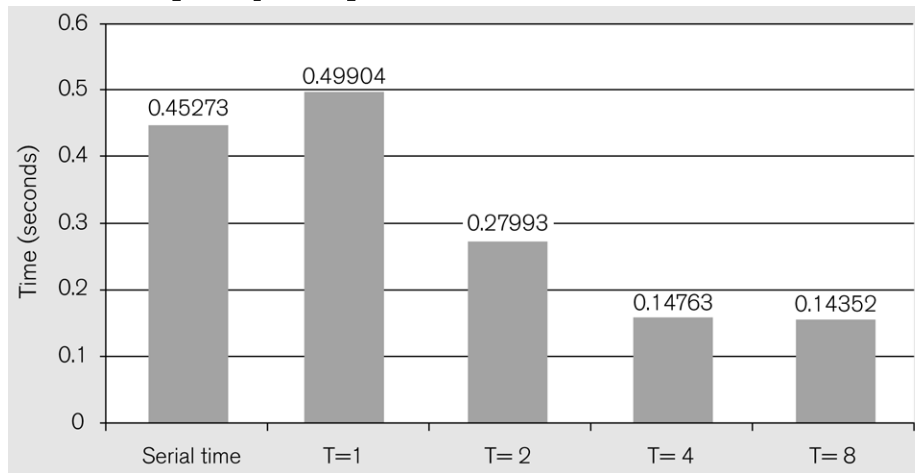


Try 2



Try 3

Variations (Try 4)

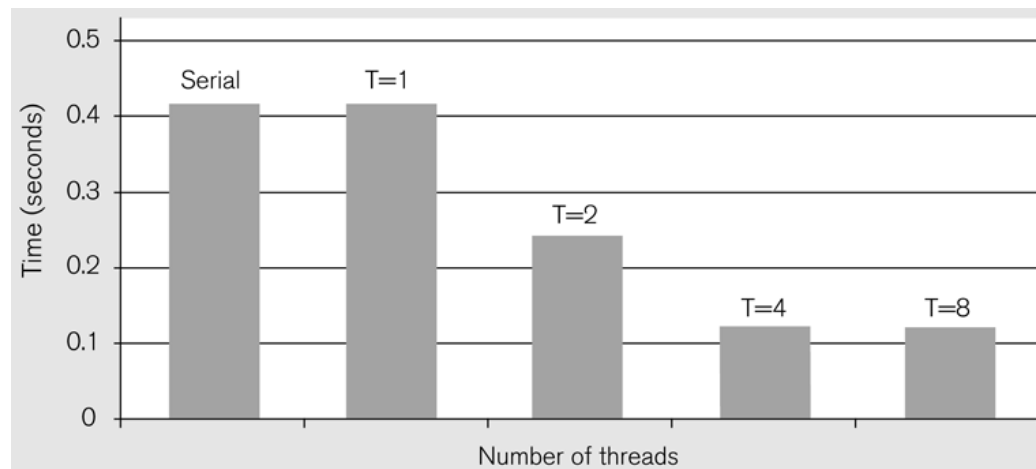


Experiments:

L3 (unified) 4MB 16-way 64B line size
L2 (unified) 1MB (per core) 8-way
L1 (I+D) 16KB/16KB 8-way SA
8 dual-core Xeon processors @ 2.6GHz

50MB random entry array with 30% 3s
Average of 1,000 program runs
GNU/Linux 2.6.19
Gcc 4.1.2 -O2 optimization on

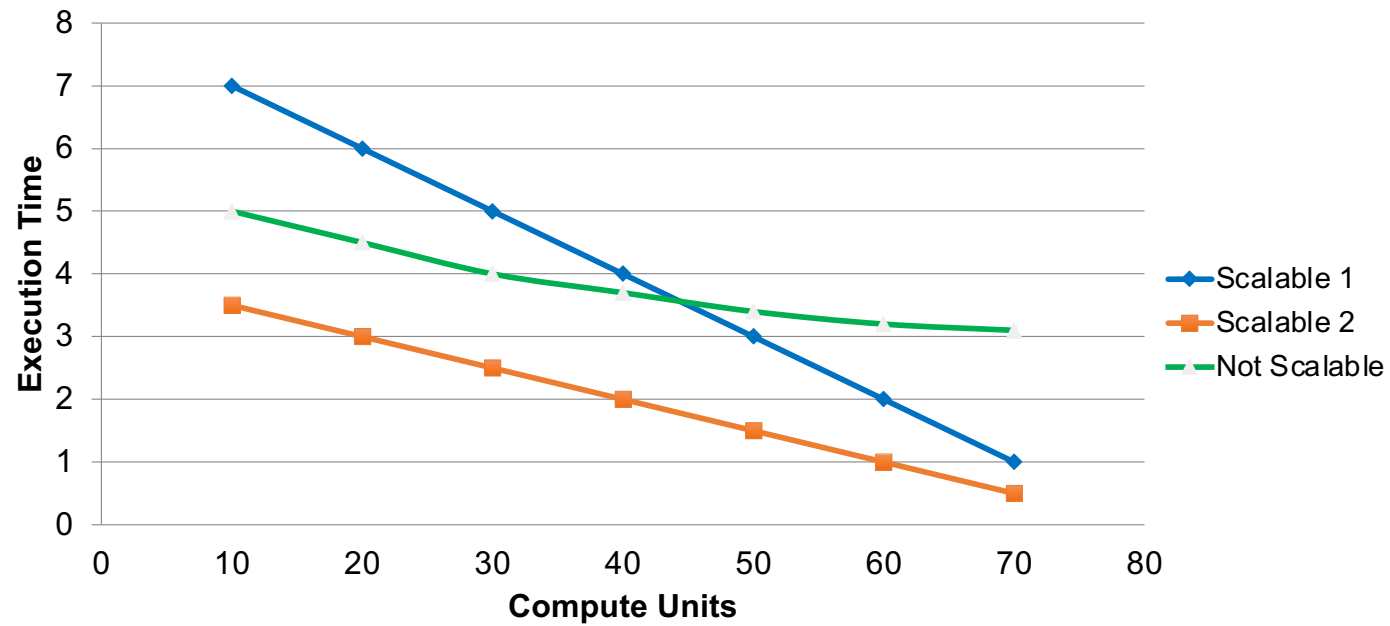
Performance for Try 4 solution on an array that **does not** contain any 3s suggests that memory bandwidth limitations are preventing performance gains for eight processors



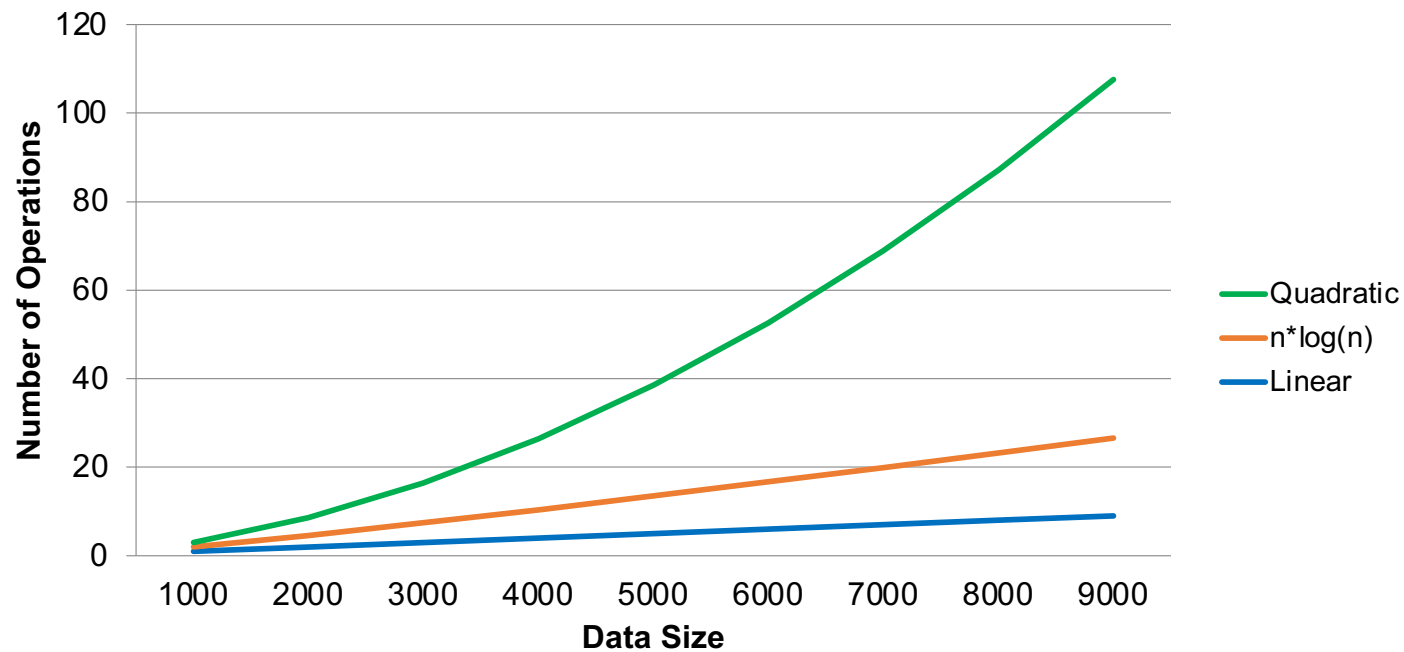
Our Goals In Parallel Programming

- **Goal:** Scalable programs with performance and portability
 - **Correct:** Obviously ...
 - **Performance:** Programs run as fast as those produced by experienced parallel programmers for the specific machine
 - **Scalable:** More processors can be “usefully” added to solve the problem faster
 - **Portability:** The solutions run well on all parallel platforms

Scalability of Parallelism



Algorithm Complexity and Data Scalability



© David Kirk/NVIDIA and Wen-mei W. Hwu, UIUC

What's The Deal With Hardware?

- Facts Concerning Hardware
 - Parallel computers differ dramatically from each other -- there is no standard architecture
 - No single programming target!
 - Parallelism introduces costs not present in vN machines -- communication; influence of external events
 - Many parallel architectures have failed
 - Details of parallel computer are ~~of~~ no greater concern to programmers than details of vN
should be

The “no single target” is key problem to solve

Our Plan

- Think about the problem abstractly
- Introduce instances of basic || designs
 - Multicore
 - Symmetric Multiprocessors (SMPs)
 - Large scale parallel machines
 - Clusters
 - Blue Gene/L
- Formulate a model of computation
- Assess the model of computation

Shared Memory

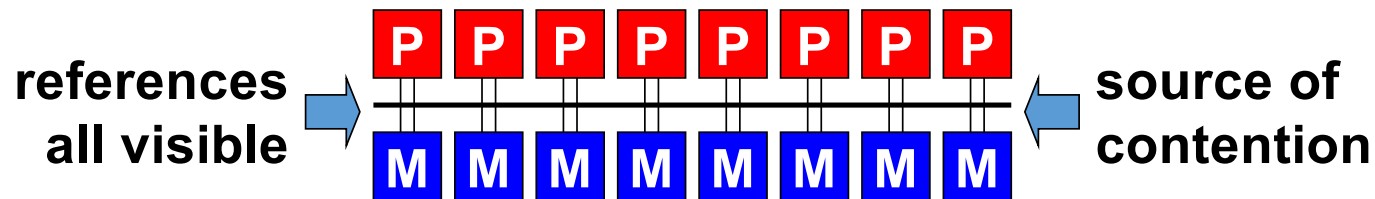
- Global memory shared among $||$ processors is the natural generalization of the sequential memory model
 - Thinking about it, programmers *assume* **sequential consistency** (SC) when they think about $||$ ism
- Recall Lamport's definition of SC:
 - "...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

Sequential Consistency

- SC difficult to achieve under all circumstances
- [*Whether SC suffices as a model at all is a deep and complex issue; there's more to say than today's points.*]
- The original way to achieve SC was literally to keep a single memory image and make sure that modifications are recorded in that memory

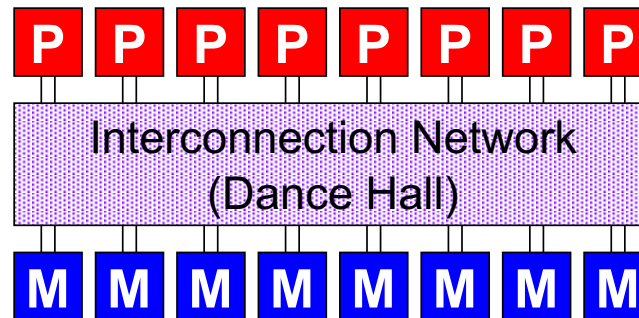
The Problem

- The “single memory” view implies ...
 - The memory is the **only source** of values
 - Processors use memory values **one-at-a-time**, not sharing or caching; if not available, stall
 - Lock when fetched, Execute, Store & unlock
- A bus can do this, but ...



Reduce Contention

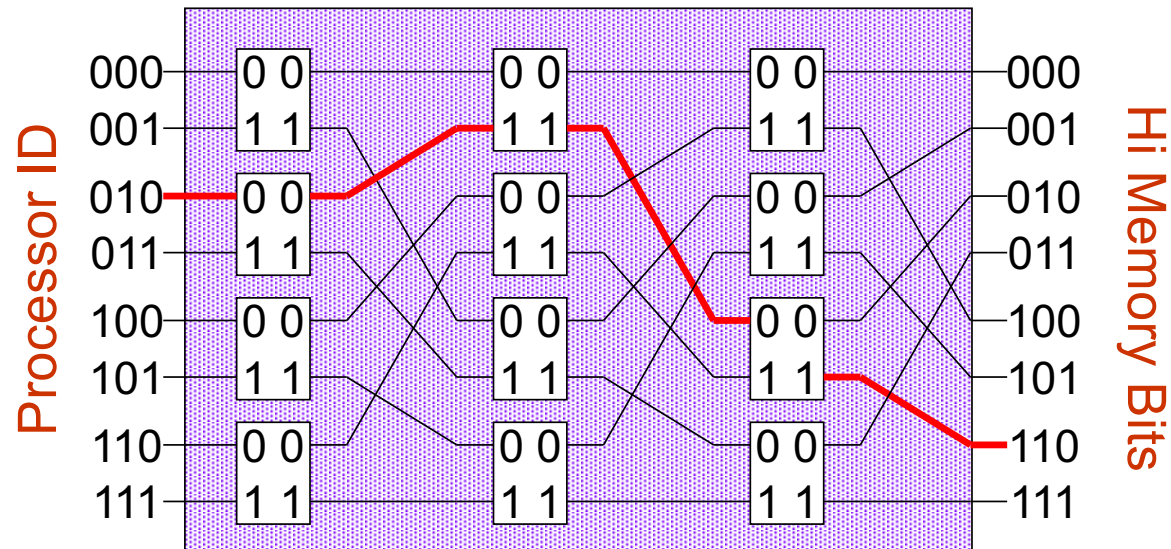
- Replace bus with network, an early design



- Network delays cause memory latency to be higher for a single reference than with a the bus, but simultaneous use should help when many references are in the air (MT)

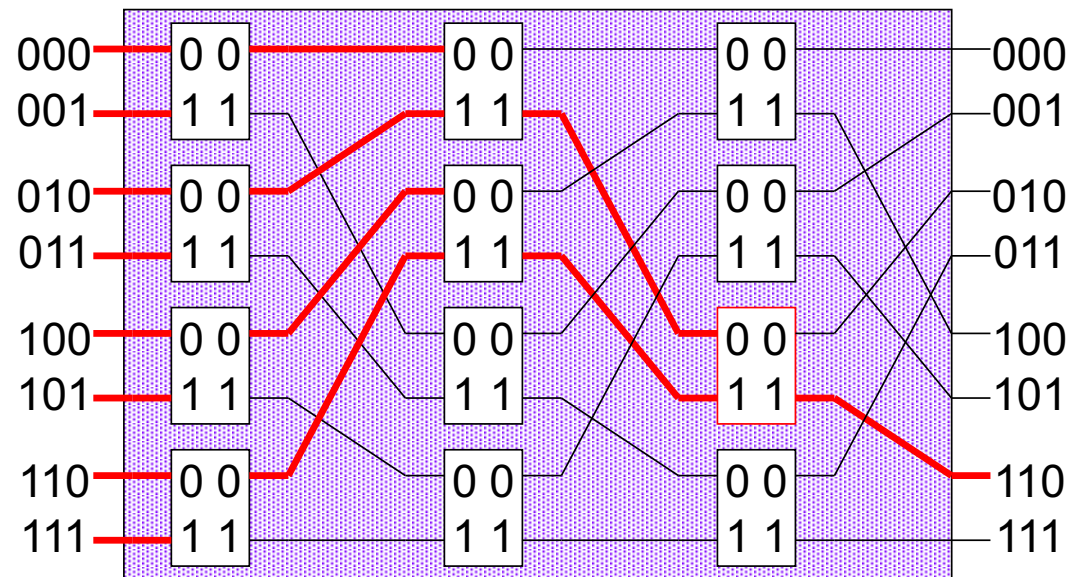
An Implementation

- Ω -Network is one possible interconnect
- Processor 2 references memory 6 (110) ←



Backing Up In Network

- Even if processors work on different data, the requests can back up in the network
- Everyone references data in memory 6



One-At-A-Time Use

- The critical problem is that only one processor at a time can use/change data
 - Cache read-only data (& programs) only
 - Check-in/Check-out model most appropriate
 - Conclusion: Processors stall a lot ...
- Solution: Multi-threading
 - When stalled, change to another waiting activity
 - Must make transition quickly, keeping context
 - Need ample supply of waiting activities
 - Available at different granularities

Briefly recap, Multithreading

- Multithreading: Executing multiple threads “at once”
- The threads are, of course, simply sequential programs executing a von Neumann model of computation
- Executed “at once” means that the context switching among them is not implemented by the OS, but takes place opportunistically in the hardware ... 3 related cases

Facts of Instruction Execution

- The von Neumann model requires that each instruction be executed to completion before starting the next
 - Once that was the way it worked
 - Now it is a conceptual model
- Multi-issue architectures start many instructions at a time, and do them when their operands are available leading to **out of order execution**

```
ld r1,0(r2)
add r1,r5
mult r8,r6
sw r1,0(r2)
li r1,0xabc
sw r1,4(r2)
```

Fine Grain Multithreading: Tera

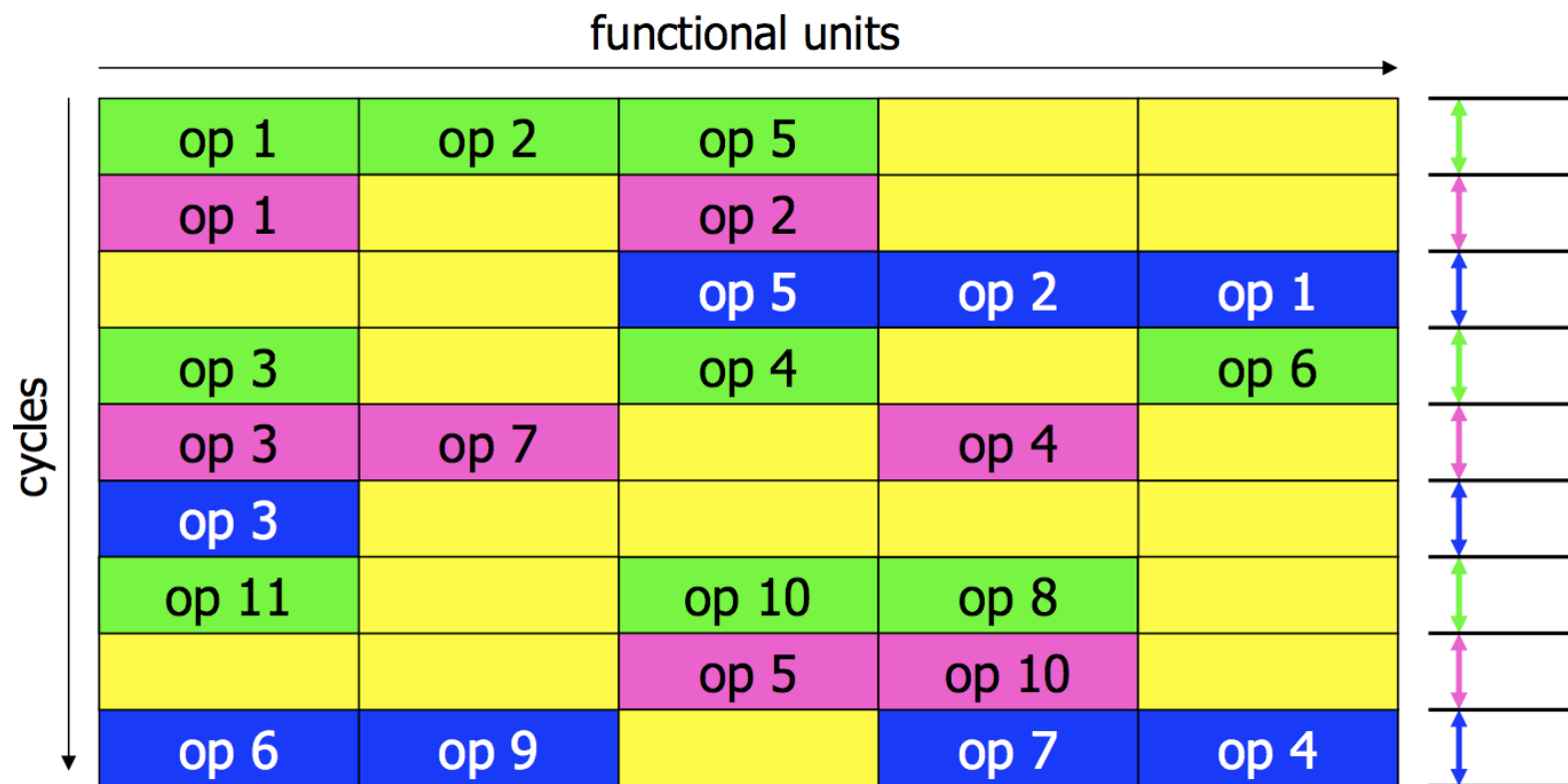
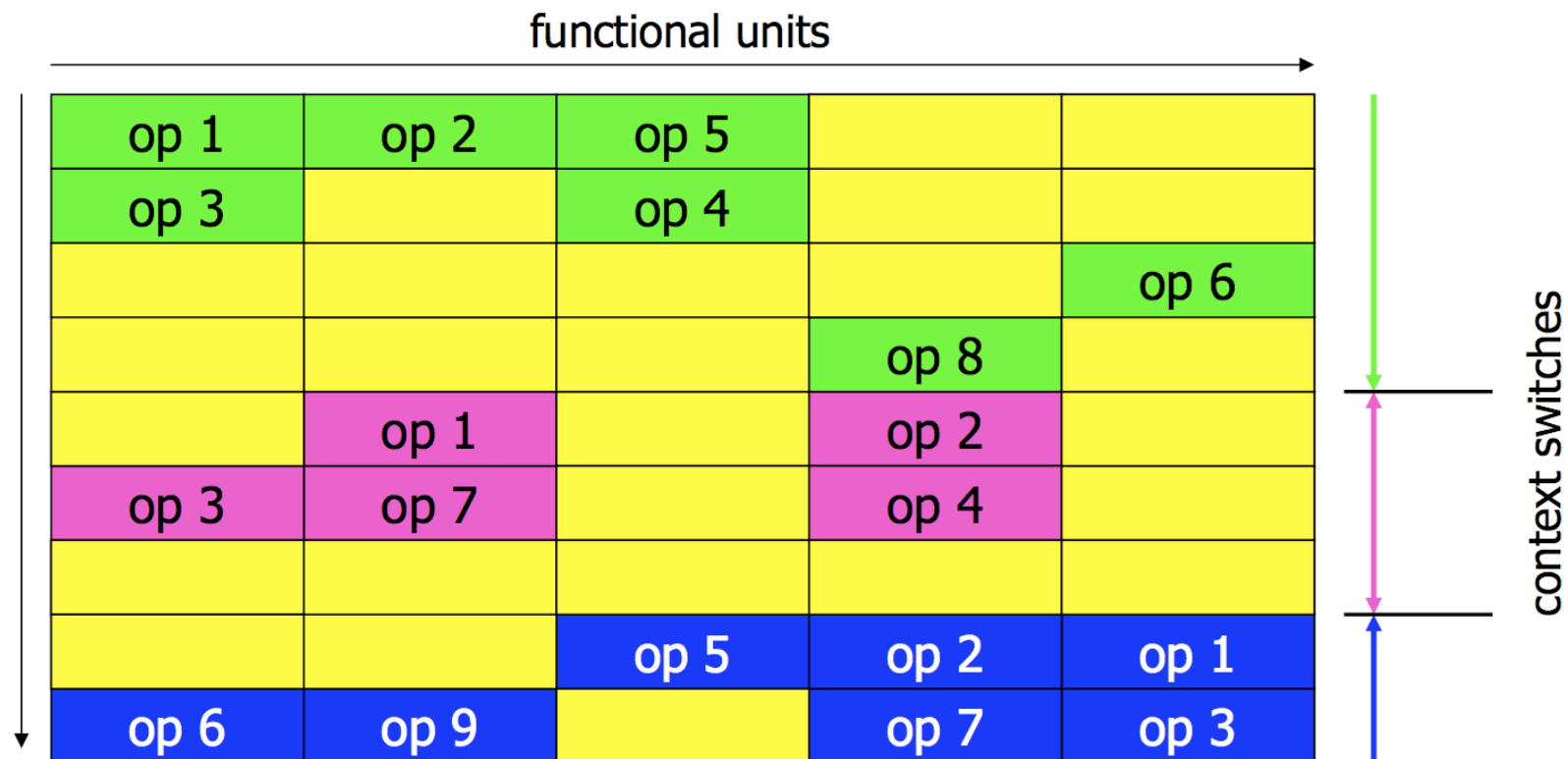
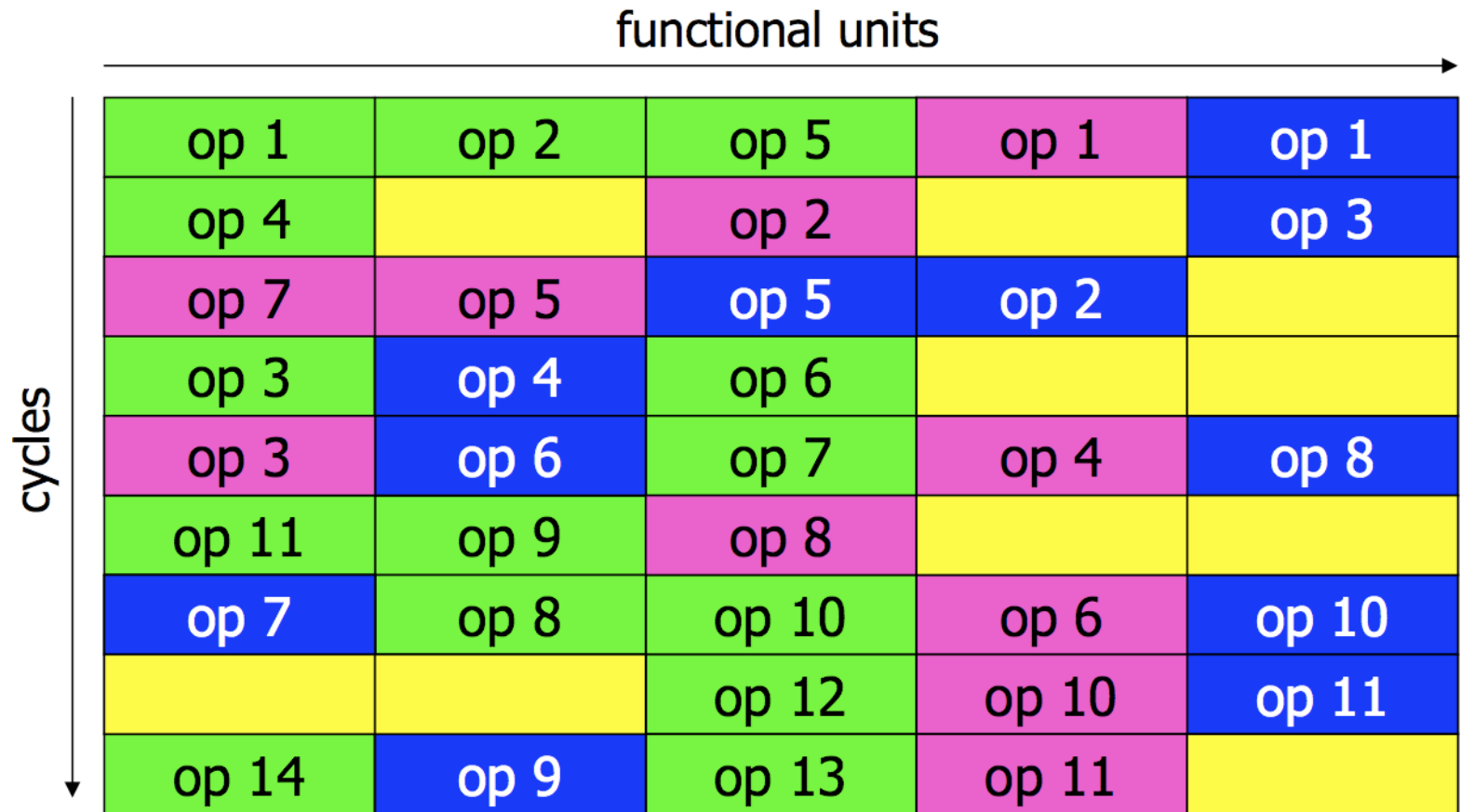


Figure from: Paolo.lenne@epfl.ch

Coarse Grain Multithreading: Alewife



Simultaneous Multi-threading: SMT



Multi-threading Grain Size

- The point when the activity switches can be
 - Instruction level, at memory reference: Tera MTA
 - Basic block level, with L1 cache miss: Alewife
 - ...
 - At process level, with page fault: Time sharing
- Another variation (3-address code level) is to execute many threads ($P^*/\log P$) in batches, called Bulk Synchronous Programming

No individual activity improved, but less wait time

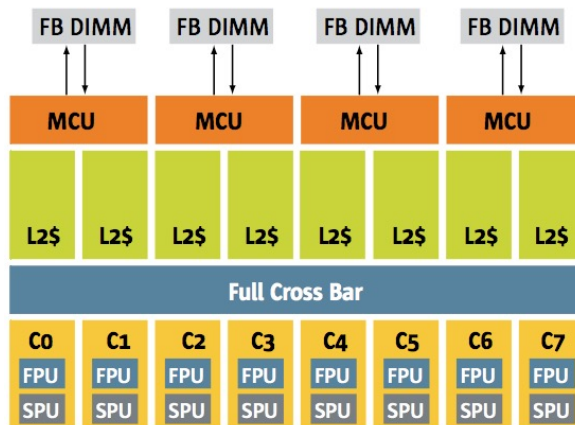
Problems with Multi-threading

- Cost (time, resources) of switching trades off with work: larger switching cost means more useful work completed before switch ... instruction level too low?
- Need many threads w/o dependences & ...
 - Threads must meet preceding criterion
 - Computations grow & shrink thread count (loop control) implies potential thread starvation
 - Fine-grain threads most numerous, but have least locality

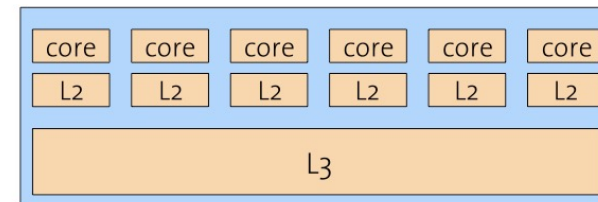
Multi-core Chips

- Multi-core means more than one processor per chip – generalization of SMT
- Consequence of Moore's Law
- IBM's PowerPC 2002, AMD Dual Core Opteron 2005, Intel CoreDuo 2006
 - 2022: Intel Core i9 (16 cores); AMD EPYC (64 cores)
- A small amount of multi-threading included
- Main advantage: More ops per tick
- Main disadvantages: Programming, BW

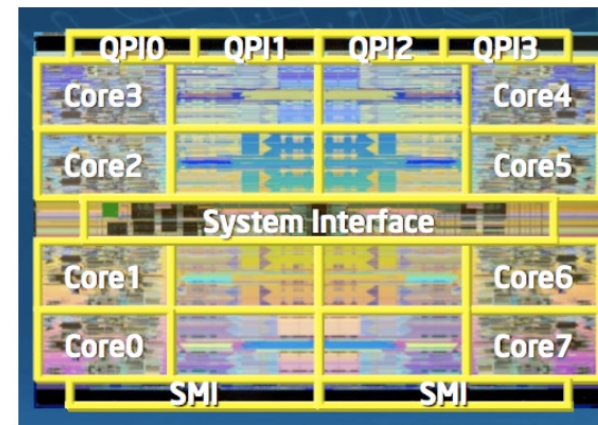
Diversity Among Small Systems



Sun Niagara T2



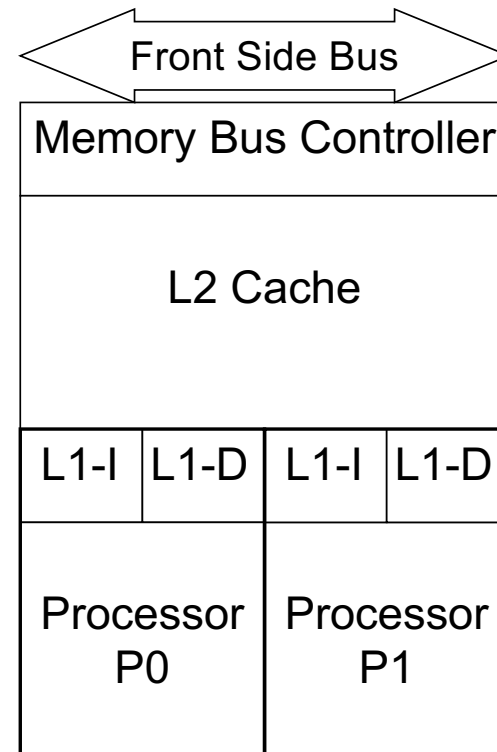
AMD Opteron (Istanbul)



Intel Nehalem (Beckton)

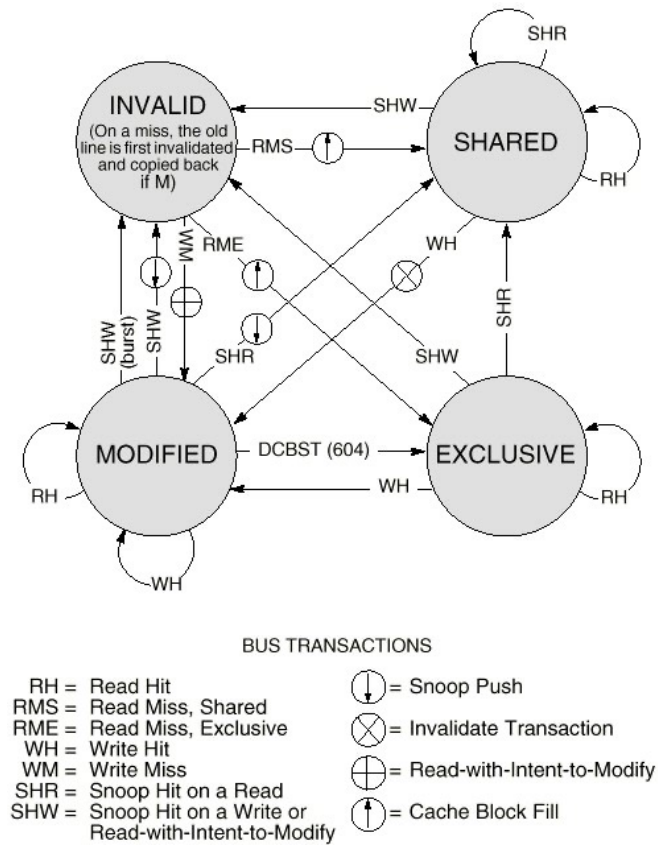
Intel CoreDuo

- Two 32-bit Pentiums
- Private 32K L1s
- Shared 2M-4M L2
- MESI cc-protocol
- Shared bus control and memory bus



MESI Protocol

- Standard Protocol for cache - coherent shared memory
 - Mechanism for multiple caches to give single memory image
 - We will not study it
 - 4 states can be amazingly rich



Thanks: Slater & Tibrewala of CMU

MESI, Intuitively

Modified
Exclusive
Shared
Invalid

- Upon loading, a line is marked E, subsequent reads are OK; write marks M
- Seeing another load, mark as S
- A write to an S, sends I to all, marks as M
- Another's read to an M line, writes it back, marks it S
- Read/write to an I misses
- Related scheme: MOESI (used by AMD)

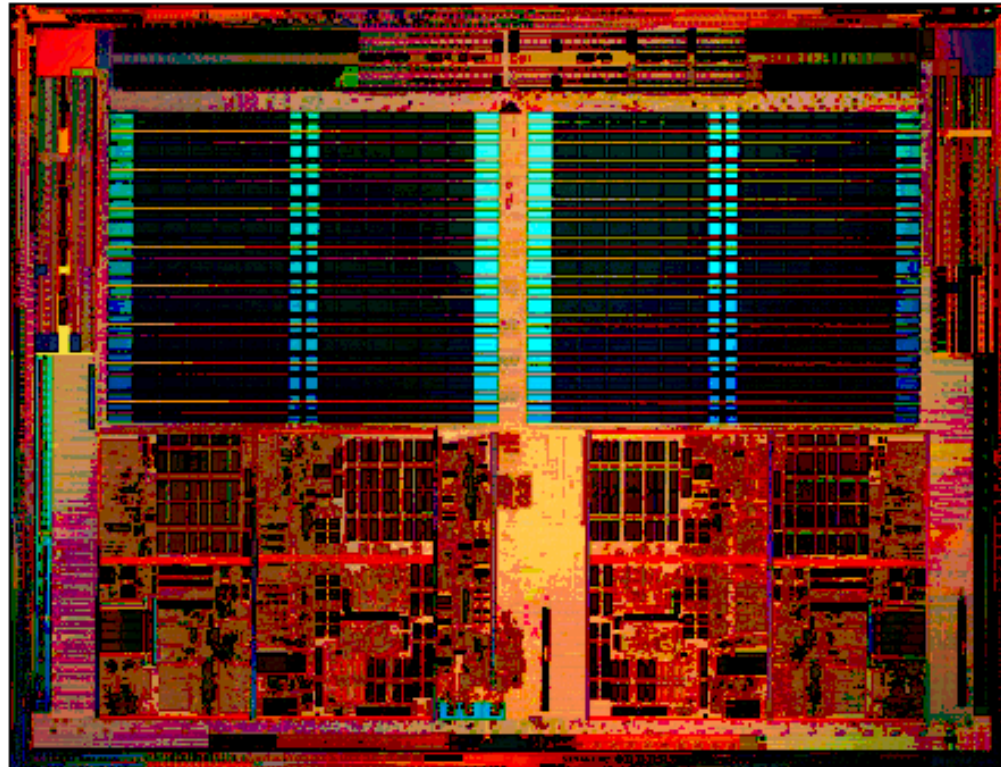
	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

The only valid combinations of states for the same cache line

Owned state supports cache line updates without access to Main memory (interconnect is more complex)

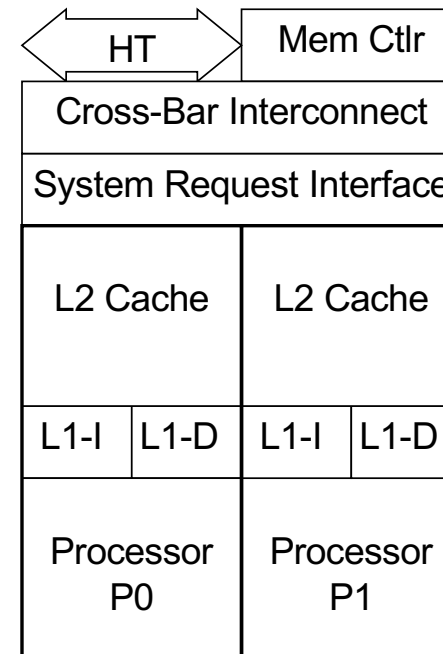
	M	O	E	S	I
M	✗	✗	✗	✗	✓
O	✗	✗	✗	✓	✓
E	✗	✗	✗	✗	✓
S	✗	✓	✗	✓	✓
I	✓	✓	✓	✓	✓

AMD Dual Core Opteron

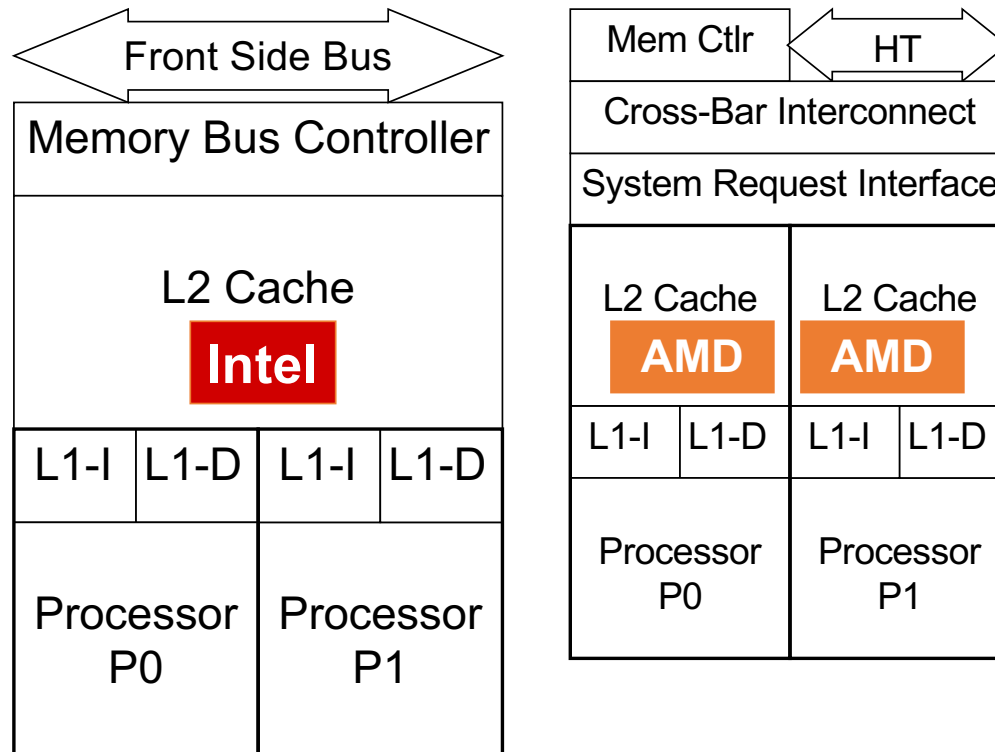


AMD Dual Core Opteron

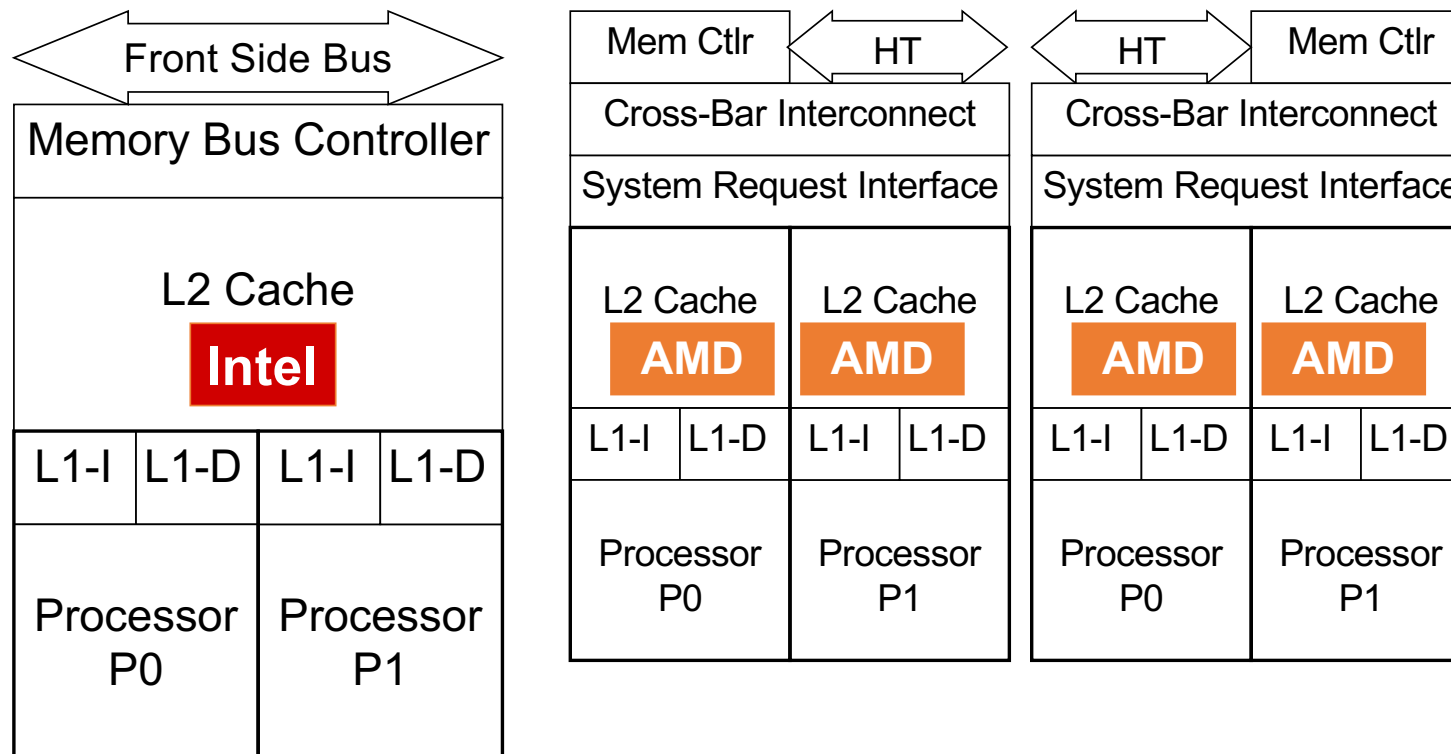
- Two 64-bit Opterons
- 64K private L1s
- 1 MB private L2s
- MOESI cc-protocol
- Direct connect shared memory



Comparing Core Duo/Dual Core

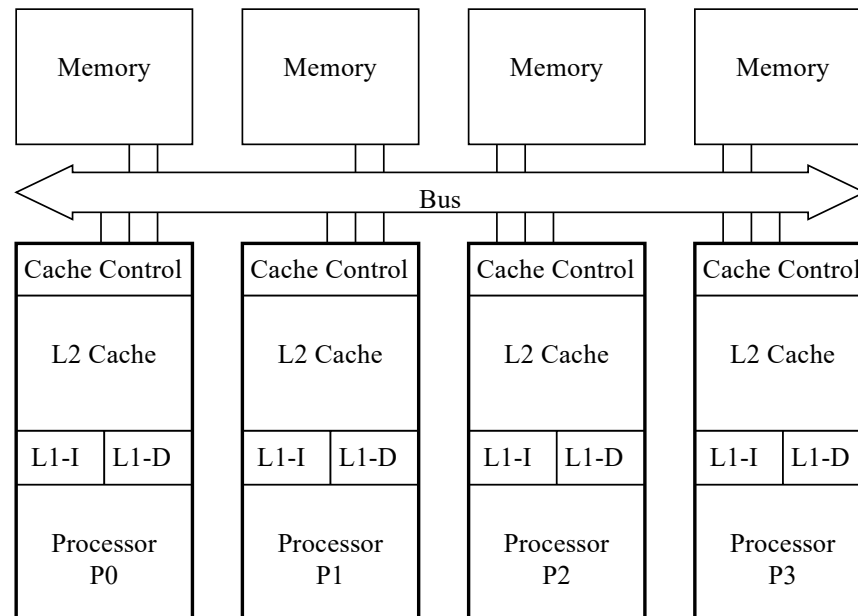


Comparing Core Duo/Dual Core

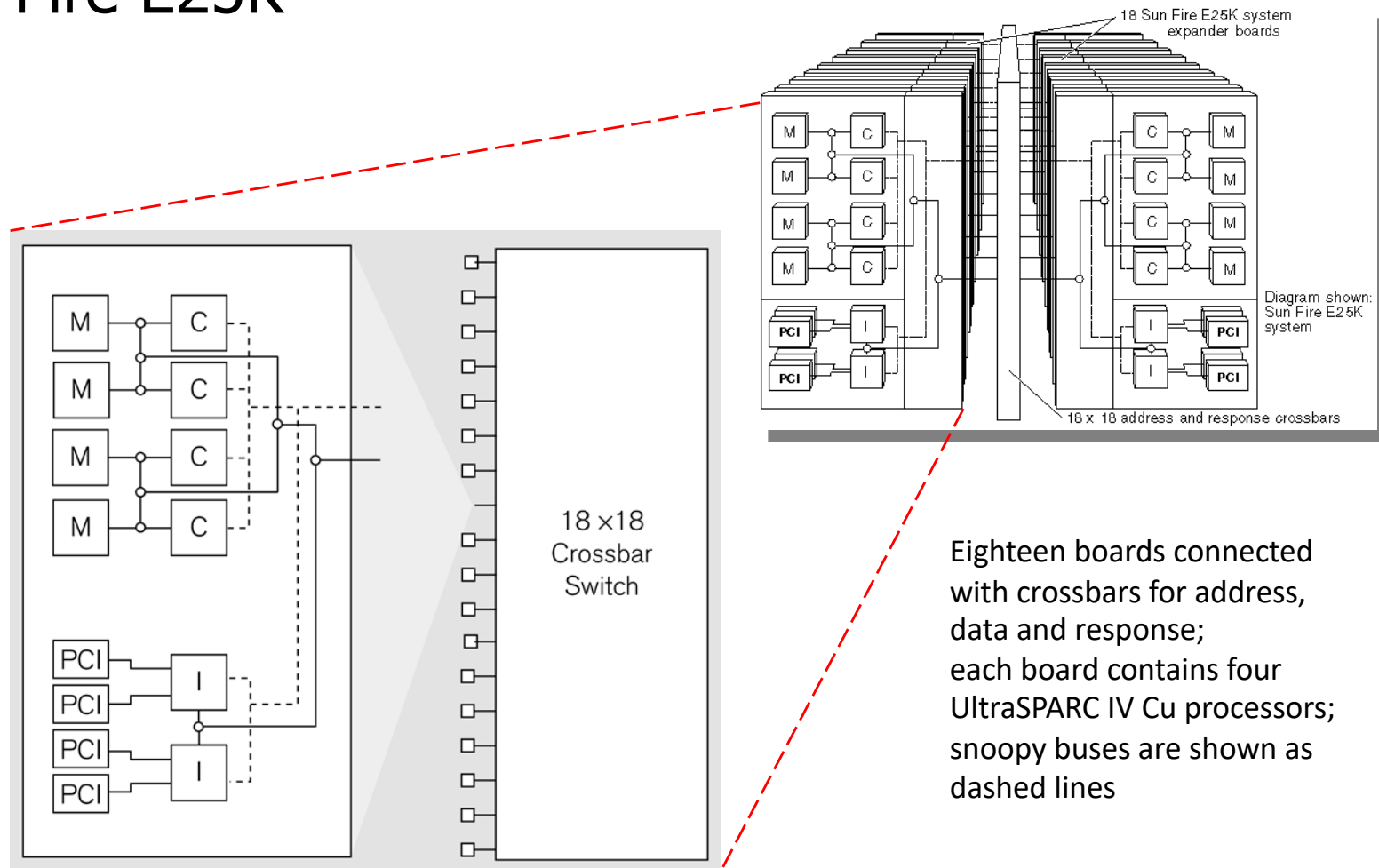


Symmetric Multiprocessor on a Bus

- The bus is a point that serializes references
- A serializing point is a shared mem enabler

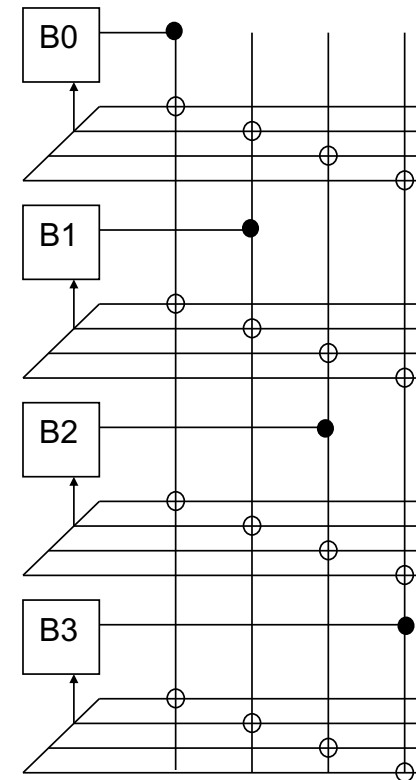


Sun Fire E25K



Cross-Bar Switch

- A crossbar is a network connecting each processor to every other processor
- Used in CMU's 1971 C.MMP, 16 proc PDP-11s
- Crossbars grow as n^2 making them impractical for large n



Sun Fire E25K

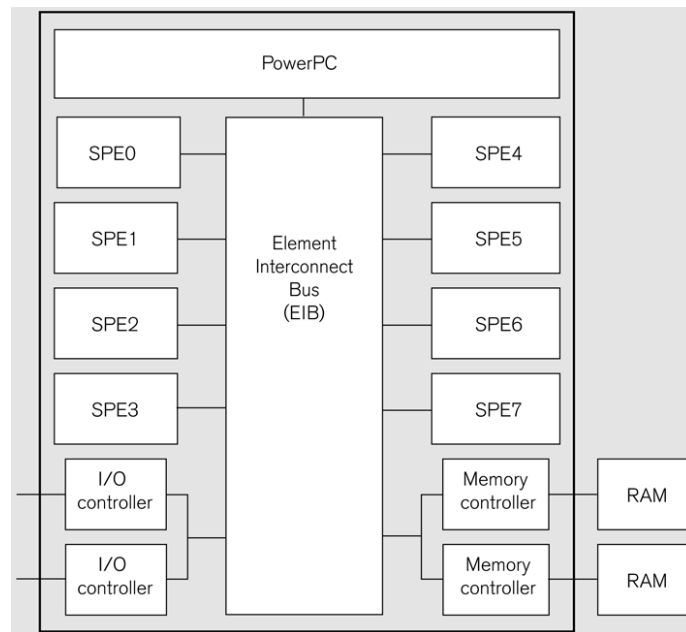
- X-bar gives low latency for snoops allowing for shared memory
- 18 x 18 X-bar is basically the limit
- Raising the number of processors per node will, on average, increase congestion
- How could we make a larger machine?

Co-Processor Architectures

- A powerful parallel design is to add one or more subordinate processors to standard design
 - Floating point instructions once implemented this way
 - Graphics Processing Units – massive #thr, deep pipelining
 - Cell Processor - multiple SIMD units
 - Attached FPGA chip(s) - compile to a circuit
 - TPUs – tensor processing units (custom)
- Some of these architectures will be discussed later

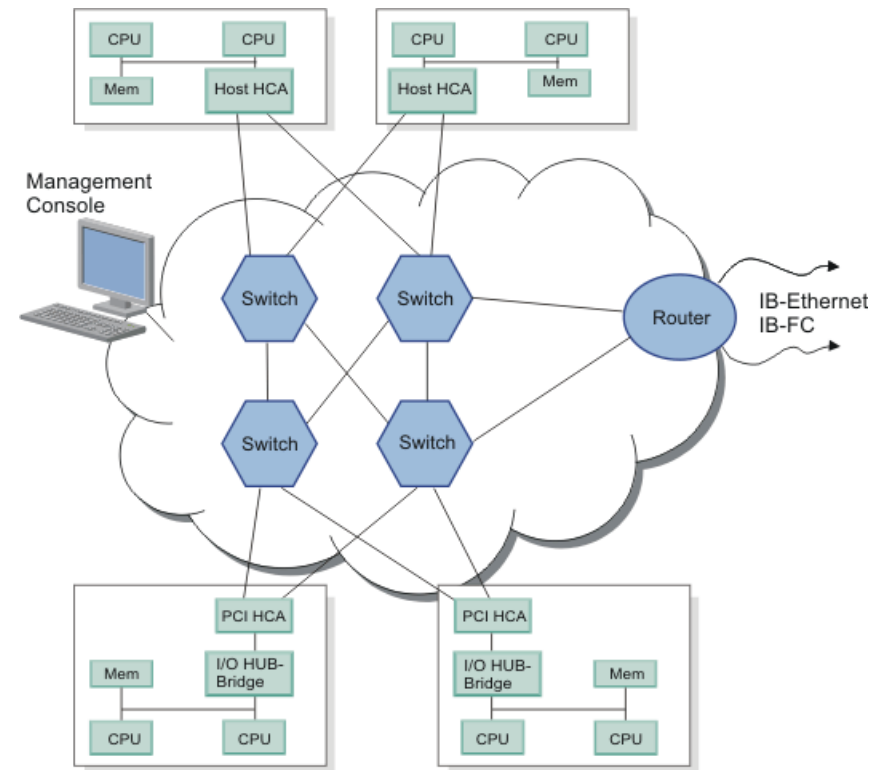
The Cell Processor

- Architecture designed to move data
 - high speed I/O controllers with 76.8 GB/s
 - two channels to RAM of 12.8 GB/s
 - EIB is theoretically capable of 204.8 GB/s.



Clusters

- Interconnecting with InfiniBand
- Switch-based technology
 - Host channel adapters (HCA)
 - Peripheral computer interconnect (PCI)



IPHAES05-1

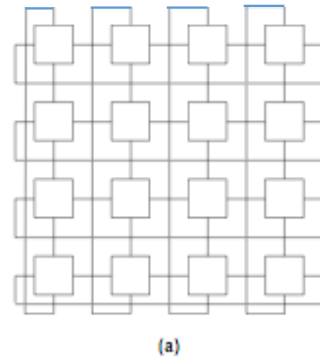
Thanks: IBM's Clustering systems using InfiniBand Hardware

Clusters

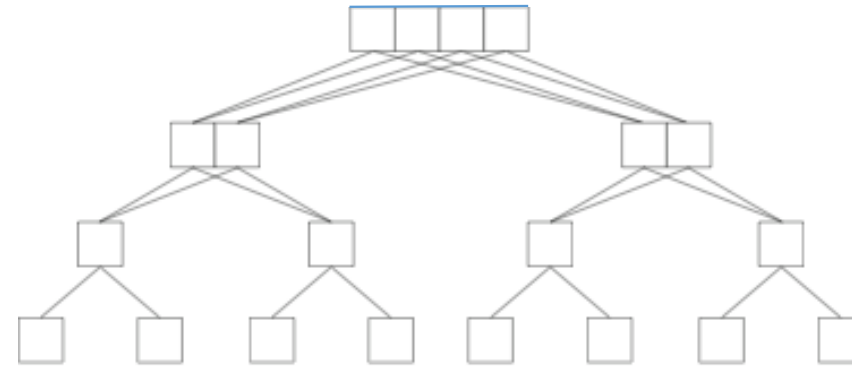
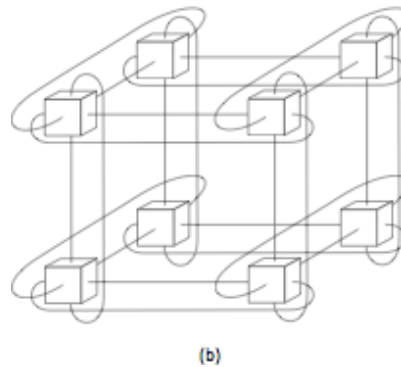
- Cheap to build using commodity technologies
- Effective when interconnect is “switched”
- Easy to extend, usually in increments of one
- Processors often have disks “nearby”
- No shared memory
- Latencies are usually large
- Programming uses message passing (tbd later)

Networks

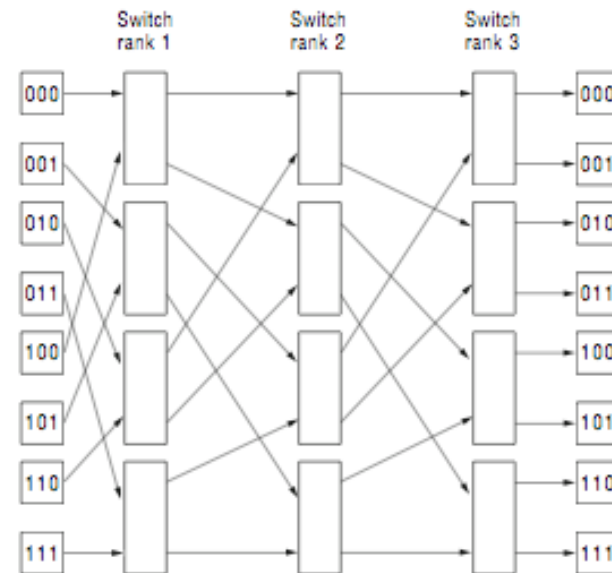
**Torus
(Mesh)**



**Hyper-
Cube**



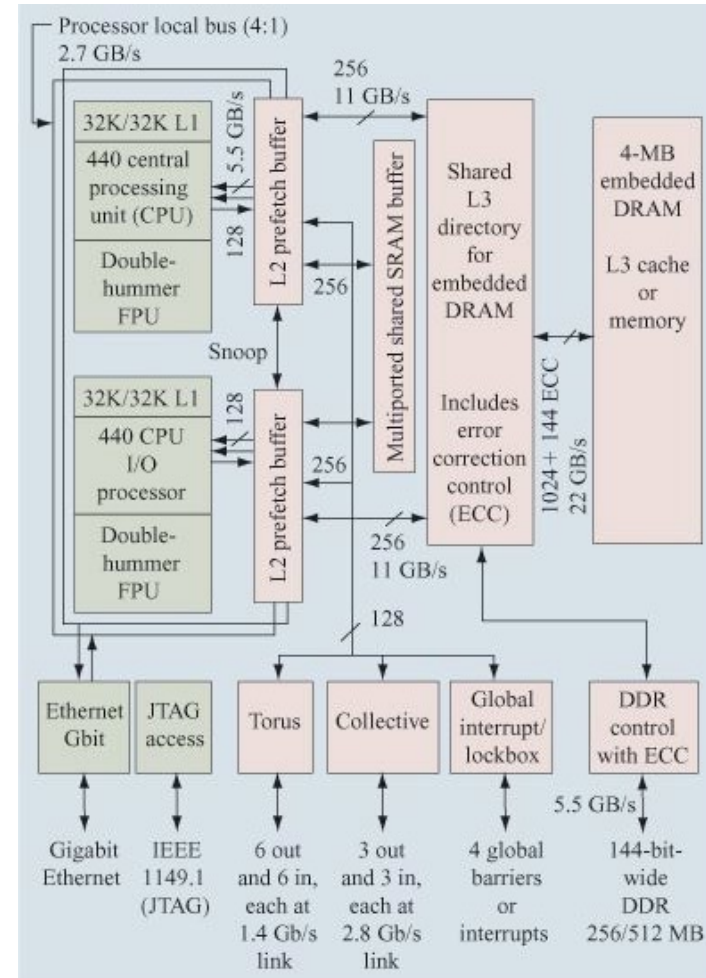
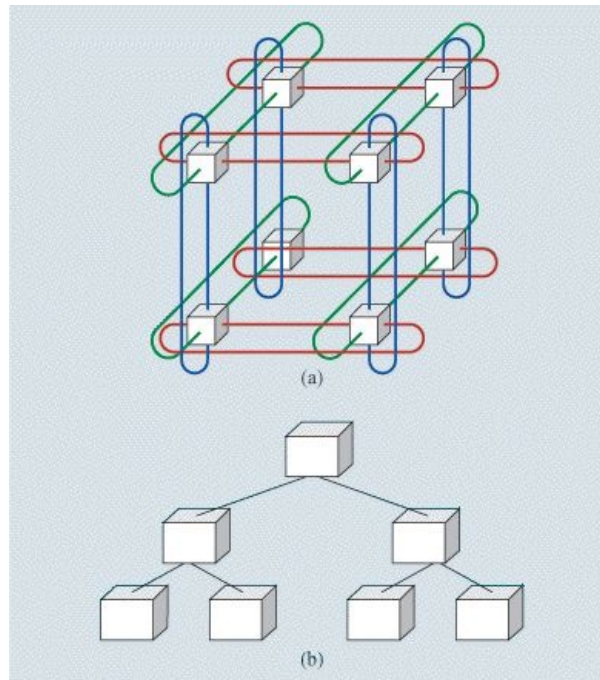
Fat Tree



Omega Network

Supercomputer

- BlueGene/L



BlueGene/L Specs

- A 64x32x32 torus = 65K 2-core processors
- Cut-through routing gives a worst-case latency of 6.4 μ s
- Processor nodes are dual PPC-440 with “double hummer” FPUs
- Collective network performs global reduce for the “usual” functions

Summarizing Architectures

- Two main classes
 - Complete connection: CMPs, SMPs, X-bar
 - Preserve single memory image
 - Complete connection limits scaling to ...
 - Available to everyone
 - Sparse connection: Clusters, Supercomputers, Networked computers used for parallelism (Grid)
 - Separate memory images
 - Can grow “arbitrarily” large
 - Available to everyone with air conditioning
- Differences are significant; world views diverge

The Parallel Programming Problem

- Some computations can be platform specific
- Most should be platform independent
- Parallel Software Development Problem: How do we neutralize the machine differences given that
 - Some knowledge of execution behavior is needed to write programs that perform
 - Programs must port across platforms effortlessly, meaning, by at most recompilation

Options for Solving the PPP

- Leave the problem to the compiler ...

Options for Solving the PPP

- Leave the problem to the compiler ...
 - Very low level parallelism (ILP) is already being exploited
 - Sequential languages cause us to introduce unintentional sequentiality
 - Parallel solutions often require a paradigm shift
 - Compiler writers' track record over past four decades not promising ... recall High Performance Fortran (HPF) 1995
 - Bottom Line: Compilers will get more helpful, but they probably won't solve the PPP (or P^3)

Options for Solving the PPP

- Adopt a very abstract language that can target to any platform ...

Options for Solving the PPP

- Adopt a very abstract language that can target to any platform ...
 - No one wants to learn a new language, no matter how cool
 - How does a programmer know how efficient or effective his/her code is? Interpreted code?
 - What are the “right” abstractions and statement forms for such a language?
 - Emphasize programmer convenience?
 - Emphasize compiler translation effectiveness?

Options for Solving the PPP

- Agree on a set of parallel primitives (spawn process, lock location, etc.) and create libraries that work w/ sequential code ...

Options for Solving the PPP

- Agree on a set of parallel primitives (spawn process, lock location, etc.) and create libraries that work w/ sequential code ...
 - Libraries are a mature technology
 - To work with multiple languages, limit base language assumptions ... L.C.D. facilities
 - Libraries use a stylized interface (function call) limiting possible parallelism-specific abstractions
 - Achieving consistent semantics is difficult

Options for Solving the PPP

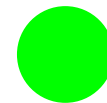
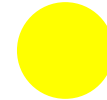
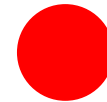
- Create an abstract machine model that accurately describes common capabilities and let the language facilities catch up ...

Options for Solving the PPP

- Create an abstract machine model that accurately describes common capabilities and let the language facilities catch up ...
 - Not a full solution until languages are available
 - The solution works in sequential world (RAM)
 - Requires discovering (and predicting) what the common capabilities are
 - Solution needs to be (continually) validated against actual experience

Summary of Options for PPP

- Leave the problem to the compiler ...
- Adopt a very abstract language that can target to any platform ...
- Agree on a set of parallel primitives (spawn process, lock location, etc.) and create libraries that work w/ sequential code ...
- Create an abstract machine model that accurately describes common capabilities and let the language facilities catch up ...



Why is Sequential Programming Successful

When we write programs in C they are ...

- **Efficient** -- programs run fast, especially if we use performance as a goal
 - traverse arrays in row major order to improve caching
- **Economical** -- use resources well
 - represent data by packing memory
- **Portable** -- run well on any computer with C compiler
 - all computers are universal, but with C fast programs are fast everywhere
- **Easy to write** -- we know many 'good' techniques
 - reference data, don't copy

These qualities all derive from von Neuman model

Von Neumann (RAM) Model

- Call the 'standard' model of a random access machine (RAM) the von Neumann model
 - A processor interpreting 3-address instructions
 - PC pointing to the next instruction of program in memory
 - "Flat," randomly accessed memory requires 1 time unit
 - Memory is composed of fixed-size addressable units
 - One instruction executes at a time, and is completed before the next instruction executes
- The model is not literally true, e.g., memory is hierarchical but made to "look flat"

C directly implements this model in a HLL

Why Use Model That's Not Literally True?

- Simple is better, and many things--GPRs, floating point format--don't matter at all
- Avoid embedding assumptions where things could change ...
 - Flat memory, though originally true, is no longer right, but we don't retrofit the model; we don't want people "programming to the cache"
 - Yes, exploit spatial locality
 - No, avoid blocking to fit in cache line, or tricking cache into prefetch, etc.
 - Compilers bind late, particularize and are better than you are!

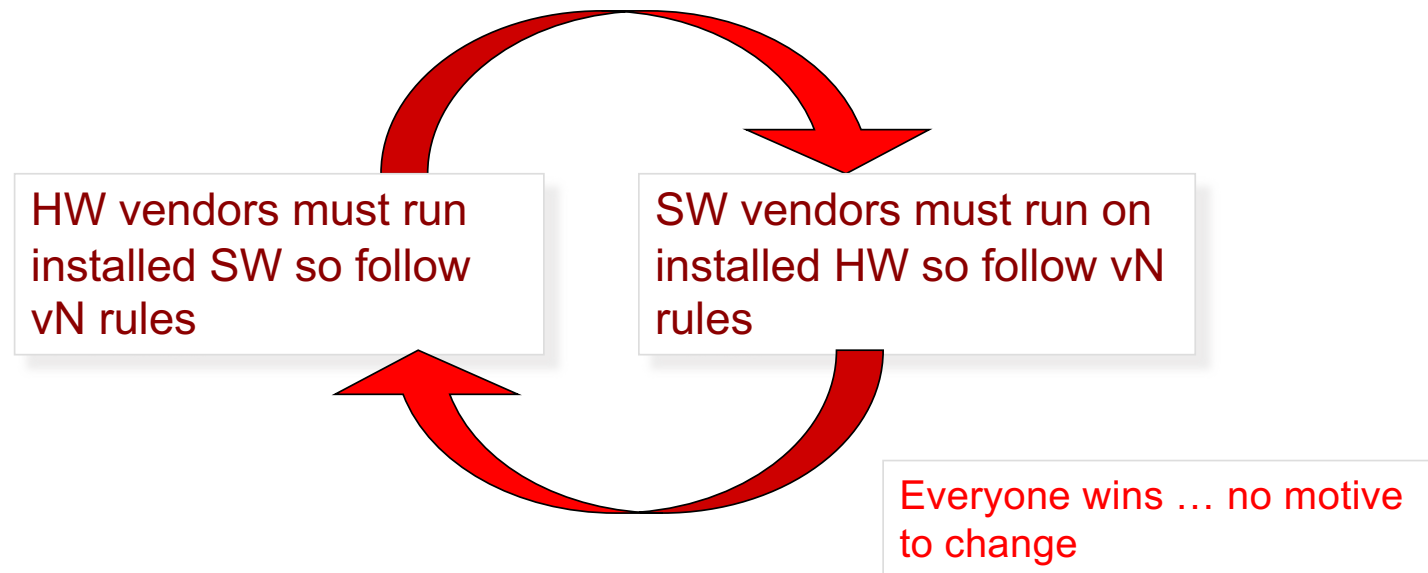
vN Model Contributes To Success

- The cost of C statements on the vN machine is “understood” by C programmers ...
- How much time does $A[r][s] += B[r][s];$ take?
 - Load row_size_A, row_size_B, r, s, A_base, B_base (6)
 - $\text{tempa} = (\text{row_size_A} * r + s) * \text{data_size}$ (3)
 - $\text{tempb} = (\text{row_size_B} * r + s) * \text{data_size}$ (3)
 - $A_base + \text{tempa}; B_base + \text{tempb};$ load both values (4)
 - Add values and return to memory (2)
 - Same for many operations, any data size
- Result is measured in “instructions” not time

Widely known and effectively used

Portability

- Most important property of the C-vN coupling:
It is approximately right everywhere
- Why so little variation in sequential computers?



Von Neumann Summary

- The von Neumann model “explains” the costs of C because C expresses the facilities of the von Neumann machines in programming terms
- Knowing the relationship between C and the von Neumann machine is essential for writing fast programs
- Following the rules produces good results everywhere because everyone benefits
- These ideas are “in our bones” ... it’s how we think

What is the parallel version of vN?

Two searching computations (vN example)

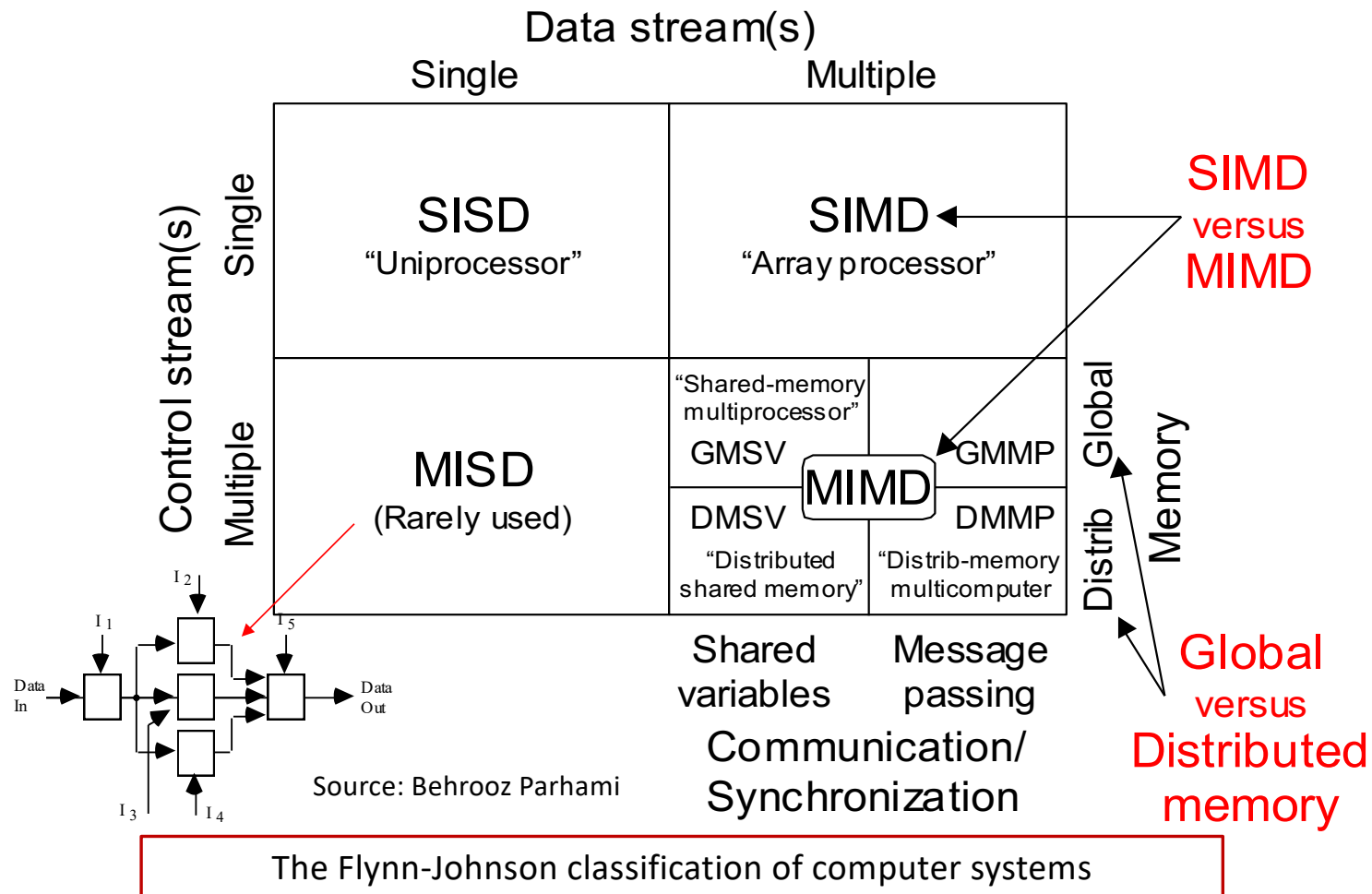
- linear search and binary search

```
1  location=-1;
2  for(j=0; j<n; j++)
3  {
4      if(A[j]==searchee)
5      {
6          location=j;
7          break;
8      }
9  }
```

```
1  location=-1;
2  hi=n-1;
3  lo=0;
4  while(lo!=hi)
5  {
6      mid=lo+floor((hi-lo+1)/2);
7      if(A[mid]==searchee)
8          break;
9      if(A[mid]>searchee)
10         hi=mid;
11     else
12         lo=mid+1;
13 }
```

What is the parallel version of vN?

Flynn-Johnson Classification



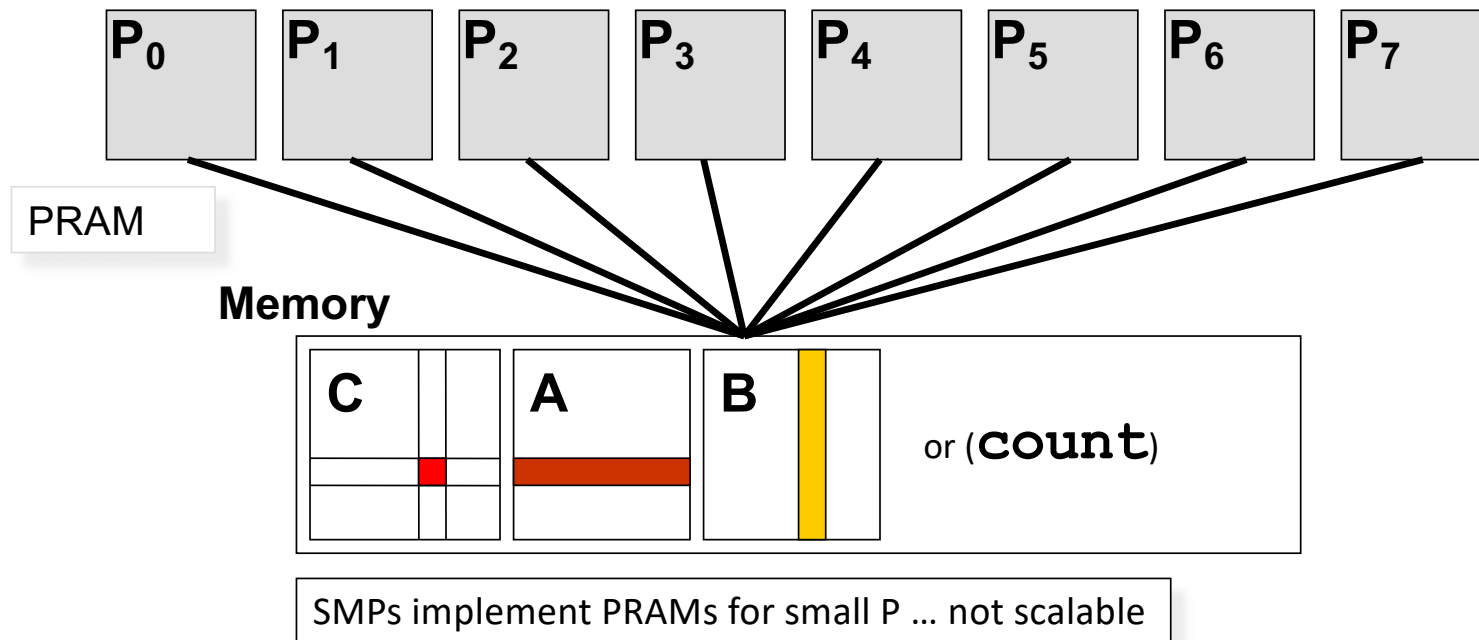
PRAM Often Proposed As A Candidate

- PRAM (Parallel RAM) ignores memory organization, collisions, latency, conflicts, etc.
- Ignoring these are *claimed* to have benefits ...
 - Portable everywhere since it is very general
 - It is a simple programming model ignoring only insignificant details -- off by "only log P"
 - Ignoring memory difficulties is OK because hardware can "fake" a shared memory
 - Good for getting started: Begin with PRAM then refine the program to a practical solution if needed

Recall Parallel Random-Access Machine

PRAM has any number of processors

- Every proc references any memory in "time 1"
- Memory read/write collisions must be resolved



Variations on PRAM

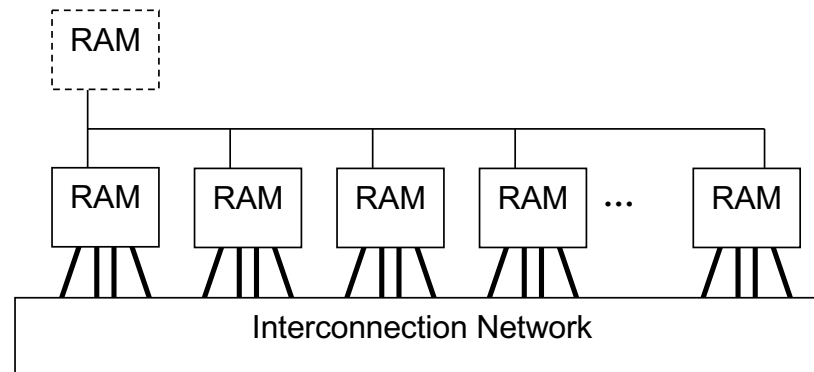
Resolving the memory conflicts considers read and write conflicts separately

- Exclusive read/exclusive write (EREW)
 - The most limited model
- Concurrent read/exclusive write (CREW)
 - Multiple readers are OK
- Concurrent read/concurrent write (CRCW)
 - Various write-conflict resolutions used
- There are at least a dozen other variations

All theoretical -- not used in practice

CTA Model

- Candidate Type Architecture: A model with P standard processors, d degree, λ latency



- Node == processor + memory + NIC

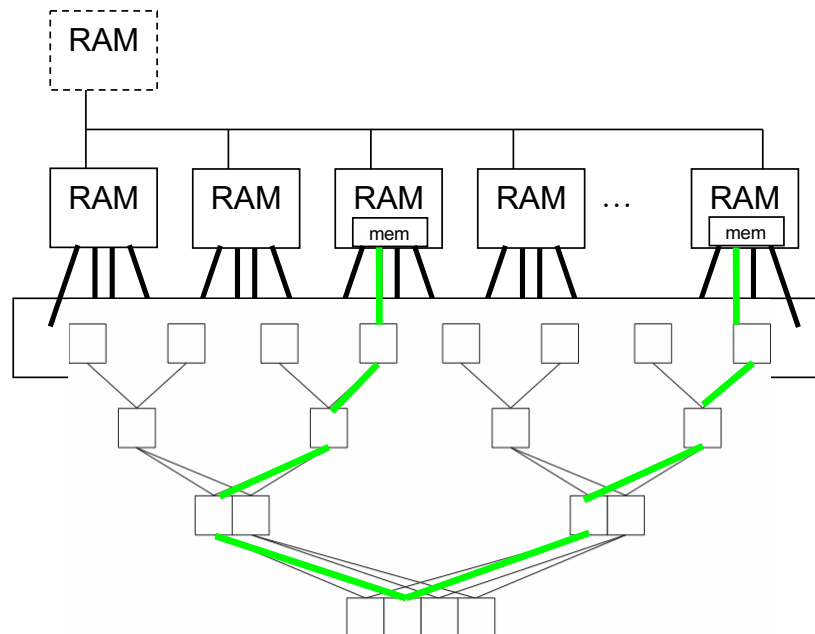
Key Property: Local memory ref is 1, global memory is λ

What CTA Doesn't Describe

- CTA has no global memory ... but memory could be globally *addressed*
- Mechanism for referencing memory not specified: shared, message passing, 1-side
- Interconnection network not specified
- λ is not specified beyond $\lambda \gg 1$ -- cannot be because every machine is different
- Controller, combining network "optional"

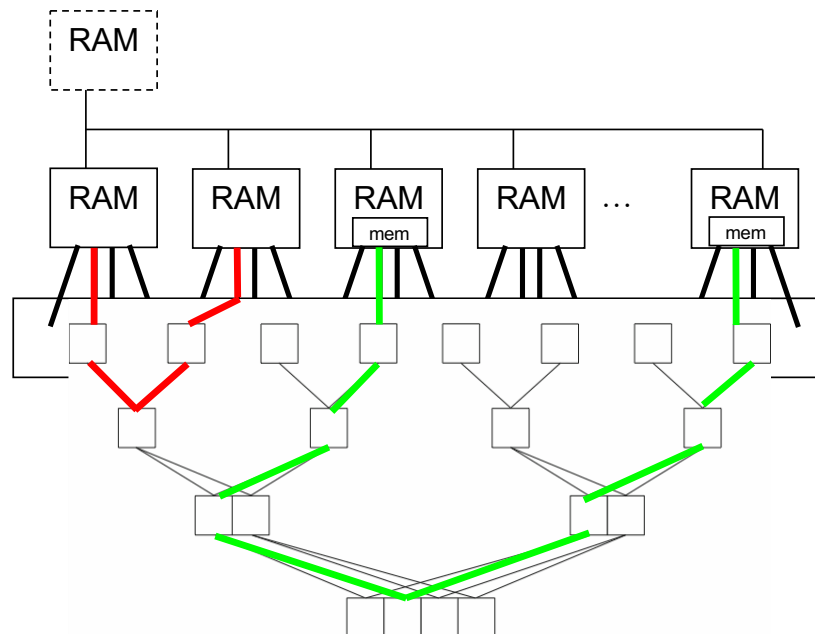
More On the CTA

- Consider what the diagram means...



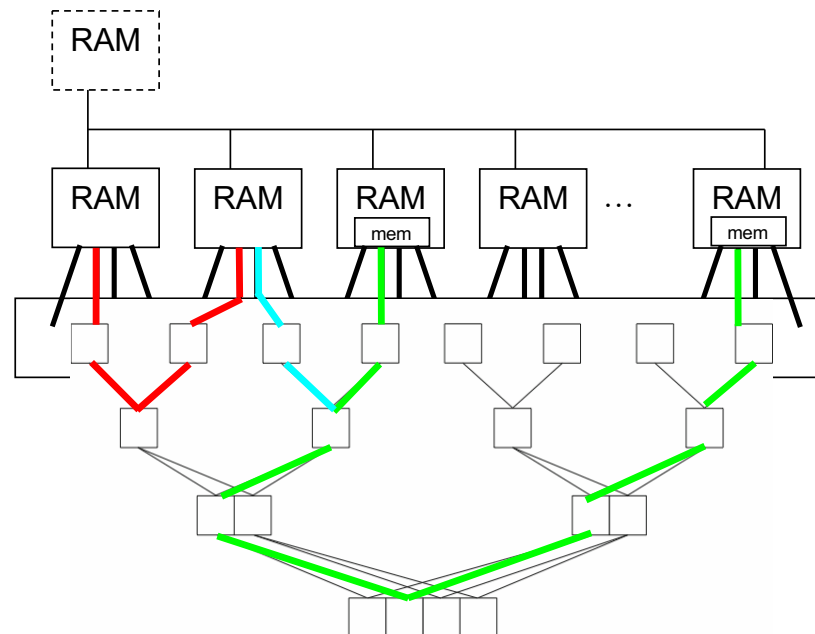
More On the CTA

- Consider what the diagram means...



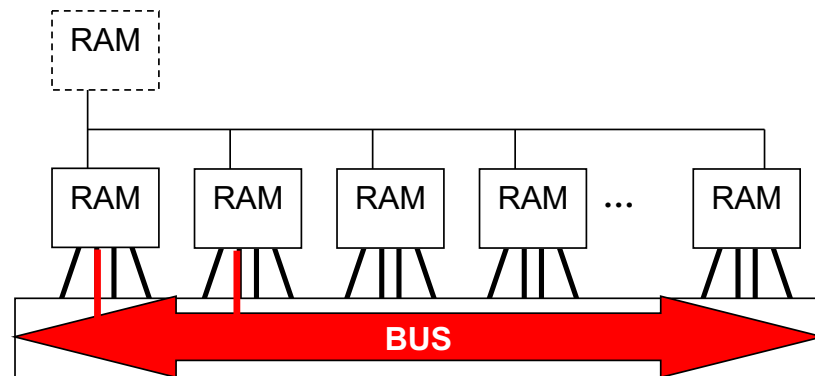
More On the CTA

- Consider what the diagram means...



More On the CTA

- Consider what the diagram **doesn't** mean...



- After ACKing that CTA doesn't model buses, accept that it's a good first approximation

Typical Values for λ

- Lambda can be estimated for any machine (given numbers include **no** contention or congestion)

CMP	AMD	100
SMP	Sun Fire E25K	400-660
Cluster	Itanium + Myrinet	4,100-5,100
Super	BlueGene/L	5,000

} Lg λ range
=> cannot
be ignored

As with merchandizing: **It's location, location, location!**

Measured Numbers

- Values (approximating) λ for small systems

System	Cache	Latency cycles	Throughput msgs/kcycle
2×4-core Intel	shared	180	11.97
	non-shared	570	3.78
2×2-core AMD	same die	450	3.42
	one-hop	532	3.19
4×4-core AMD	shared	448	3.57
	one-hop	545	3.53
	two-hop	659	3.19
8×4-core AMD	shared	538	2.77
	one-hop	613	2.79
	two-hop	682	2.71

Communication Mechanisms

- Shared addressing
 - One consistent memory image; primitives are load and store
 - Must protect locations from races
 - Widely considered most convenient, though it is often tough to get a program to perform
 - CTA implies that best practice is to keep as much of the problem private; use sharing only to communicate

A common pitfall: Logic is too fine grain

Communication Mechanisms

- Message Passing
 - No global memory image; primitives are `send()` and `recv()`
 - Required for most large machines
 - User writes in sequential language with message passing library:
 - Message Passing Interface (MPI)
 - Parallel Virtual Machine (PVM)
 - CTA implies that best practice is to build and use own abstractions

Lack of abstractions makes message passing brutal

Communication Mechanisms

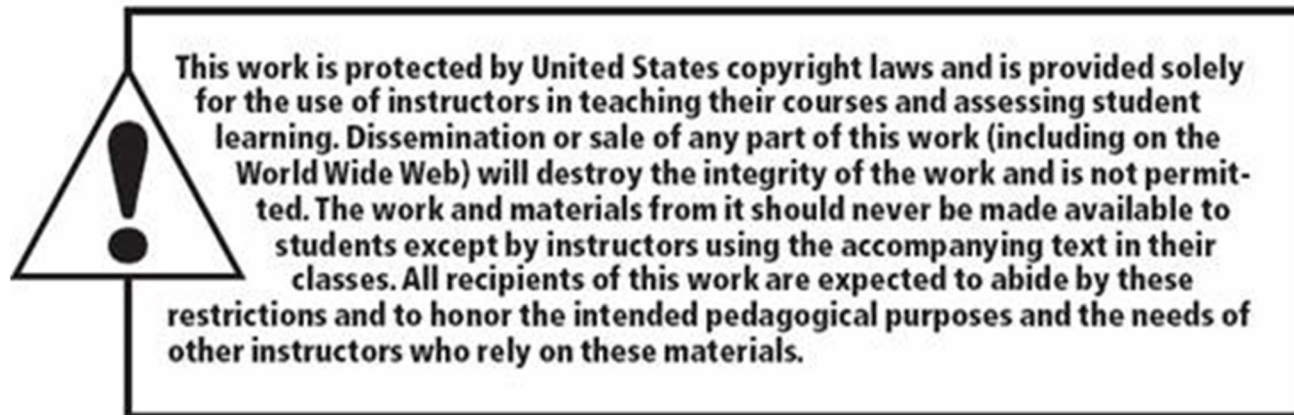
- One Sided Communication
 - One global address space; primitives are `get()` and `put()`
 - Consistency is the programmer's responsibility
 - Elevating mem copy to a comm mechanism
 - Programmer writes in sequential language with library calls -- not widely available unfortunately
 - CTA implies that best practice is to build and use own abstractions

One-sided is lighter weight than message passing

Summary

- Parallel hardware is a critical component of improving performance through ||-ism ... but there's a Catch-22
 - To have portable programs, we must abstract away from the hardware
 - To write performant programs requires that we respect the hardware realities
- Solve the problem with CTA -- an abstract machine with just enough (realizable) detail to support critical programming decisions

Copyright



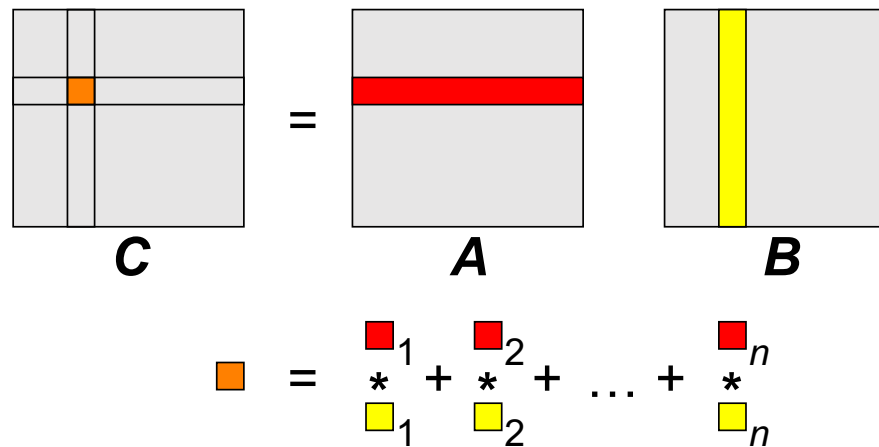
Matrix Product: || Poster Algorithm

- Matrix multiplication is most studied parallel algorithm (analogous to sequential sorting)
- Many solutions known
 - Illustrate a variety of complications
 - Demonstrate great solutions
- Our goal: explore variety of issues
 - Amount of concurrency
 - Data placement
 - Granularity

Exceptional by requiring $O(n^3)$ operations on $O(n^2)$ data

Recall the computation...

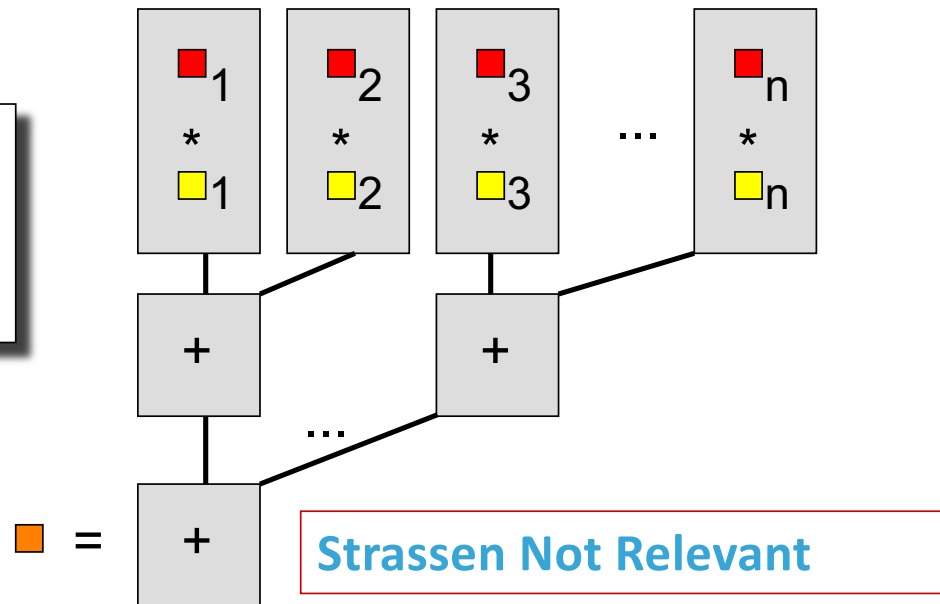
- Matrix multiplication of (square $n \times n$) matrices \mathbf{A} and \mathbf{B} producing $n \times n$ result \mathbf{C} where $\mathbf{C}_{rs} = \sum_{1 \leq k \leq n} \mathbf{A}_{rk} * \mathbf{B}_{ks}$



Extreme Matrix Multiplication

- The multiplications are independent (do in any order) and the adds can be done in a tree

$O(n)$ processors for
each result element
implies $O(n^3)$ total
Time: $O(\log n)$



$O(\log n)$ MM in the real world ...

Good properties

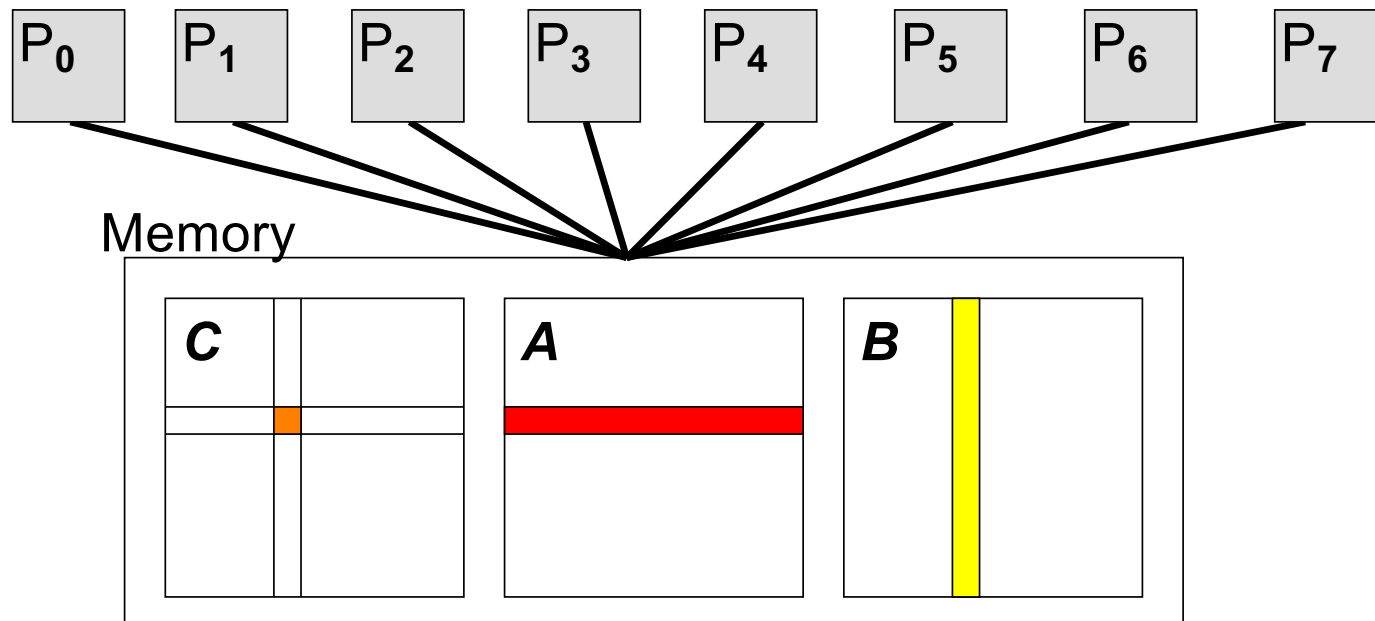
- Extremely parallel ... shows limit of concurrency
- Very fast -- $\log_2 n$ is a good bound ... faster?

Bad properties

- Ignores memory structure and reference collisions
- Ignores data motion and communication costs
- Under-uses processors -- half of the processors do only 1 operation

Where is the data?

- Data references collisions and communication costs are important to final result ... need a model ... can generalize the standard RAM to get PRAM



Parallel Random Access Machine

- Any number of processors, including n^c
- Any processor can reference any memory in “unit time”
- Resolve Memory Collisions
 - Read Collisions -- simultaneous reads to location are OK
 - Write Collisions -- simultaneous writes to loc need a rule:
 - Allowed, but must all write the same value
 - Allowed, but value from highest indexed processor wins
 - Allowed, but a random value wins
 - Prohibited

Caution: The PRAM is *not* a model we advocate

PRAM says $O(\log n)$ MM is good

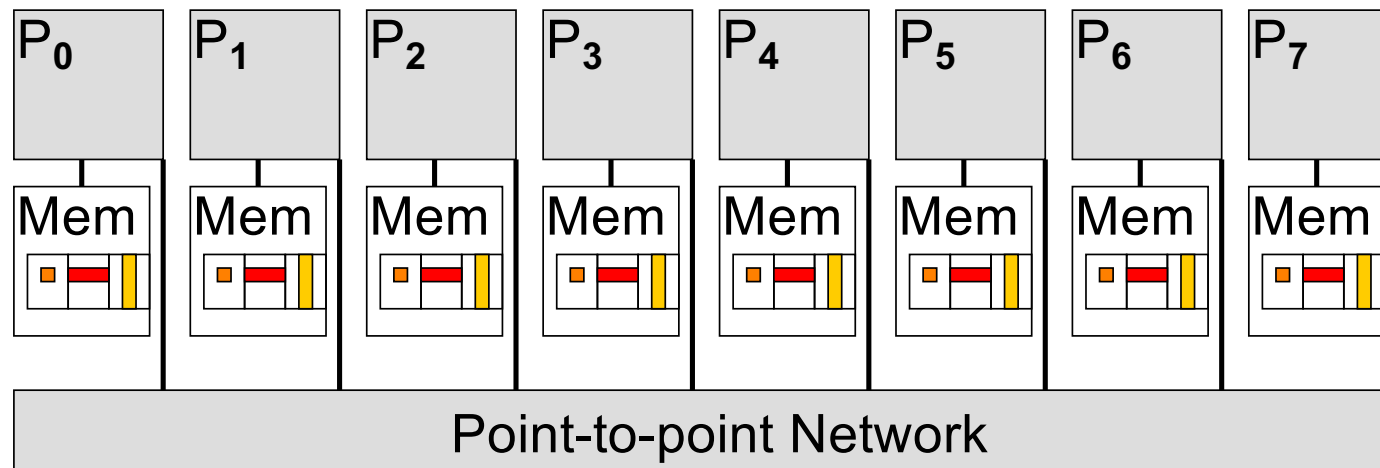
- PRAM allows any # processors $\Rightarrow O(n^3)$ OK
- **A** and **B** matrices are read simultaneously, but that's OK
- **C** is written simultaneously, but no location is written by more than 1 processor \Rightarrow OK

PRAM model implies $O(\log n)$ algorithm is best ... but in real world, we suspect not

We return to this point later

Where else could data be?

- Local memories of separate processors ...

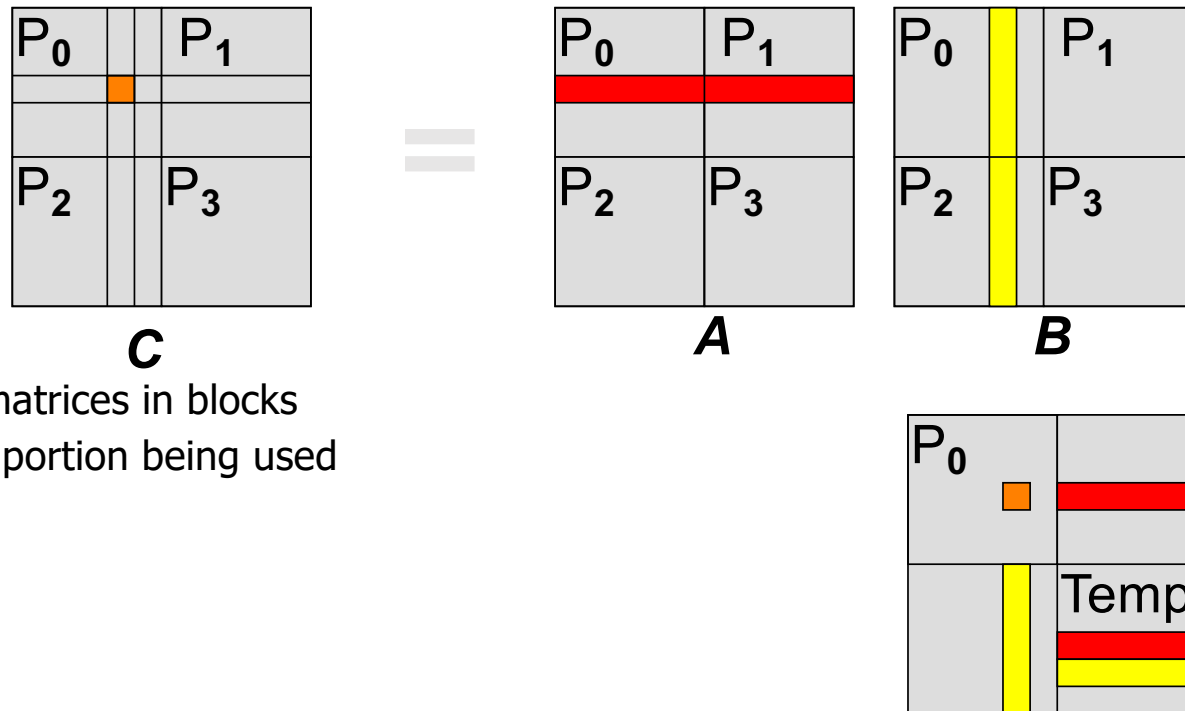


- Each processor could compute block of ***C***
 - Avoid keeping multiple copies of ***A*** and ***B***

Architecture common for servers

Data Motion

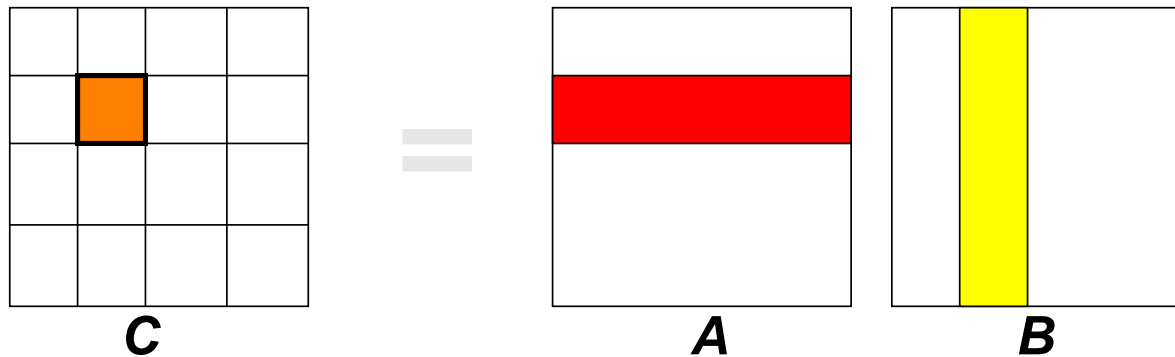
- Getting rows and columns to processors



- Allocate matrices in blocks
- Ship only portion being used

Blocking Improves Locality

- Compute a $b \times b$ block of the result



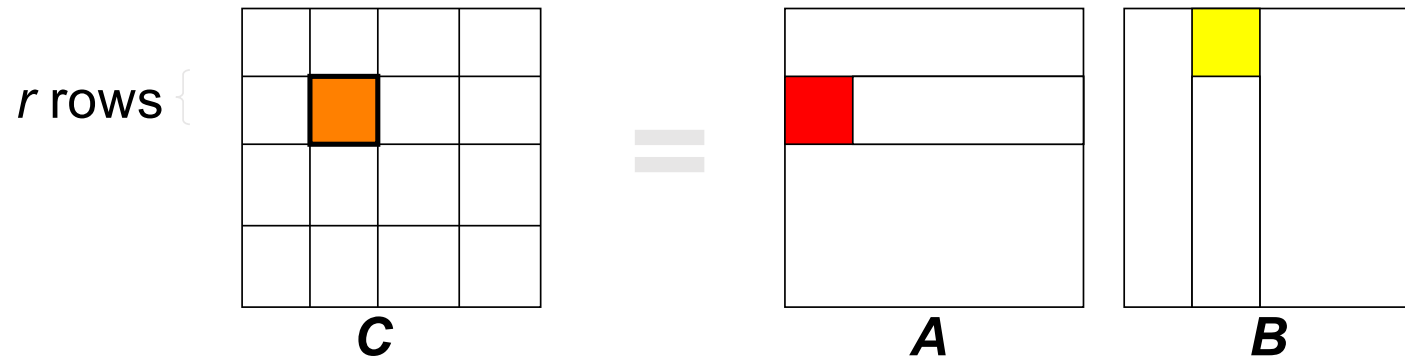
- Advantages
 - Reuse of rows, columns = caching effect
 - Larger blocks of local computation = high locality

Caching in Parallel Computers

- Blocking = caching ... why not automatic?
 - Blocking improves locality, but it is generally a manual optimization in sequential computation
 - Caching exploits two forms of locality
 - Temporal locality -- refs clustered in time
 - Spatial locality -- refs clustered by address
- *When multiple threads touch the data, global reference sequence may not exhibit clustering features typical of one thread -- thrashing*

Sweeter Blocking

- It's possible to do even better blocking ...

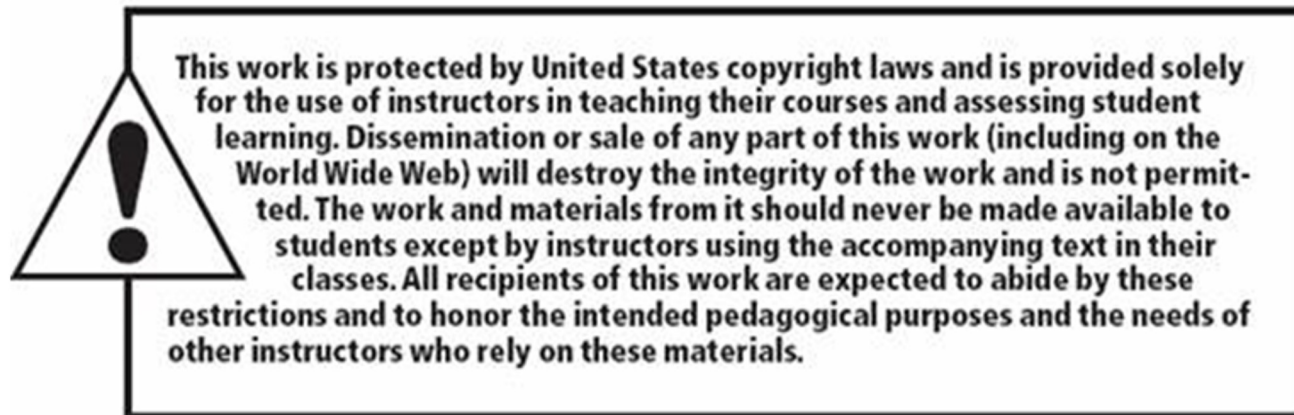


- Completely use the cached values before reloading

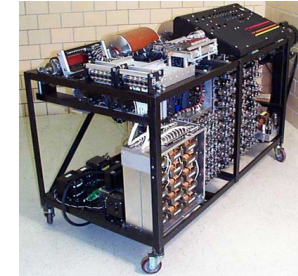
Best MM Algorithm?

- We haven't decided on a good MM solution
- A variety of factors have emerged
 - A processor's connection to memory, unknown
 - Number of processors available, unknown
 - Locality--always important in computing--
 - Using caching is complicated by multiple threads
 - Contrary to high levels of parallelism
- **Conclusion:** Need a better understanding of the constraints of parallelism

Copyright



Historical Milestones



- 1935 – ABC the first electronic computer ever (John V. Atanasoff: **seven processing units** for solving a system of 30 differential equations)
- 1968 – Univ. Illinois built a parallel computer (lost in fire 😞)
- 1981 – DAP (**4K processing units**)
- 1986 – Connection Machine 1 (**65K processors**, parallel Fortran)
- 1990 – Parallel computing becomes mainstream in HPC
- 2005 – personal computers with dual-core processors
- 2005 – GPUs outperform CPUs on LU factorization
- 2010 – multi-core processors become mainstream in high performance computing
- 2012 – first parallel computer with more than **one million cores**
- 2021 – still waiting for the first Exa-scale supercomputer

RUG Milestones

- 1991 - appoints first professor of parallel computing in NL
- 1992 – Connection Machine CM5 (2K proc)
first parallel computer in NL, (rank 123 in Top 500 list)
- 1993 – Center for HPC
- 1996 – Cray system (32 processors)
- 2000 – first large PC cluster (256 computers)
- 2005 – IBM Blue Gene (12K processors)
(rank 6 in Top 500 list)
- 2015 – Peregrine cluster (4,368 cores, 30 Tbyte)
- 2021 – the upgrade of the system is in progress

Top500 list: www.top500.org

Released in June and November since 1993

Benchmark: Linpack kernel used in solvers of dense systems of linear equations. (DAXPY performance **does not** reflect the real performance of a given system)

- **Rmax** - performance on Linpack (used for Top500 rank)
- **Rpeak** – theoretical peak performance (number of arithmetic floating point operations a system can execute per second)

There is also the Green500 list (performance per Watt):

<https://www.top500.org/lists/green500/list/2020/11/>

Q: How do you think are these two lists related?

The DAXPY loop (the core of Linpack)

```
void daxpy(size_t n, double a, const double x[], double y[])
{
    for (size_t i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

Q: How useful is the above kernel for sequence alignment?

Real genomics problem: Identify the **Fragile X syndrome** (FXS) condition

In nucleic acid sequences, e.g., AACCTGA..., look for the following pattern:

- **One** occurrence of **GCG**
- Followed by **any number** of **CGG or AGG** (typically 55..200)
- Followed by **CTG**

Q2: How relevant is the Linpack performance of your machine?

What are flop/s?

- flop/s (FLOPS) - floating-point operations per second

Common abbreviations

- Megaflop/s (Mflop/s) = 10^6 flop/s 1975
- Gigaflop/s (Gflop/s) = 10^9 flop/s 1985
- Teraflop/s (Tflop/s) = 10^{12} flop/s 1997
- Petaflop/s (Pflop/s) = 10^{15} flop/s 2008
- Exaflop/s (Xflop/s) = 10^{18} flop/s **202?**

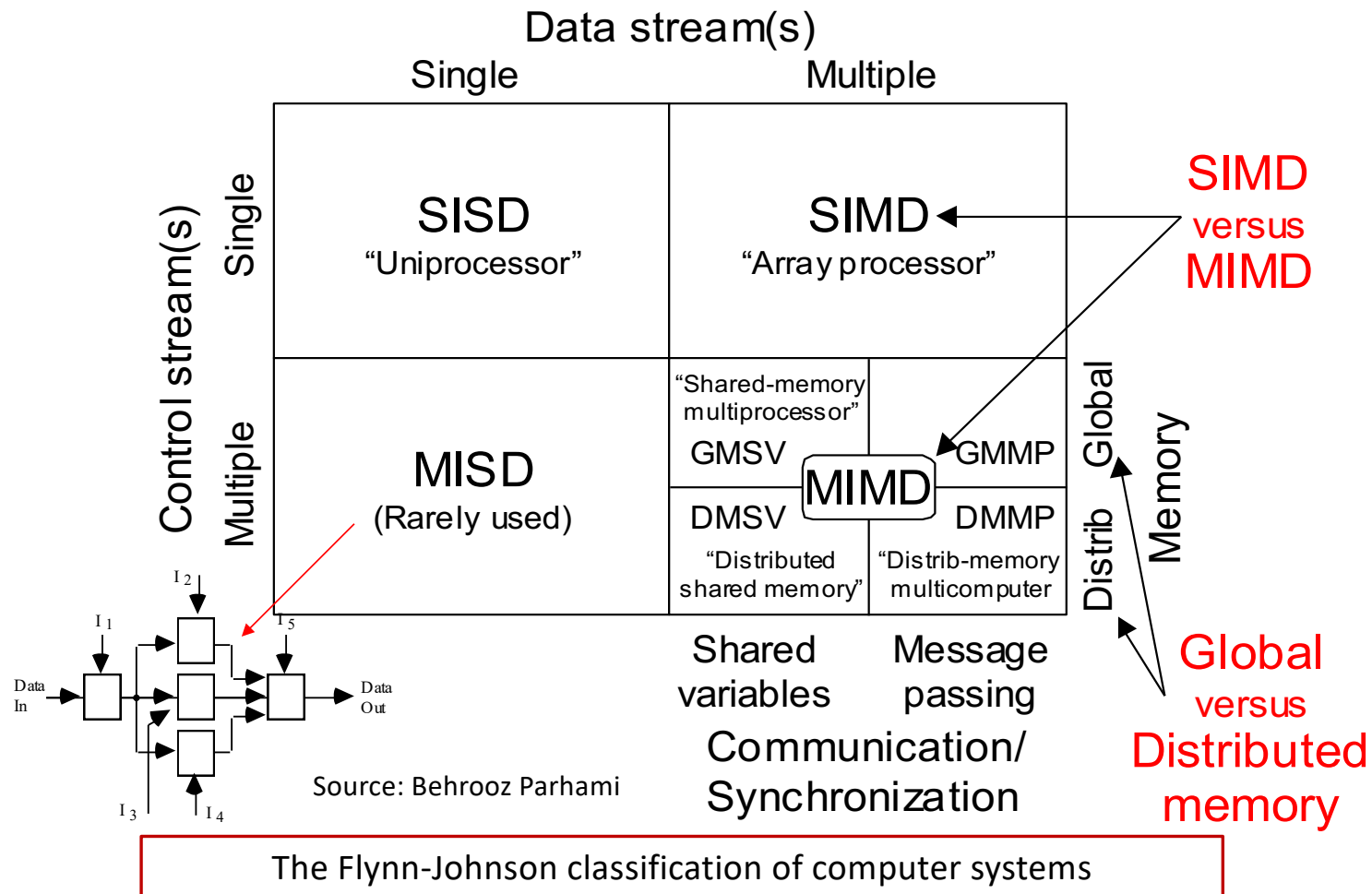
Q: Why the exaflop deadline for a single machine keeps shifting?

Q2: Are the DAXPY flop/s a fair metric?

Q3: What about single-(half-) precision floating-point?

Wikipedia: In April 2020, the distributed computing Folding@home network attained **one exaFLOPS** of computing performance.

Flynn-Johnson Classification



SIMD

- Single Instruction, Multiple Data
- One instruction stream is broadcast to all processors
- Each processor, also called a *processing element* (PE), is usually simplistic and logically is essentially an ALU
 - PEs do not store a copy of the program nor have a program control unit
- Individual processors can remain idle during execution of segments of the program (based on a data test)

SIMD (cont.)

- All active processors execute the same instruction **synchronously**, but on different data
- Technically, on a memory access, all active processors must access the *same location* in their local memory
 - This requirement is sometimes relaxed a bit
- The data items form an array (or vector) and an instruction can act on the complete array in one cycle
- Examples:
 - ILLIAC IV (1974) was the first SIMD computer
 - The STARAN and MPP
 - Connection Machine CM2 (by Thinking Machines)
 - MasPar MP-1 (for Massively Parallel) computers

How to View a SIMD Machine

- Think of all soldiers in a unit
- The commander selects certain soldiers as active, e.g., the first row
- The commander barks out an order to all the active soldiers, who execute the order synchronously
 - The remaining soldiers do not execute orders until they are re-activated

Single Program Multiple Data (SPMD), e.g., GPUs, **is not** == SIMD

MIMD

- Multiple Instructions, Multiple Data
- Processors are **asynchronous** (execute independently different programs on different data sets)
- Communications are handled either
 - through Shared Memory (SM) (**multiprocessors**)
 - by use of Message Passing (MP) (**multicomputers**)
- MIMD's have been widely considered to include the most powerful and least restricted computers

MIMD (cont. 2/4)

- Have very major communication costs
 - When compared to SIMDs
 - Internal 'housekeeping activities' are often overlooked
 - Maintaining distributed memory & distributed databases
 - Synchronization or scheduling of tasks
 - Load balancing between processors
- One method for programming MIMDs is for all processors to execute the same program
 - Execution of tasks by processors is still asynchronous
 - Called **SPMD** method (Single Program, Multiple Data)
 - Usual method for massive number of processors (GPU)
 - Considered to be a "data parallel programming" style for MIMDs

MIMD (cont 3/4)

- A more common prog. technique is **multi-tasking**:
 - The problem solution is broken up into various tasks
 - Tasks are distributed among processors initially
 - If new tasks are produced during executions, these may be handled by parent processor or distributed
 - Each processor concurrently executes its collection of tasks
 - If some of its tasks must wait for results from other tasks or new data, the processor will focus the remaining tasks
 - Larger programs usually run a load balancing algorithm in the background that re-distributes the tasks assigned to the processors during execution
 - Either dynamic load balancing or called at specific times
 - Dynamic scheduling algorithms may be needed to assign a higher execution priority to time-critical tasks
 - e.g., on critical path, more important, earlier deadline, etc.

MIMD (cont 4/4)

- Recall, there are two principle types of MIMD computers:
 - Multiprocessors (with shared memory)
 - Multicomputers (with message passing)
- Both are important and are covered in great detail

Multiprocessors (Shared Memory)

- All processors have access to all memory locations
- Two types: UMA and NUMA
- **UMA** (uniform memory access)
 - Frequently called *symmetric multiprocessors* or *SMPs*
 - Similar to uniprocessor, except additional, identical CPU's are added to the bus
 - Each processor has equal access to memory and can do anything that any other processor can do
 - SMPs have been and remain very popular

Multiprocessors (cont.)

- **NUMA** (non-uniform memory access)
 - Has a distributed memory system
 - Each memory location has the same address for all PEs
 - Memory access time to a given location **varies considerably** for different CPUs
- Normally, fast cache is used to reduce the problem of different NUMA memory access times
 - Creates problem of ensuring all copies of the same data in different memory locations are identical

Q: Why is the latter important?

Multicomputers (Message-Passing)

- Processors are connected by a **network**
 - Interconnection network connections is one possibility
 - Also, may be connected by Ethernet links or a bus
- Each processor has a local memory and can only access its own local memory
- **Data is passed** between processors **using messages**, when specified by the program
- Message passing between processors is controlled by a message passing language (typically MPI)
- The problem is divided into ***processes*** or ***tasks*** that can be executed concurrently on individual processors. Each processor is normally assigned multiple processes.

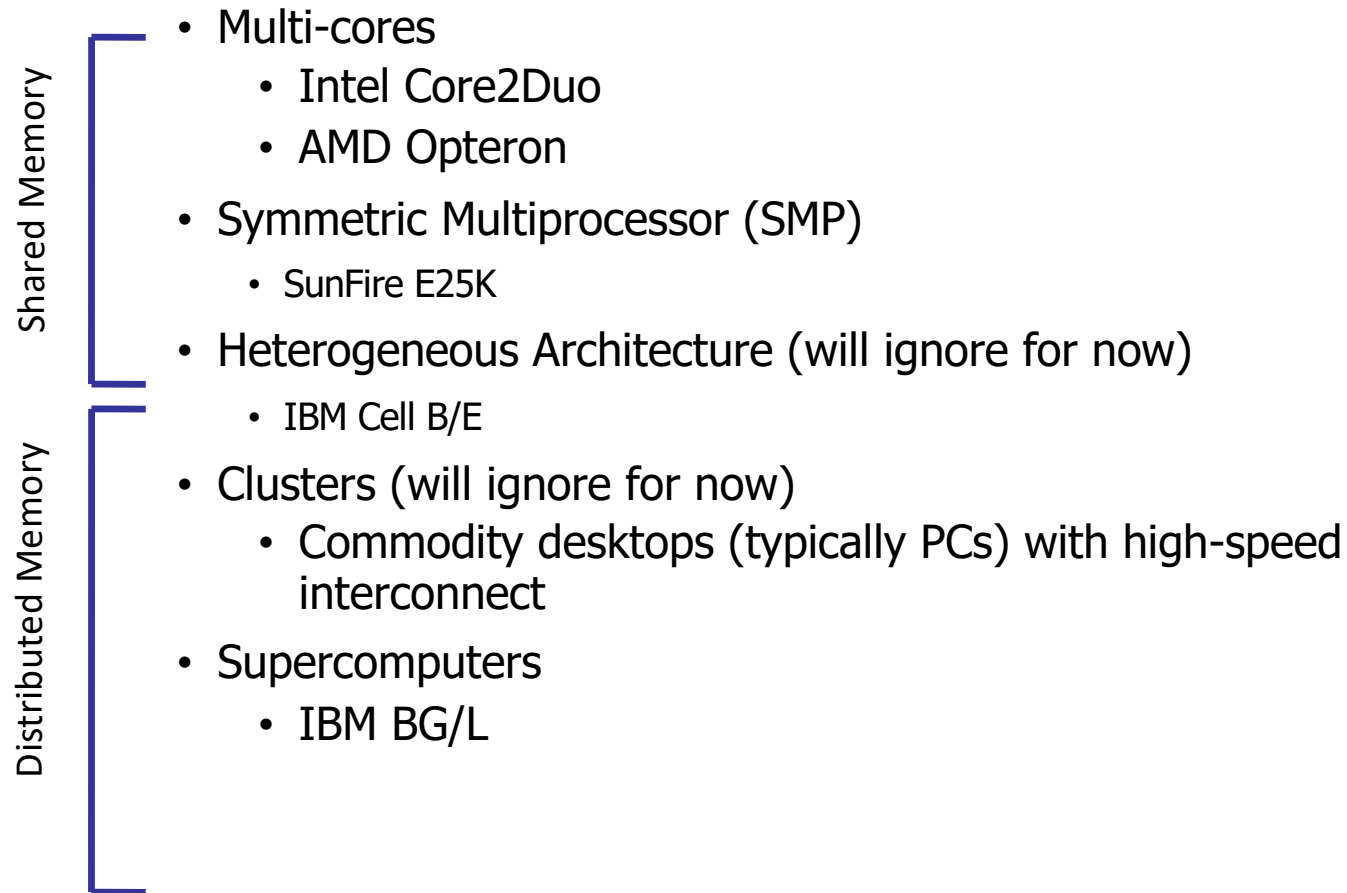
Multiprocessors vs Multicomputers

- Programming disadvantages of message-passing
 - Programmers must make explicit message-passing calls in the code
 - This is low-level programming and is quite error prone
 - Data is not shared between processors but copied, which increases the total data size
 - Data integrity problem: Difficulty to maintain correctness of multiple copies of a data item

Multiprocessors vs Multicomputers (cont)

- Programming advantages of message-passing
 - No problem with simultaneous access to data
 - Allows different PCs to operate on the same data independently
 - Allows PCs on a network to be easily upgraded when faster processors become available
- Mixed “distributed shared memory” systems exist
 - Lots of current interest in a cluster of SMPs
- Easier to build systems with a very large number of processors

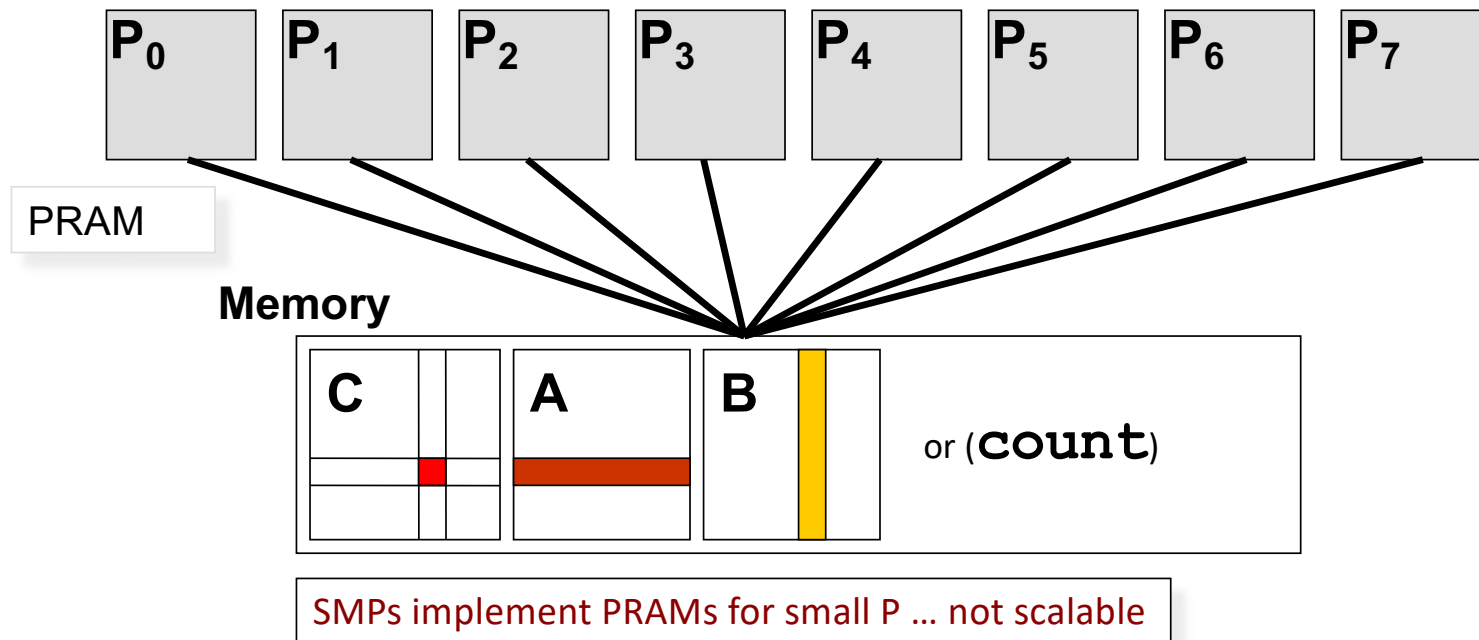
The Six Parallel Architectures



Recall Parallel Random-Access Machine

PRAM has any number of processors

- Every proc references any memory in "time 1"
- Memory read/write collisions must be resolved



Variations on PRAM

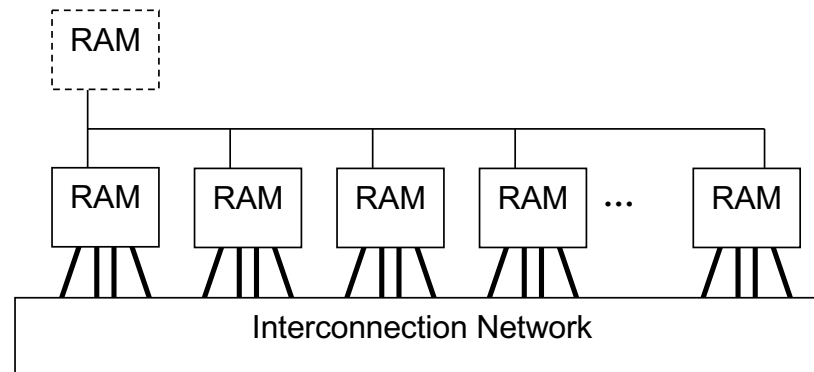
Resolving the memory conflicts considers read and write conflicts separately

- Exclusive read/exclusive write (EREW)
 - The most limited model
- Concurrent read/exclusive write (CREW)
 - Multiple readers are OK
- Concurrent read/concurrent write (CRCW)
 - Various write-conflict resolutions used
- There are at least a dozen other variations

All theoretical -- not used in practice

CTA Model

- Candidate Type Architecture: A model with P standard processors, d degree, λ latency



- Node == processor + memory + NIC

Key Property: Local memory ref is 1, global memory is λ

What CTA Doesn't Describe

- CTA has no global memory ... but memory could be globally *addressed*
- Mechanism for referencing memory not specified: **shared, message passing, 1-side**
- Interconnection network not specified
- λ is not specified beyond $\lambda \gg 1$ -- cannot be because every machine is different
- Controller, combining network "optional"

Communication Mechanisms (1)

- **Shared addressing**

- One consistent memory image; primitives are load and store
- Must protect locations from races
- Widely considered most convenient, though it is often tough to get a program to perform
- CTA implies that best practice is to keep as much of the problem private; use sharing only to communicate

A common pitfall: Logic is too fine grain

Communication Mechanisms (2)

- **Message Passing**

- No global memory image; primitives are `send()` and `recv()`
- Required for most large machines
- User writes in sequential language with message passing library:
 - Message Passing Interface (MPI)
 - Parallel Virtual Machine (PVM)
- CTA implies that best practice is to build and use own abstractions

Lack of abstractions makes message passing brutal

Communication Mechanisms (3)

- **One Sided Communication**

- One global address space; primitives are `get()` and `put()`
- Consistency is the programmer's responsibility
- Elevating mem copy to a comm mechanism
- Programmer writes in sequential language with library calls -- not widely available unfortunately
- CTA implies that best practice is to build and use own abstractions

One-sided is lighter weight than message passing

Finishing the Discussion on CTA

- The CTA is supposed to guide us in finding good computations to run on parallel machines
- Using it should
 - Aid in producing programs exploiting locality
 - Insure the program distributes work 'well'
 - Other features (to be discussed later)

A motivational example

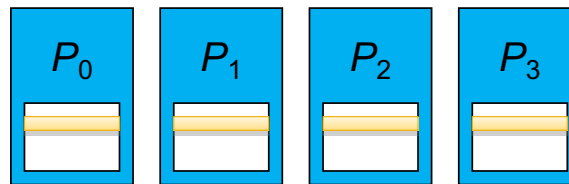
- **Task:** Recognize the well-formedness of ((xyz))
- An easy sequential solution ...

```
open = 0;                                // keep count of opens
for (i=0; i<n; i++) {                    // proceeding L to R
    if (A[i] == '(' ) open++;           // found one
    if (A[i] == ')' ) {                  // here's a match
        open--;
        if (open < 0) break;           // oops, mismatch
    }
}
```

Does this look totally sequential??

Proceeding As Usual

- Allocate a contiguous sequence of symbols to a processor



- Each processor gets an ill-formed subsequence
 - $(x))(((x)xxx(x))$
- Begin by resolving locally
 - ~~(x)~~ $)$ $($ ~~(x)~~ $xxx(x)$

Leaving unresolved closes and opens

The Global Steps

- The unresolved values from each subproblem produce a similar problem, except optimized
) (becomes 1 1 and) ((((becomes 1 4
- Adjacent pairs *combine* their unresolved counts to a new pair describing the larger sequence:
 1 1 and 1 4 become 1 4
 2 4 and 1 2 become 2 5
- Resolved to the root: 0 0 is balanced


Summarizing the Solution

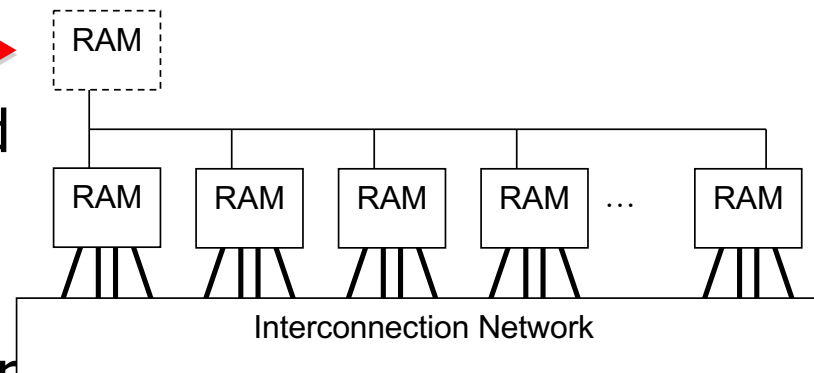
- Allocate contiguous subsequences of size n/P to each processor, starting with P_0
- Sequentially, locally resolve, creating $c \ o \ \quad cn/P$
- Combine pairs to produce new $c \ o$ descriptors by inducing a tree on PE indices: [0-1][2-3] ... for level 1, [0-3][4-7] ... for level 2, etc.
- Log levels of the tree to produce a final descriptor: $c \ o \quad c \log_2 P$
- Only a result of $0 \ 0$ means balanced

Where Was The Focus?

- **First step:** Allocated work to processors, generally by dividing it evenly
- **Second step:** Found local, independent work to perform
- **Next step:** Focused on combining subproblems into a tree network
- Made correctness and termination conditions explicit

Completing the CTA Discussion

- Controller 
 - Not strictly needed
 - Often available
- How well does the CTA match other parallel architectures?
 - CMPs & SMPs
 - Clusters
 - Blue Gene



Precision of the CTA

- The CTA is a 'machine model' – an abstraction
- How can it be wrong?
 - Architecture has more features – shared memory
 - CTA predicts a certain behavior and features in the architecture make the program much faster
 - If it mispredicts ... it's "in trouble"
- Isn't it a mistake for the CTA to ignore all the great stuff architects put in a processor?

The CTA focuses on the parts that matter

Using the CTA

- Why should we believe it's right?
 - In his thesis (1993) Calvin Lin did a careful study of using the CTA as a programming model against the models used by others (whatever they were)
 - CTA consistently pointed programmers to better solutions
 - The CTA's effectiveness was independent of architecture
 - The apparent value of the model is emphasizing locality – always a benefit in computing
- The greatest value of the CTA would be if it is the basis for parallel programming languages

Threads

- A thread consists of program code, a program counter, call stack, and a small amount of thread-specific data
 - Threads share access to memory (and the file system) with other threads
 - Threads communicate through the shared memory
 - Though it may seem odd, apply the CTA model to thread programming -
 - emphasize locality, expect sharing to be costly (slow)

Threads are familiar, but don't use standard model

Processes

- A process is a thread in its own private address space
 - Processes do not communicate through shared memory, but need another mechanism like message passing
 - **Key issue:** How is the problem divided among the processes, which includes data and work
 - Processes (logically subsume) threads

Compare Threads & Processes

- Both have code, PC, call stack, local data
 - Threads -- One address space
 - Processes -- Separate address spaces
- Weight and Agility
 - **Threads:** lighter weight, faster to setup, tear down, more dynamic
 - **Processes:** heavier weight, setup and tear down more time consuming, communication is slower

Mostly 'thread' & 'process' are used interchangeably in the book

Terminology

- Terms used to refer to a unit of parallel computation include: thread, process, processor, ...
 - Technically, thread and process are software (SW), processor (including simultaneous multithreading) is hardware (HW)
 - Usually, it doesn't matter
 - We will (try to) use "thread/process" for logical parallelism, and "processor" when we mean physical parallelism

Parallelism vs Performance

- Naïvely, many people think that applying P processors to a T time computation will result in T/P time performance
- **Generally wrong**
 - For a few problems (Monte Carlo) it is possible to apply more processors directly to the solution
 - For most problems, using P processors requires a *paradigm shift* in approach/thinking
- Assume “ P processors $\Rightarrow T/P$ time” to be the best case possible to achieve

Better Intuition

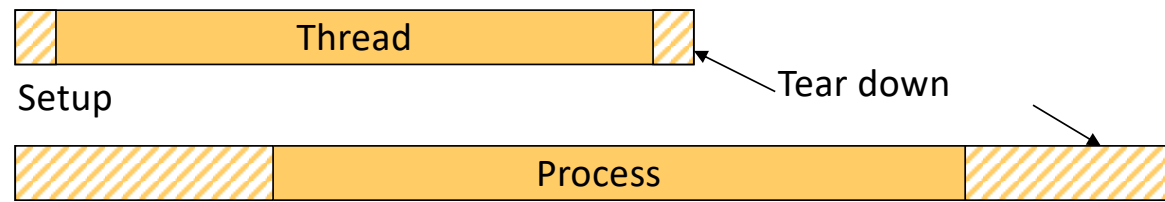
- (Because of the presumed paradigm shift) the sequential and parallel solutions differ so we **do not** expect a simple performance relationship between the two
 - More or fewer instructions must be executed
- Examples of other differences
 - The hardware is different
 - Parallel solution has difficult-to-quantify costs such as communication time, wait time, etc. that the serial solution does not have

More Instructions Needed

- To implement parallel computations requires overhead that sequential computations do not need
 - All costs associated with communication are overhead: locks, cache flushes, coherency, message passing protocols, etc.
 - All costs associated with thread/process setup
 - Lost optimizations -- many compiler optimizations not available in parallel setting
 - Instruction reordering

Performance Loss: Overhead

- Threads and processes incur overhead



- Obviously, the cost of creating a thread or process must be recovered through parallel performance:

$$(t_1 + o_{su} + o_{td} + \text{cost}(t_2))/2 <$$

t_p = p proc execution time
 o_{su} = setup, o_{td} = tear down
 $\text{cost}(t_2)$ = all other || costs

More Instructions Needed (Continued)

- Redundant execution can avoid communication -- a parallel optimization

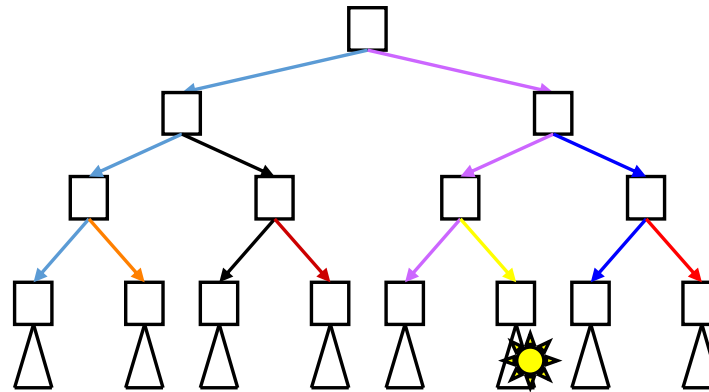
New random number needed for loop iteration:

- (a) Generate one copy, have all threads ref it ... requires communication
- (b) Communicate seed once, then each thread generates its own random number ... removes communication and gets parallelism, but by increasing instruction load

A common (**and recommended**) programming trick

Fewer Instructions

- Searches illustrate the possibility of parallelism requiring fewer instructions



- Independently searching subtrees means an item is likely to be found faster than sequential

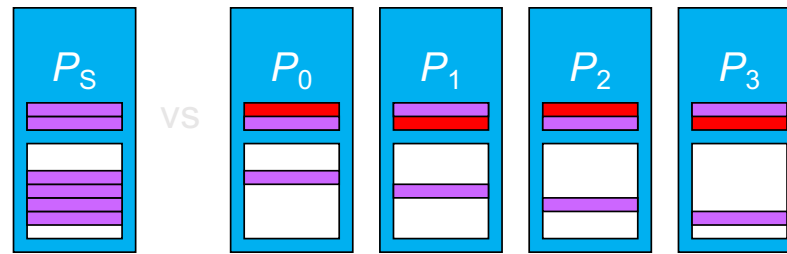
One versus Many

- Sequential hardware \neq parallel hardware
 - There is more parallel hardware, e.g., memory
 - There is more cache on parallel machines
 - Sequential computer \neq 1 processor of || computer, because of coherence HW, power, etc.
 - Important in multicore context
- Parallel channels to memory and disk (possibly)

These differences *tend* to favor || machine

Superlinear Speed up

- Additional cache is an advantage of ||ism



- The effect is to make execution time $< T/P$ because data (& program) memory references are faster
- Cache-effects help mitigate other || costs

Other Parallel Costs

- Wait: All computations must wait at points, but serial computation waits are well known
- Parallel waiting ...
 - For serialization to assure correctness
 - Congestion in communication facilities
 - Bus contention; network congestion; etc.
 - Stalls: data not available/recipient busy
- These costs are generally time-dependent, implying that they are highly variable

Bottom Line ...

- Applying P processors to a problem with a time T (serial) solution can be either ...
better or worse ...
- It's up to programmers to exploit the advantages and avoid the disadvantages

Amdahl's Law, review

- If $1/S$ of a computation is inherently sequential, then the maximum performance improvement is limited to a factor of S

$$T_P = 1/S \times T_S + (1-1/S) \times T_S / P$$

- Remember? Amdahl's Law, ~ the Law of Supply fact

T_S =sequential time
 T_P =parallel time
 P =no. processors

Gene Amdahl -- IBM Mainframe Architect

Interpreting Amdahl's Law

- Consider the equation

$$T_P = 1/S \times T_S + (1-1/S) \times T_S / P$$

- With no charge for || costs, let $P \rightarrow \infty$ then $T_P \rightarrow 1/S \times T_S$

The best parallelism can do to is to eliminate the parallelizable work; the sequential work remains

- Amdahl's Law applies to problem *instances*

Parallelism seemingly has little potential

More On Amdahl's Law

- Amdahl's Law assumes a fixed problem instance: Fixed n , fixed input, perfect speedup
 - The algorithm can change to become more ||
 - Problem instances grow implying proportion of work that is sequential may be smaller %
 - ... Many, many realities including parallelism in 'sequential' execution imply analysis is simplistic
- *Amdahl is a fact; it's not a show-stopper*

Digress: Inherently Sequential

- As an artifact of P -completeness theory, we have the idea of *Inherently Sequential* -- computations not appreciably improved by parallelism

Circuit Value Problem:

Given a circuit α over Boolean inputs, values b_1, \dots, b_n and designated output value y , is the circuit true for y ?

- Probably not much of a limitation

Gustaffson's Law (another approach)

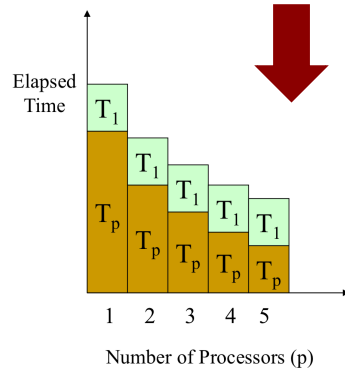
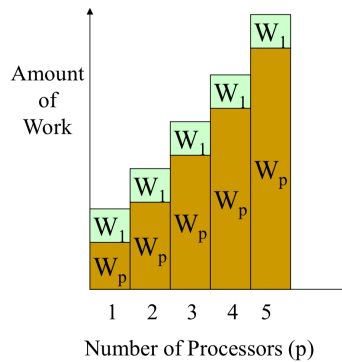
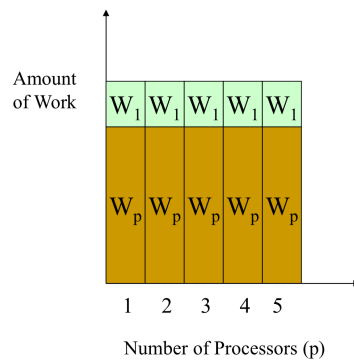
- Comes with the concept of **Scalable Computing** where problem sizes increase with the machine sizes
 - J. L. Gustafson, G. R. Montry, and R. E. Benner, "*Development of Parallel Methods for a 1024-Processor Hypercube*," SIAM Journal on Scientific and Statistical Computing, Vol. 9, No.4, 1988
 - John Gustafson, "*Reevaluation of Amdahl's Law*," Communications of the ACM, Vol. 31, No. 5, May 1988
- Large machines are not (only) to solve existing problems faster, they should solve otherwise unsolvable large problems
 - assuming problem size increases with the machine size

$$\begin{aligned} S'p &= \text{Uniprocessor Time of Solving } W' / \text{Parallel Time of Solving } W' = \\ &= \text{Uniprocessor Time of Solving } W' / \text{Uniprocessor Time of Solving } W = \\ &= W' / W \end{aligned}$$

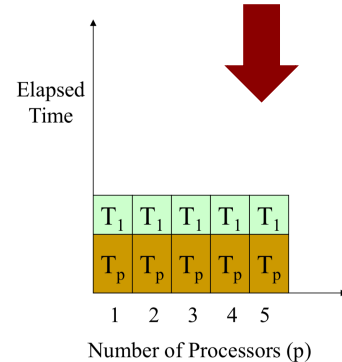
where the problem size is scaled from W to W'

Is the assumption more work within the same time T generally applicable?

Both Laws Together



Amdahl's view
(fixed problem size)



Gustaffson's view
(scalable problem size)

Amdahl (v.2):

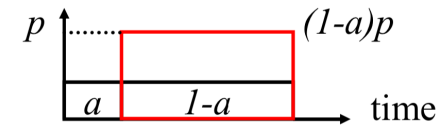
$$\alpha = \frac{t_s}{t_s + t_p}$$

$$T_p = \alpha \cdot T_{\text{base}} + (1-\alpha) \cdot T_{\text{base}} / p$$

where, α is the fraction of the serial code and p – the speedup factor of the parallel portion

$$T_p = (\alpha + (1-\alpha) / p) \cdot T_s$$

$$S_p = T_s / T_p = 1 / (\alpha + (1-\alpha) / p); \lim_{p \rightarrow \infty} = 1 / \alpha$$



Gustaffson:

$$S_p = \text{Work}(p) / \text{Work}(1) =$$

$$= (\alpha \cdot W + (1-\alpha) \cdot p \cdot W) / W = \alpha + (1-\alpha) \cdot p$$

linear speedup is assumed

With $\alpha = 0.1$ (10% serial code)
Amdahl's speedup is maximal **10**, while
Gustaffson claims **$0.1 + 0.9 \cdot p$**

Two kinds of performance

- **Latency** -- time required before a requested value is available
 - Latency, measured in seconds; called *transmit time* or *execution time* or just *time*
- **Throughput** -- amount of work completed in a given amount of time
 - Throughput, measured in “work”/sec, where “work” can be bits, instructions, jobs, etc.; also called *bandwidth* in communication

Both terms apply to computing and communications

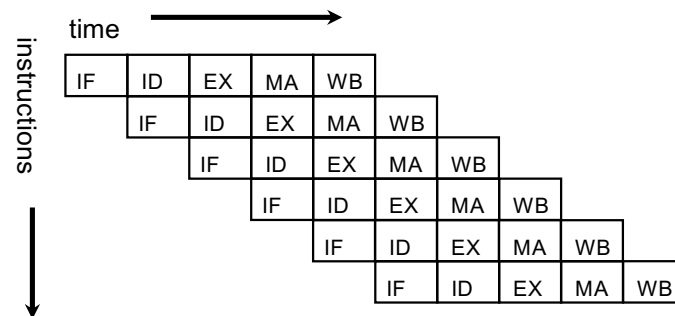
Latency

- Reducing latency (execution time) is a principal goal of parallelism
- There is upper limit on reducing latency
 - Speed of light, especially for bit transmissions
 - In networks, switching time (node latency)
 - (Clock rate) x (issue width), for instructions
 - Diminishing returns (overhead) for problem instances

Hitting the upper limit is rarely a worry

Throughput

- Throughput improvements are often easier to achieve by adding hardware
 - More wires improve bits/second
 - Use processors to run separate jobs
 - Pipelining is a powerful technique to execute more (serial) operations in unit time



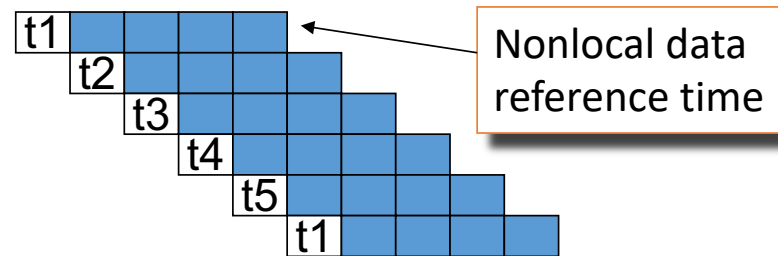
Better throughput often hyped as if better latency

Latency Hiding

- Reduce wait times by switching to work on different operation (multithreading)
 - Old idea, dating back to Multics
 - In parallel computing it's called *latency hiding*
- Idea most often used to lower impact of λ cost
 - Have many threads ready to go ...
 - Execute a thread until it makes nonlocal ref
 - Switch to next thread
 - When nonlocal ref is filled, add to ready list

Latency Hiding (Continued)

- Latency hiding requires ...
 - Consistently large supply of threads $\sim \lambda/e$
where e = average # cycles between nonlocal refs
 - Enough network throughput to have many requests in the air at once



- Latency hiding has been claimed to make shared memory feasible in the presence of large λ

There are difficulties

Latency Hiding (Continued)

- Challenges to supporting shared memory
 - Threads must be numerous, and the shorter the interval between nonlocal refs, the more
 - Running out of threads stalls the processor
 - Context switching to next thread has overhead
 - Many hardware contexts -- or --
 - Waste time storing and reloading context
 - Tension between latency hiding & caching
 - Shared data must still be protected somehow
 - Other technical issues

Performance Loss: Contention

- Contention -- the action of one processor interferes with another processor's actions -- is an elusive quantity
 - Lock contention: One processor's lock stops other processors from referencing; they must wait
 - Bus contention: Bus wires are in use by one processor's memory reference
 - Network contention: Wires are in use by one packet, blocking other packets
 - Bank contention: Multiple processors try to access different locations on one memory chip simultaneously

Contention is very time dependent, that is, variable

Performance Loss: Load Imbalance

- Load imbalance, work not evenly assigned to the processors, underutilizes parallelism
 - The assignment of *work*, not data, is key
 - Static assignments, being rigid, are more prone to imbalance
 - Because dynamic assignment carries overhead, the quantum of work must be large enough to amortize the overhead
 - With flexible allocations, load balance can be solved late in the design programming cycle

The Best Parallel Programs ...

- Performance is maximized if processors execute continuously on local data without interacting with other processors
 - To unify the ways in which processors could interact, we adopt the concept of dependence
 - A *dependence* is an ordering relationship between two computations
 - Dependences are usually induced by read/write
 - Dependences that cross process boundaries induce a need to synchronize the threads

Dependences are well-studied in **compilers**

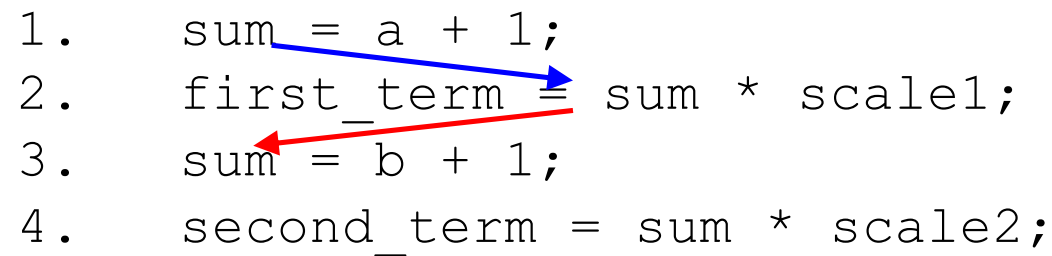
Dependences

- Dependences are orderings that must be maintained to guarantee correctness
 - Flow-dependence: read after write (RaW) True
 - Anti-dependence: write after read (WaR) False
 - Output-dependence: write after write (WaW) False
- True dependences affect correctness
- False dependences arise from memory reuse

Example of Dependences

- Both true and false dependences

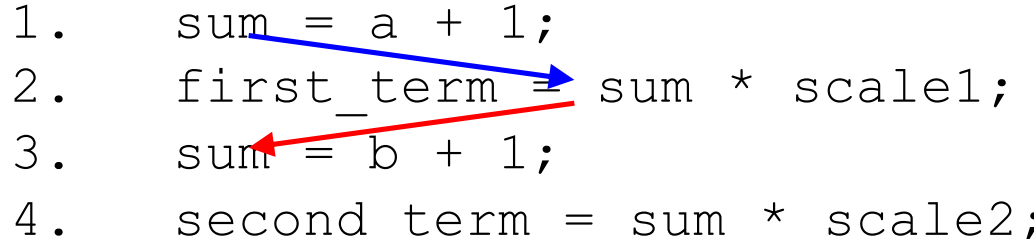
```
1.  sum = a + 1;  
2.  first_term = sum * scale1;  
3.  sum = b + 1;  
4.  second_term = sum * scale2;
```



Example of Dependences

- Both **true** and **false** dependences

```
1.  sum = a + 1;  
2.  first_term = sum * scale1;  
3.  sum = b + 1;  
4.  second_term = sum * scale2;
```



- Flow-dependence** read after write; must be preserved for correctness
- Anti-dependence** write after read; can be eliminated with additional memory

Removing Anti-dependence

- Change variable names

```
1.  sum = a + 1;  
2.  first_term = sum * scale1;  
3.  sum = b + 1;  
4.  second_term = sum * scale2;
```



```
1.  first_sum = a + 1;  
2.  first_term = first_sum * scale1;  
3.  second_sum = b + 1;  
4.  second_term = second_sum * scale2;
```

Granularity

- Granularity is used in many contexts...here *granularity* is the amount of work between cross-processor dependences
 - Important because interactions are usually costly
 - Generally, larger grain is better
 - + fewer interactions, more local work
 - can lead to load imbalance
 - Batching is an effective way to increase grain
 - (aggregate work)

Locality

- The CTA motivates us to maximize locality
 - Caching is the traditional way to exploit locality ... but it doesn't translate directly to ||ism
 - Redesigning algorithms for parallel execution often means repartitioning to increase locality
 - Locality often requires redundant storage and redundant computation, but in limited quantities they help

Measuring Performance

- Execution time ... what's time?
 - 'Wall clock' time
 - Processor execution time
 - System time
- Paging and caching can affect time
 - Cold start vs warm start
- Conflicts w/ other users/system components
- Measure kernel or whole program

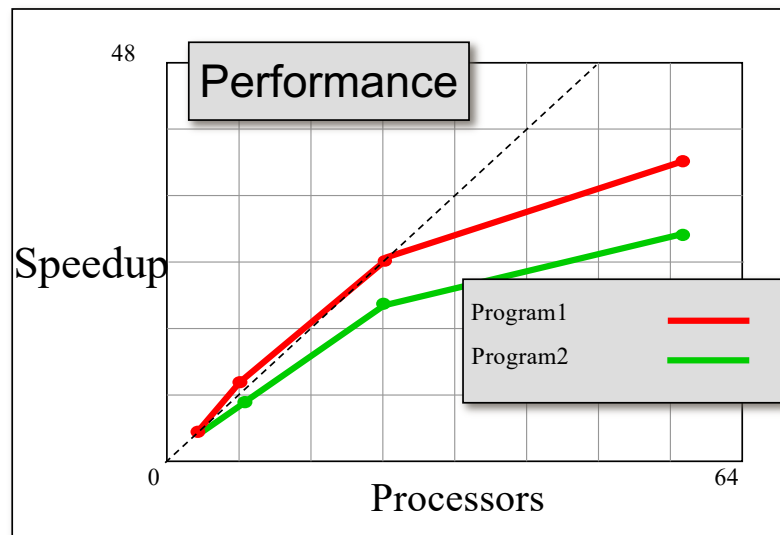
FLOPS

- Floating Point Operations Per Second is a common measurement for scientific programs
 - Even scientific computations use many integers
 - Results can often be influenced by small, low-level tweaks having little generality: multiply/add
 - Translates poorly across machines because it is hardware dependent
 - Limited application ... but it won't go away!

In 2007 Intel made an experimental multi-core (80) POLARIS
1 to 2 TFLOPS for 100 to 200W. Dedicated programming model (Ct) but ...

Speedup and Efficiency

- Speedup is the factor of improvement for P processors: T_S / T_P



$$\text{Efficiency} = \text{Speedup} / P$$

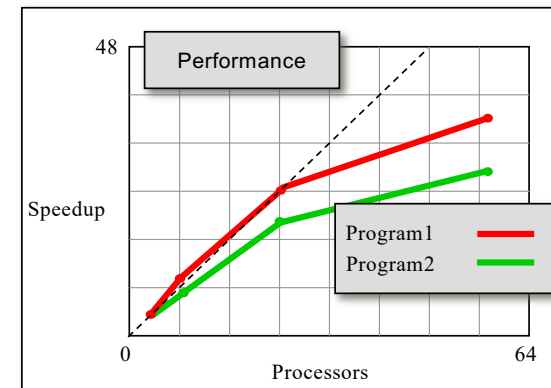
Issues with Speedup, Efficiency

- Speedup is best applied when hardware is constant, or for family within a generation
 - Need to have computation, communication in same ratio
 - Great sensitivity to the T_S value
 - T_S should be the time of the best sequential program on 1 processor of the $||$ -machine
 - $T_{P=1} \neq T_S$ Measures **relative speedup**

Relative speedup is often important
but it must be labeled as such

Scaled versus Fixed Speedup

- As P increases, the amount of work per processor diminishes, often below the amount needed to amortize costs
- Speedup curves bend down
- Scaled speedup keeps the work per processor constant, allowing other effects to be seen
- Both are important



If not explicitly stated,
speedup is fixed speedup

Strong Scaling vs Weak Scaling (intermezzo)

- Amdahl's Law — Strong Scaling
 - Fixed Problem Size
 - How much does parallelism reduce the execution time of a problem?

Main question: How much faster am I with P processors for fixed problem size?

- Gustafson's Law — Weak Scaling
 - Fixed Execution Time
 - How much longer does it take for the problem without parallelism?

Main question: How well does the parallel fraction scale among P processors?

I wrote a shared memory code. How well does my code run in parallel? (strong or weak?)

“Cooking” The Speedup Numbers

- The sequential computation should not be charged for **any** || costs
... consider



- If referencing memory in other processors takes time (λ) and data is distributed, then one processor solving the problem results in greater t compared to true sequential

This complicates methodology for large problems

What If Problem Doesn't Fit?

- Cases arise when sequential doesn't fit in 1 processor of parallel machine
- Best solution is relative speed-up
 - Measure $T_{\pi=\text{smallest possible}}$
 - Measure T_P , compute T_{π}/T_P as having P/π potential improvement

We Will Return ...

- Many issues regarding parallelism have been introduced, but they require further discussion ... we will return to them when they are relevant

Summary of Key Points

- Amdahl's Law is a fact but it doesn't impede us much
- Inherently sequential problems (probably) exist, but they don't impede us either
- Latency hiding could hide the impact of λ with sufficiently many threads and much (interconnection) bandwidth
- Impediments to parallel speedup are numerous: overhead, contention, inherently sequential code, waiting time, etc.

Review Key Points (continued)

- Concerns while parallel programming are also numerous: locality, granularity, dependences (both true and false), load balance, etc.
- Happily: Parallel and sequential computers are different: More hardware means more fast memory (cache, RAM), implying the possibility of superlinear speedup
- Measuring improvement is complicated

Copyright

