

# Parallel Programming

Overview of Parallel and Distributed Programming

Professor Yi-Ping You (游逸平)

Department of Computer Science

<http://www.cs.nctu.edu.tw/~ypyou/>



# Outline

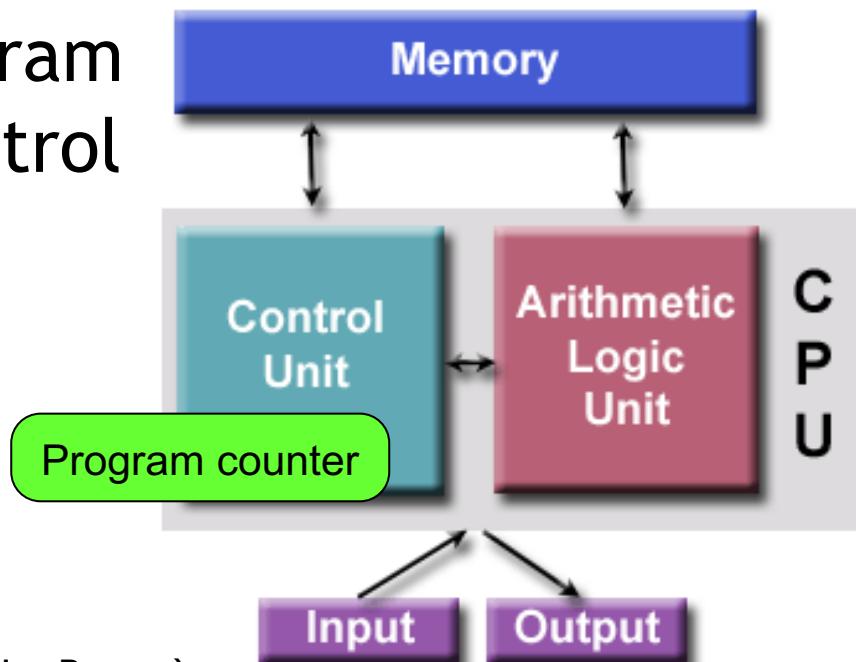
---

- The What and Why of Parallel Computing
- The Why and How of Parallel Programming
- Parallel Programming Models
- Performance Evaluation



# Sequential Computing

- von Neumann arch. with Program Counter (PC) dictates sequential execution
- Traditional programming thus follows a **single thread of control**
  - ✿ The sequence of program points reached as control flows through the program

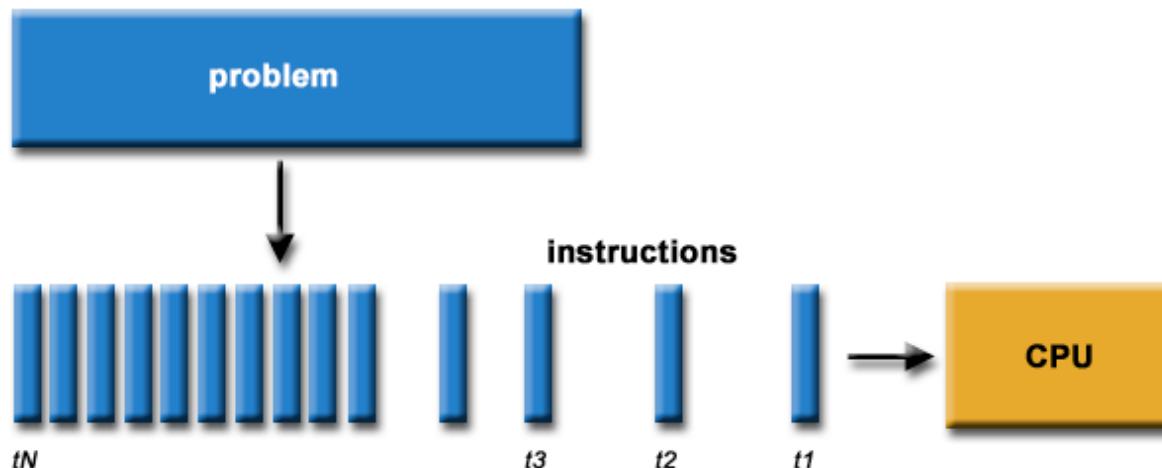


(Introduction to Parallel Computing, Blaise Barney)



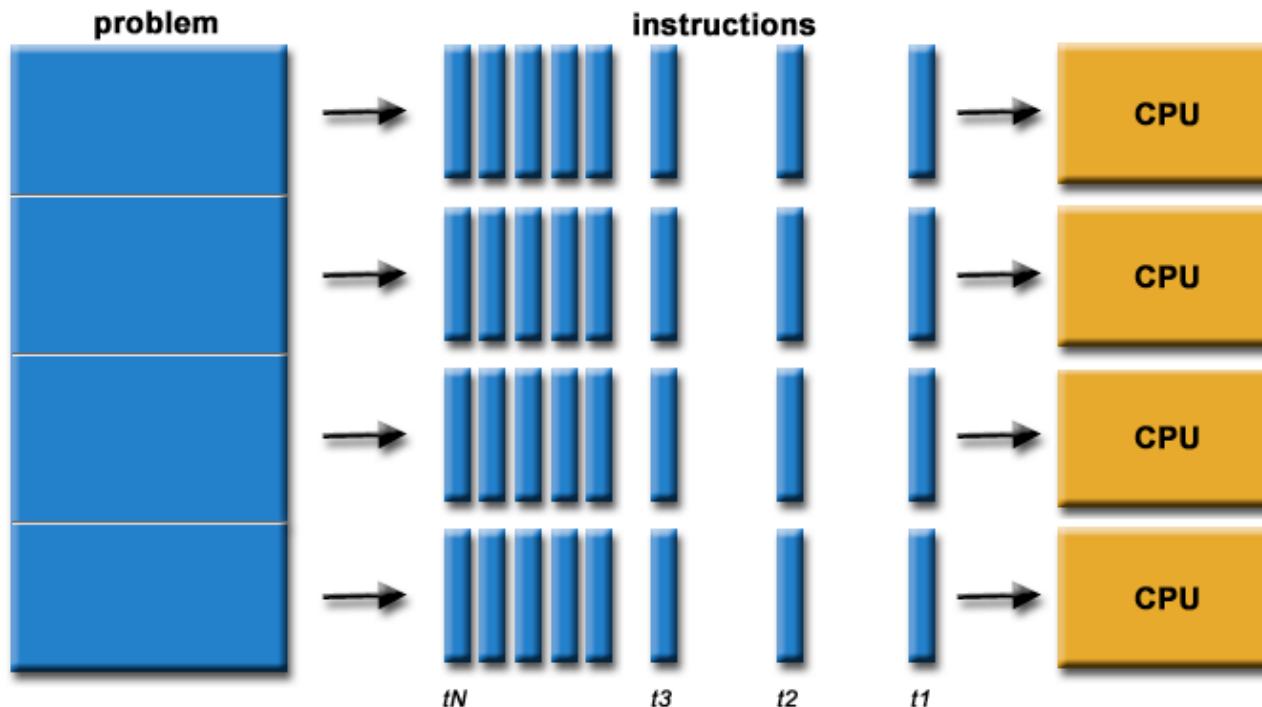
# Sequential Computing (Cont'd)

- Traditionally, software has been written for serial computing:
  - To be run on a single computer having a single CPU
  - A problem is broken into a discrete series of instructions
  - Instructions are executed one after another
  - Only one instruction may execute at any moment in time



# What is Parallel Computing?

- In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem



# Concurrent, Parallel, and Distributed Computing

---

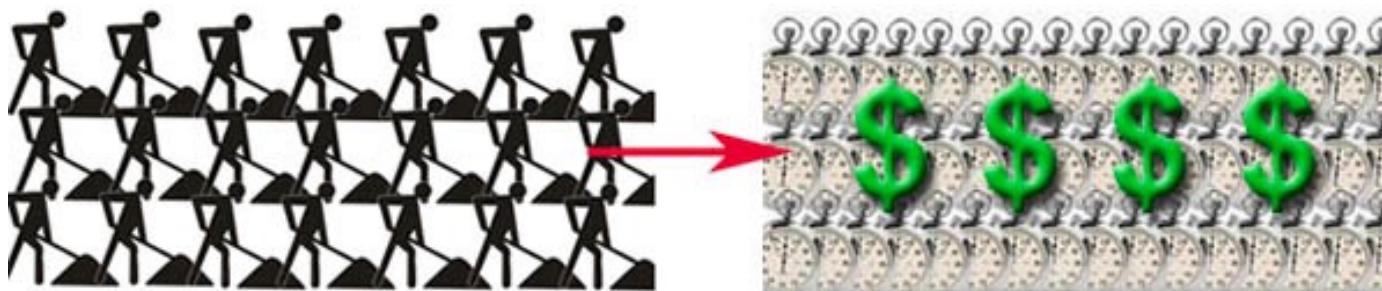
- Although there isn't complete agreement on the distinction between these terms, many authors make the following distinctions:
  - ✿ Concurrent computing: A program is one in which multiple tasks can be *in progress* at any instant
  - ✿ Parallel computing: A program is one in which multiple tasks *cooperate closely* to solve a problem
  - ✿ Distributed computing: A program may need to cooperate with other programs to solve a problem



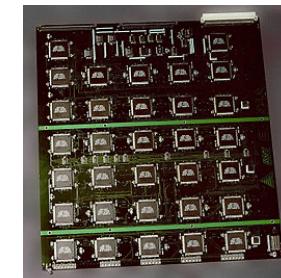
# Why Use Parallel Computing? (1/3)

## ■ Save time and/or money

- In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings
  - ◆ Parallel computers can be built from cheap, commodity components



- Time to brute force a 56-bit DES key has evolved from
  - ◆ 1998, 56 hours, \$250,000
  - ◆ 2006, 6.4 hours, \$10,000



# Why Use Parallel Computing? (2/3)

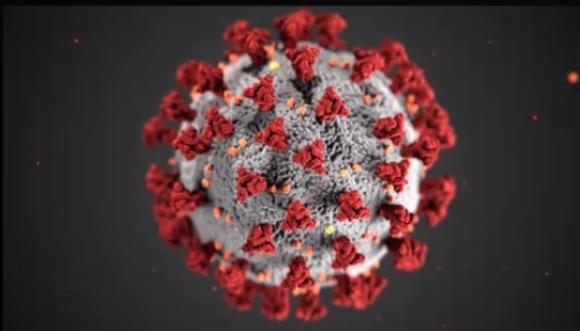
- Solve larger problems (applications demand it!)
  - Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory
    - ◆ Largest direct numerical solutions have evolved from meshes of 1024<sup>3</sup> to 8096<sup>3</sup> in the last ten years!
    - ◆ E.g., climate modeling, drug discovery, energy research, data analysis
  - Big data drivers



# COVID-19 Drug Development

## AMD'S COVID-19 HPC FUND

- Provides research institutions with computing resources to accelerate medical research on COVID-19 and other diseases
- Examples include:
  - NYU: Discovery of drugs that may be therapeutic for COVID-19, retrieval of relevant research from biomedical literature, and analysis of medical images
  - Rice: Exploration of how the coronavirus's surface proteins facilitate entrance to human cells and how to screen thousands of drug molecules to identify the best candidates for clinical tests
  - LLNL: Discovery of potential antibodies and anti-viral compounds for COVID-19 using molecular dynamics simulations and machine learning



Coronavirus



Corona Supercomputer



# Big Data Phenomenon

- “Data are becoming the new raw material of business: an economic input almost on a par with capital and labor.”  
—The Economist, 2010
- “Information will be the oil of the 21st century”  
—Gartner, 2010

**1.8ZB** in 2011

2 Days > the dawn of civilization to 2003



**750 Million**

Photos uploaded to Facebook in 2 days



**966PB**

Stored in US manufacturing (2009)



**209 Billion**

RFID tags sale in 2021:  
from 12 million in 2011



**200+TB**

A boy's 240'000 hours by  
a MIT Media Lab geek



**200PB**

Storage of a Smart City  
project in China



**\$800B**

in personal location data  
within 10 years



**\$300B** /year

US healthcare saving  
from Big Data



**\$32+B**

Acquisitions by 4 big  
players since 2010



Source: IDF2012



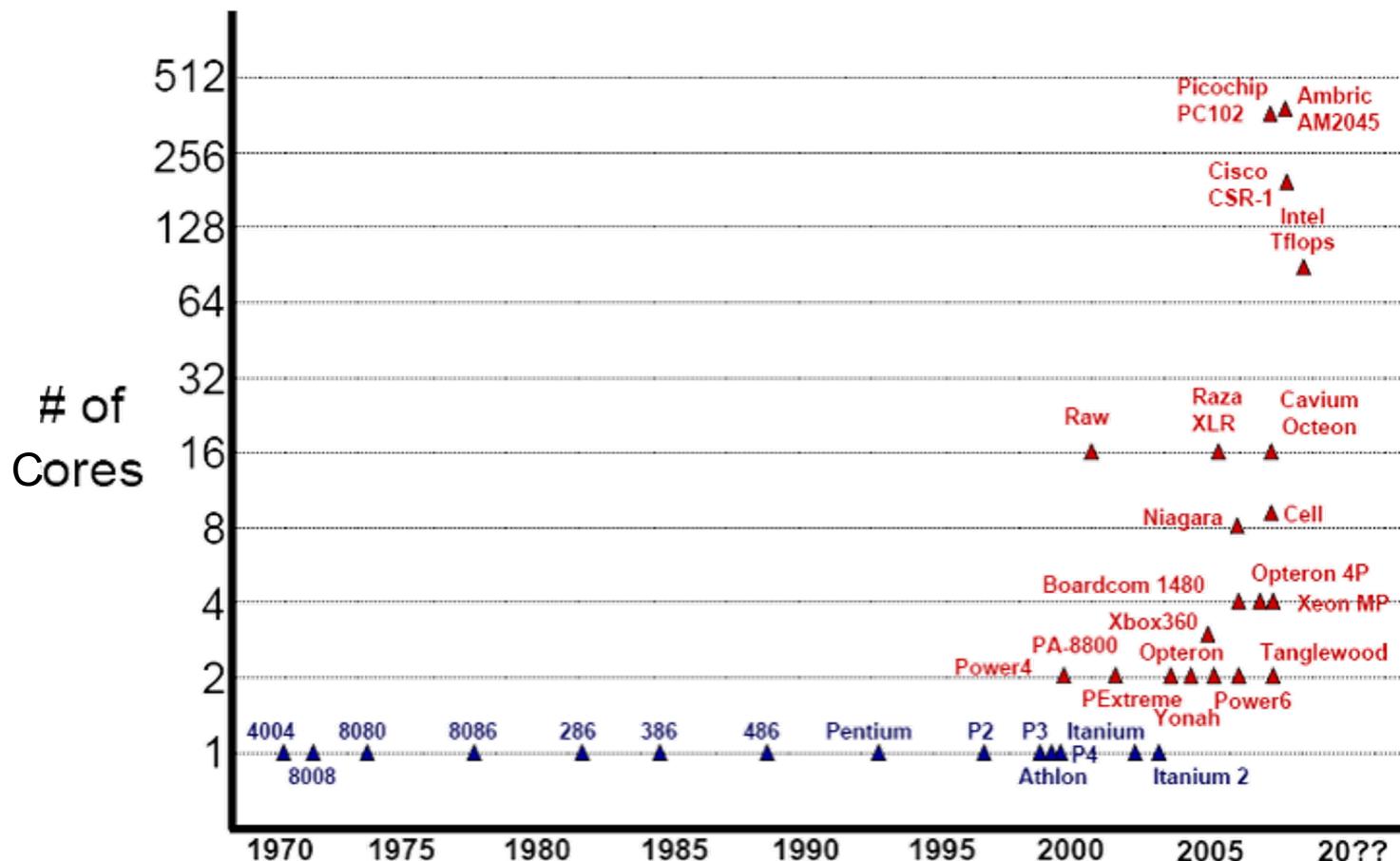
# Why Use Parallel Computing? (3/3)

---

- Architectures demand it!
  - ✿ The free lunch is over
    - ◆ Clock speed increase is no longer an option
  - ✿ Multiple/many cores are future trend
  - ✿ Simply adding more processors will not magically improve the performance of the vast majority of **serial** programs



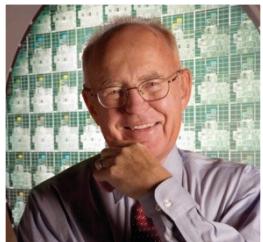
# Multicore Trends



(Source) MIT Course 6.189, "Multi-core Programming Primer: Learn and Compete in Programming the PLAYSTATION®3 Cell Processor."

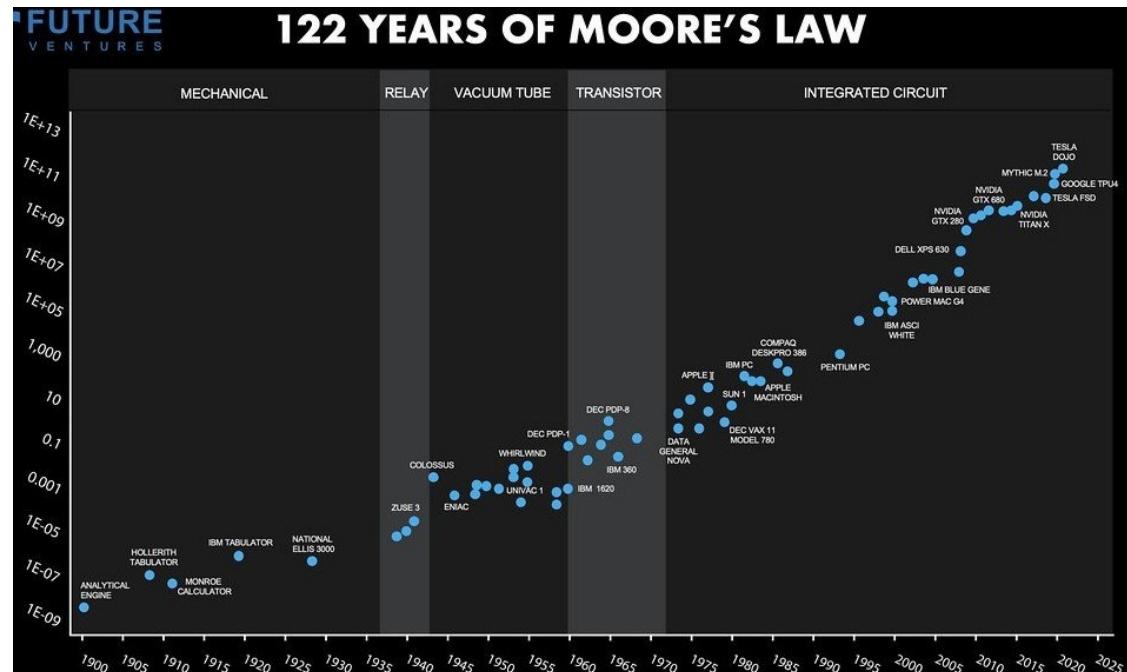


# Moore's Law



- Describes a trend in the history of computer hardware that the number of transistors can be inexpensively placed on an integrated circuit is increasing exponentially

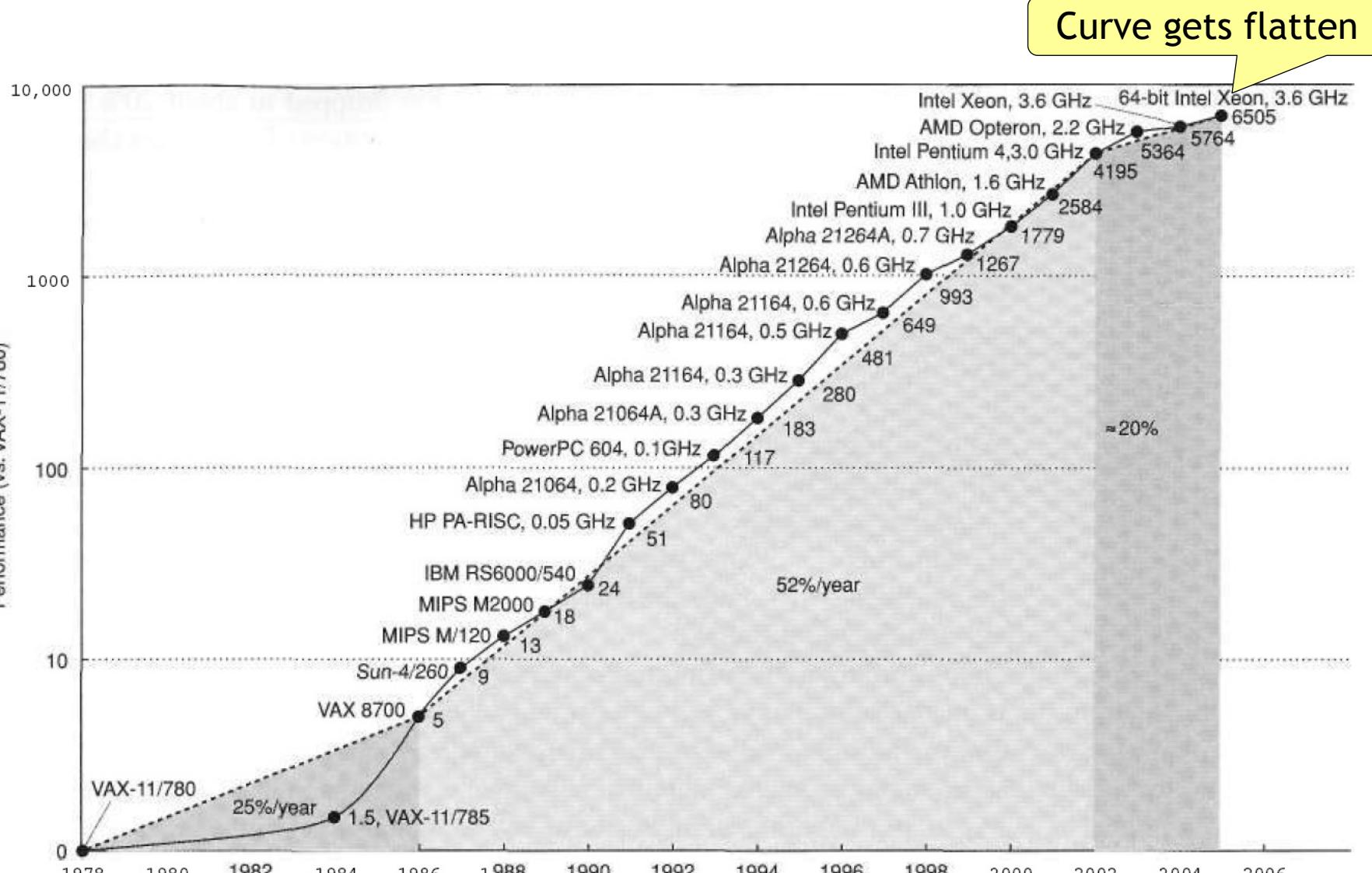
- Doubling every 18 month (Moore's original predication)
- Doubling approximately every two years (actual growth of processors)



(Source) <https://www.unite.ai/moores-law/>



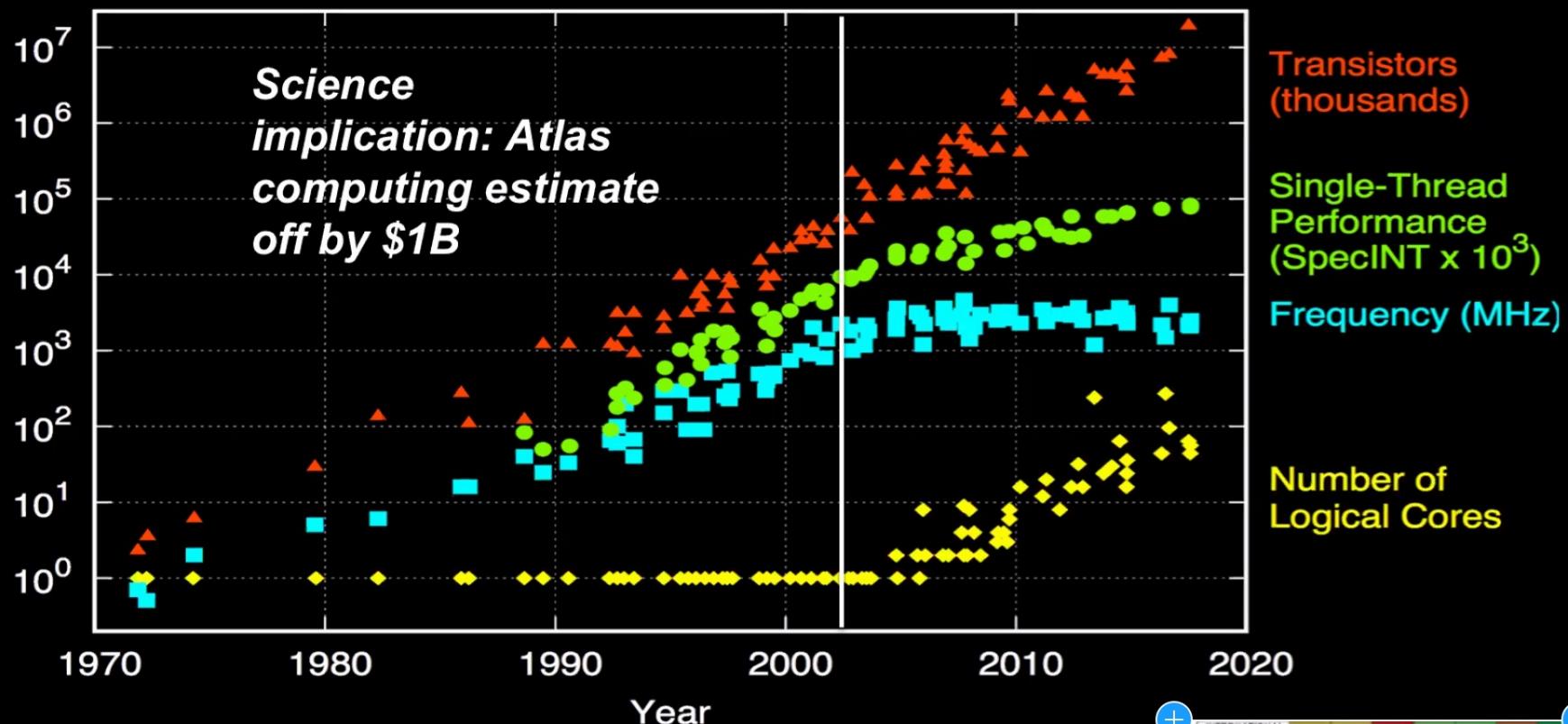
# Semiconductor Trends



(Source) Hennessy & Patterson, Computer Architecture: A Quantitative Approach, 4th edition, Oct., 2006



# Dennard Scaling is Dead; Moore's Law Will Follow



M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten, and K. Rupp

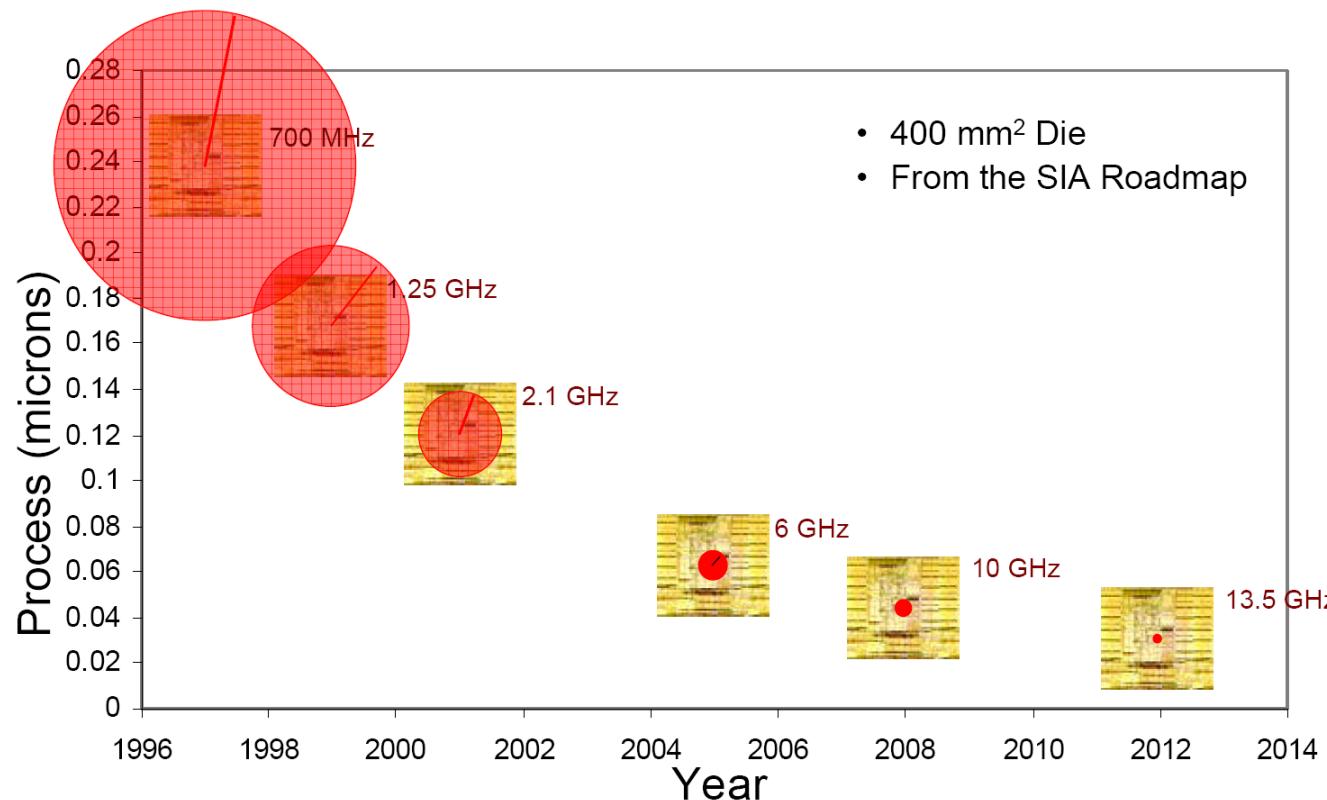


Source: Kathy Yelick's keynote speech at ICPP '20 and ICPP '23



# Challenges: Wire Delay becomes issue when clock rate raised

- The signal delay for a wire increases in proportion to the product of its resistance and capacitance ( $R*C$ )
  - Feature size shrinks → shorter wires → higher resistance and capacitance per unit length

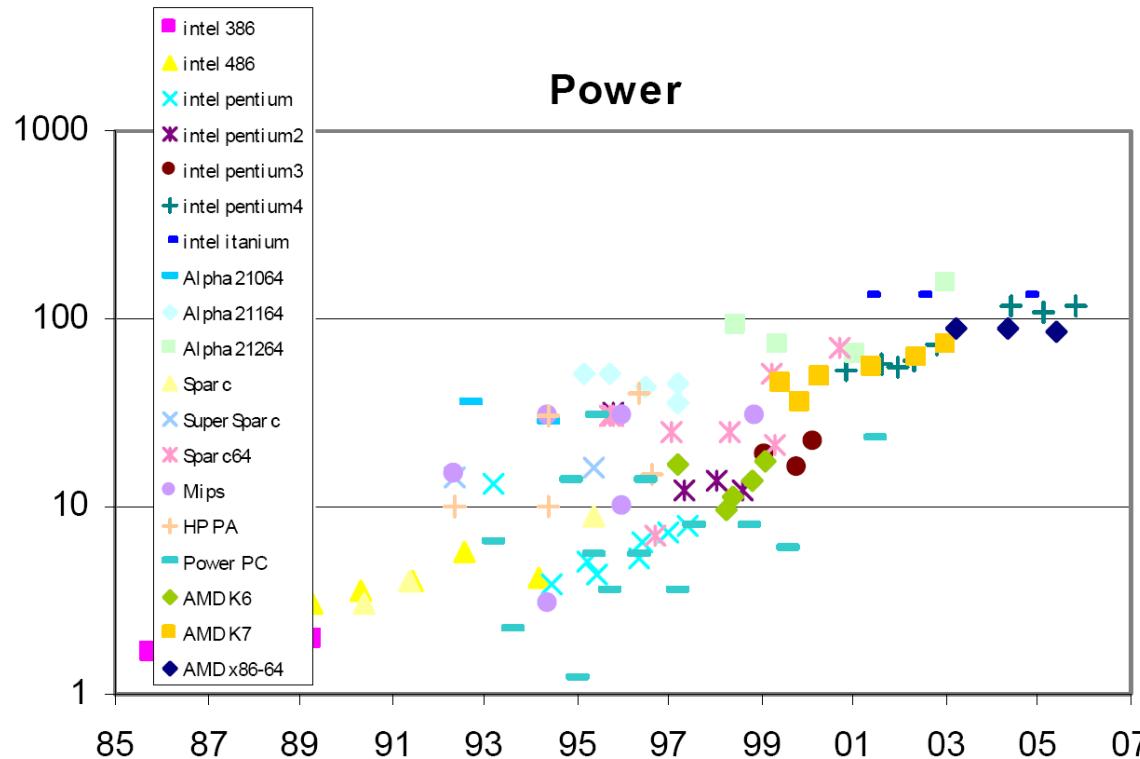


(Source) MIT Course 6.189, "Multi-core Programming Primer: Learn and Compete in Programming the PLAYSTATION®3 Cell Processor."



# Challenges: Power Consumption

- Power consumption becomes a problem:
  - ✿ Operation energy of transistors (cost, environment)
  - ✿ Heat dissipation reaching its limitation (cooling, space, cost)

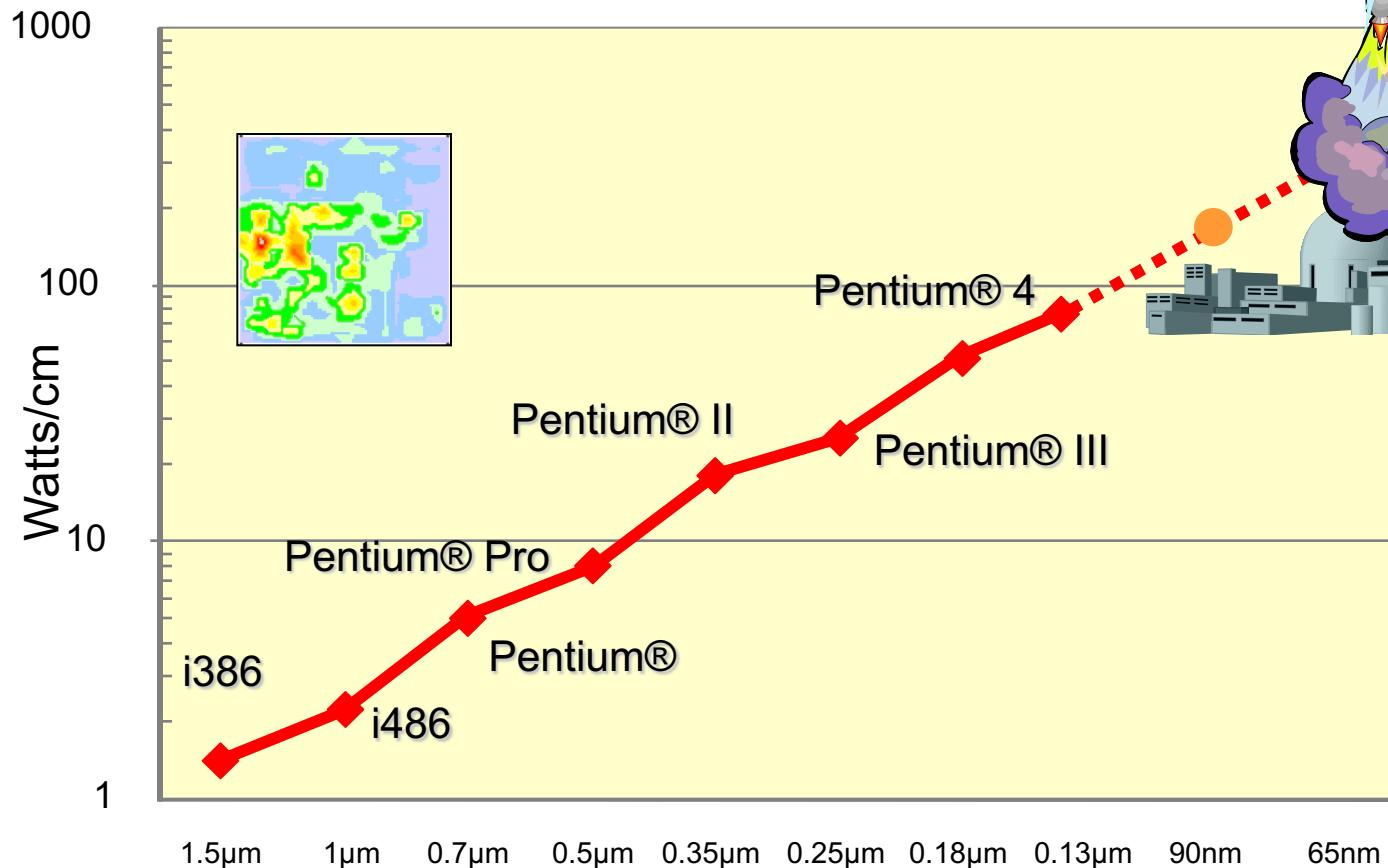


(Source) MIT Course 6.189, "Multi-core Programming Primer: Learn and Compete in Programming the PLAYSTATION®3 Cell Processor."



# Power Density Trend

Power density doubles every 4 years



Source: Intel



# Outline

---

- The What and Why of Parallel Computing
- **The Why and How of Parallel Programming**
- Parallel Programming Models
- Performance Evaluation



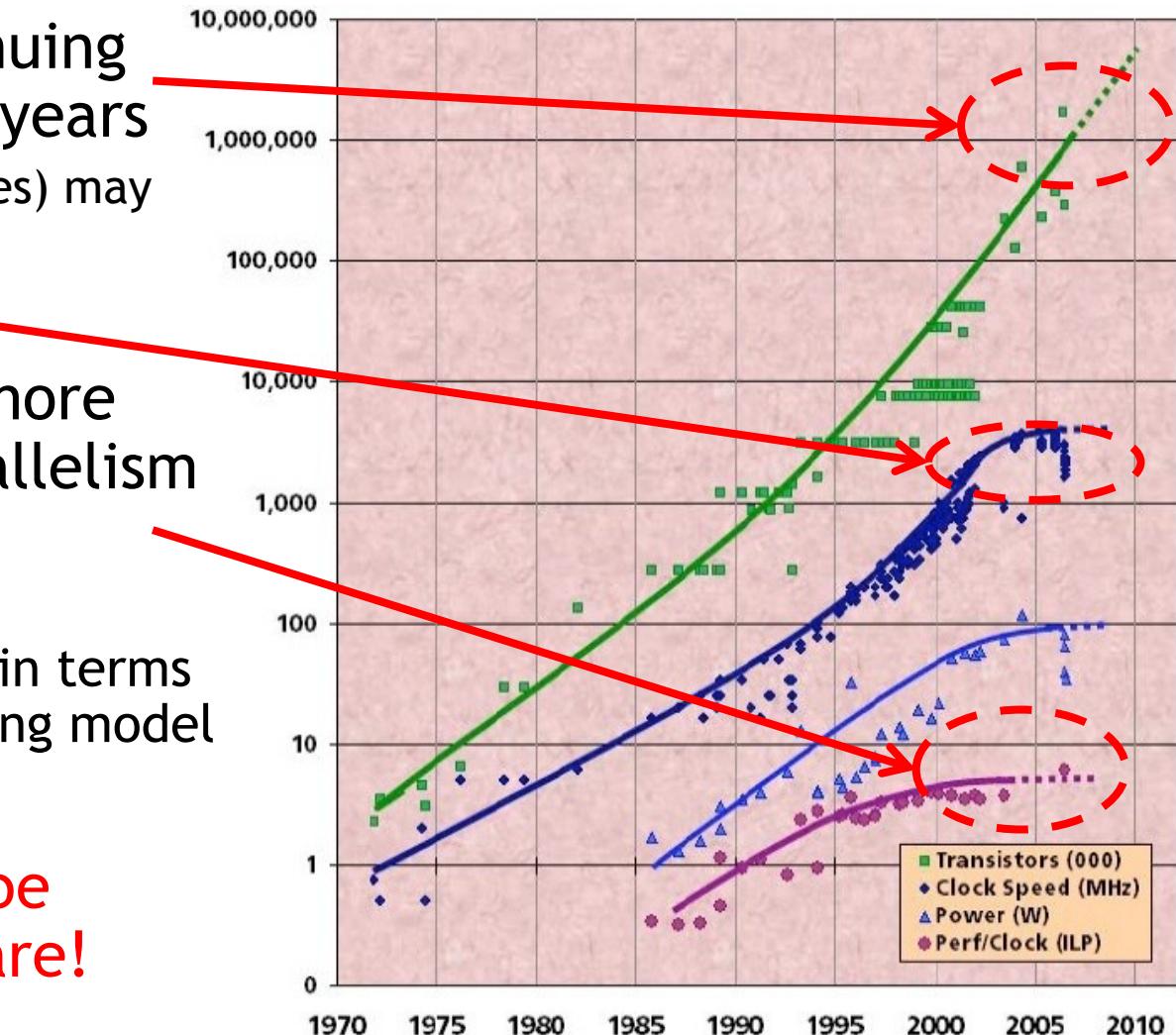
# Why We Need to Write Parallel Programs?

- Most programs that have been written for single-core systems cannot exploit the presence of multiple cores
- The bad news is that researchers have had very limited success writing programs that convert serial programs in languages such as C and C++ into parallel programs



# Limiting Forces: Clock Speed and ILP

- Chip density is continuing increase ~2x every 2 years
  - # processors/chip (cores) may double instead
- Clock speed is not
- There is little or no more instruction-level parallelism (ILP) to be found
  - Can no longer allow programmer to think in terms of a serial programming model
- Conclusion:
  - Parallelism must be exposed to software!



Source: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond)



# Parallelizing A Program: An Example

---

- As an example, suppose that we need to compute  $n$  values and add them together
- We know that this can be done with the following serial code:

```
sum = 0;  
for (i = 0; i < n; i++) {  
    x = Compute_next_value(. . .);  
    sum += x;  
}
```



# Partitioning Step

- Now suppose we also have  $p$  cores and  $p$  is much smaller than  $n$ . Then each core can form a partial sum of approximately  $n/p$  values:

```
my_sum = 0;  
my_first_i = . . . ;  
my_last_i = . . . ;  
for (my_i = my_first_i; my_i < my_last_i; my_i++) {  
    my_x = Compute_next_value(. . .);  
    my_sum += my_x;  
}
```

```
my_n = n/p;  
my_first_i = my_n * my_rank;  
my_last_i = my_first_i + my_n;
```

- Here the prefix *my* indicates that each core is using its own, private variables, and each core can execute this block of code independently of the other cores



# An Example

- For example, if there are eight cores,  $n = 24$ , and the 24 calls to Compute next value return the values

1,4,3, 9,2,8, 5,1,1, 6,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9,

- Then the values stored in my sum might be

| Core   | 0 | 1  | 2 | 3  | 4 | 5  | 6  | 7  |
|--------|---|----|---|----|---|----|----|----|
| my_sum | 8 | 19 | 7 | 15 | 7 | 13 | 12 | 14 |

- When the cores are done computing their values of *my\_sum*, they can form a global sum by sending their results to a designated “master” core, which can add their results



# Aggregation Step

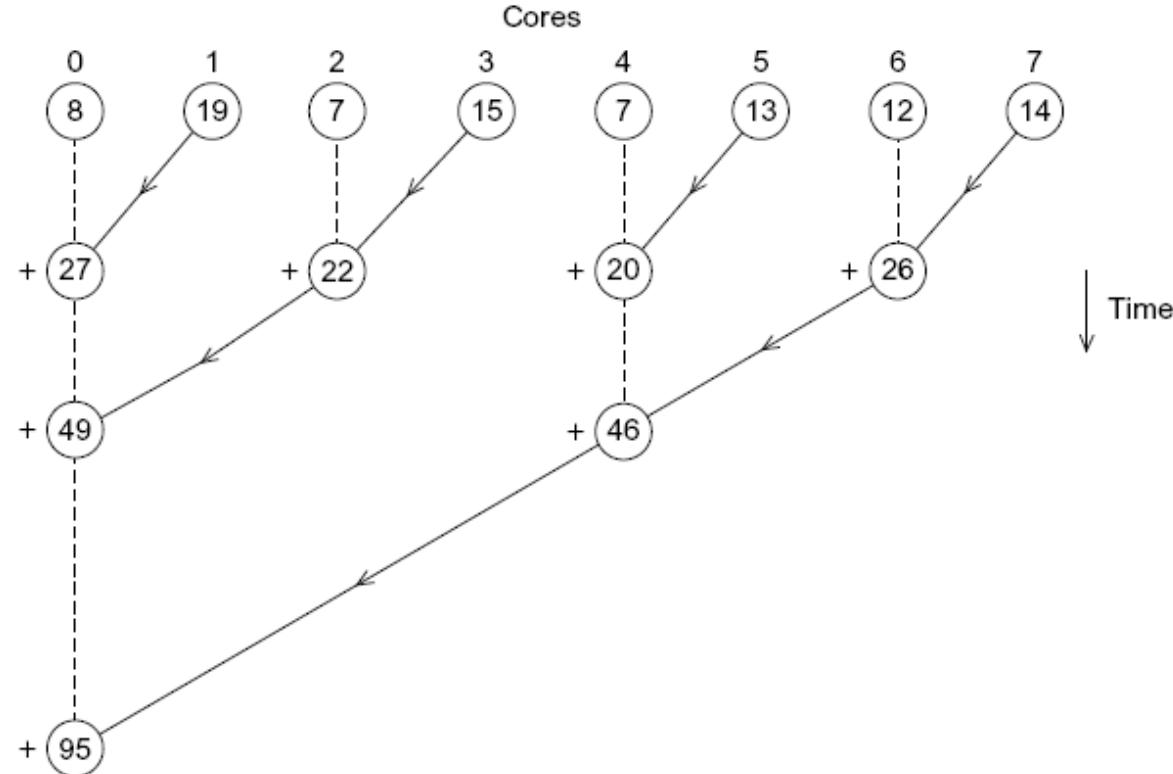
- If the master core is core 0, it would add the values  $8+19+7+15+7+13+12+14 = 95$

```
if (I'm the master core) {  
    sum = my_x;  
    for each core other than myself {  
        receive value from core;  
        sum += value;  
    }  
} else {  
    send my_x to the master;  
}
```



# A Better Aggregation

- Instead of making the master core do all the work of computing the final sum, we can pair the cores so that while core 0 adds in the result of core 1, core 2 can add in the result of core 3, core 4 can add in the result of core 5 and so on



# Comparison between Two Aggregation Methods

---

- With eight cores, the master will carry out **seven** receives and adds using the first method, while with the second method it will only carry out **three**
- The difference becomes much more dramatic with large numbers of cores
  - ✿ With 1000 cores, the first method will require **999** receives and adds, while the second will only require **10**, an improvement of almost a factor of **100!**



# Writing Parallel Software to Exploit Parallelism

---

- The point here is that it's unlikely that a translation program would “discover” the second global sum
- We cannot simply continue to write serial programs
- We must write parallel programs that exploit the power of multiple processors



# How Do We Write Parallel Programs?

---

- There are two widely used approaches:
  - 1) Task-parallelism
    - We **partition the various tasks** carried out in solving the problem among the cores
  - 2) Data-parallelism
    - We **partition the data** used in solving the problem among the cores, and each core carries out **more or less similar operations** on its part of the data
- The theme of the course focuses on data parallelism, which mostly exists within loops



# How Do We Write Parallel Programs?

---

- Currently, the most powerful parallel programs are written using *explicit* parallel constructs
- They are written using extensions to languages such as C and C++
  - ◆ These programs include explicit instructions for parallelism



# Coordination

---

- In both global sum examples, the coordination involves three types:
  - 1) Communication
    - One or more cores send their current partial sums to another core
  - 2) Load Balancing
    - We want the cores all to be assigned roughly the same number of values to compute
  - 3) Synchronization
    - Sometimes we want that each core will wait until all the cores have entered a given point



# Outline

---

- The What and Why of Parallel Computing
- The Why and How of Parallel Programming
- **Parallel Programming Models**
- Performance Evaluation



# Parallel Programming Models

---

- Programming model is made up of the languages and libraries that create an abstract view of the machine
- Control
  - How is parallelism **created**?
  - What **orderings** exist between operations?
- Data
  - What data are **private** vs. **shared**?
  - How is logically shared data accessed or **communicated**?
- Synchronization
  - What operations can be used to coordinate parallelism?
  - What are the **atomic** (indivisible) operations?



# Simple Example of Data Parallelism

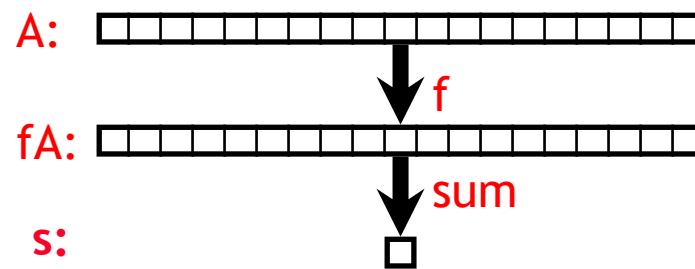
- Consider applying a function  $f$  to the elements of an array  $A$  and then computing its sum:

$$\sum_{i=0}^{n-1} f(A[i])$$

- Questions:

- Where does  $A$  live? All in single memory? Partitioned?
- What work will be done by each processors?
- They need to coordinate to get a single result, how?

$A$  = array of all data  
 $fA = f(A)$   
 $s = \text{sum}(fA)$



# Outline

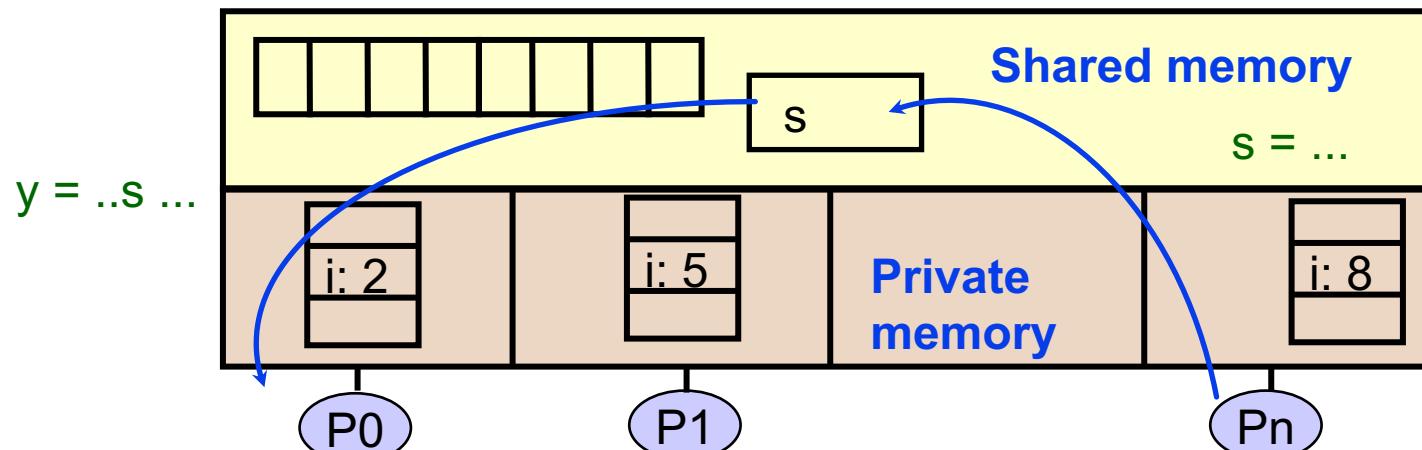
---

- The What and Why of Parallel Computing
- The Why and How of Parallel Programming
- Parallel Programming Models
  - ✿ Shared-memory model
  - ✿ Distributed-memory model
  - ✿ Data-parallel model
- Performance Evaluation



# Programming Model 1: Shared Memory

- Program is a collection of threads of control
  - ⊕ Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap
  - ⊕ Threads communicate implicitly by writing and reading shared variables
  - ⊕ Threads coordinate by synchronizing on shared variables



# Simple Example of Data Parallelism

## ■ Shared memory strategy:

- ⊕ small number  $p \ll n = \text{size}(A)$  processors
- ⊕ attached to single memory

## ■ Parallel decomposition:

- ⊕ Each evaluation and each partial sum is a task

## ■ Assign $n/p$ numbers to each of $p$ procs

- ⊕ Each computes independent “private” results and partial sum
- ⊕ Collect the  $p$  partial sums and compute a global sum

## Two Classes of Data:

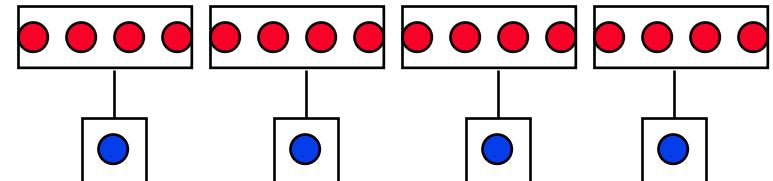
### ■ Logically Shared

- ⊕ The original  $n$  numbers, the global sum

### ■ Logically Private

- ⊕ The individual function evaluations
- ⊕ What about the individual partial sums?

$$\sum_{i=0}^{n-1} f(A[i])$$



# Shared Memory “Code” for Computing a Sum

```
fork(sum,a[0:n/2-1]);  
sum(a[n/2,n-1]);
```

```
static int s = 0;
```

Thread 1

```
for i = 0, n/2-1  
s = s + f(A[i])
```

Thread 2

```
for i = n/2, n-1  
s = s + f(A[i])
```

- What is the problem with this program?
- A race condition or data race occurs when:
  - ✿ Two processors (or two threads) access the same variable, and at least one does a write
  - ✿ The accesses are concurrent (not synchronized) so they could happen simultaneously



# An Example of Race Condition

A = 

|   |   |
|---|---|
| 3 | 5 |
|---|---|

 $f(x) = x^2$

static int s = 0;

## Thread 1

...  
compute  $f([A[i]])$  and put in reg0 9  
reg1 = s 0  
reg1 = reg1 + reg0 9  
s = reg1 9  
...

## Thread 2

...  
compute  $f([A[i]])$  and put in reg0 25  
reg1 = s 0  
reg1 = reg1 + reg0 25  
s = reg1 25  
...

- Assume A = [3, 5],  $f(x) = x^2$ , and s=0 initially
- For this program to work, s should be  $3^2 + 5^2 = 34$  at the end
  - but it may be 34, 9, or 25
- The atomic operations are reads and writes
  - Never see  $\frac{1}{2}$  of one number, but  $+=$  operation is not atomic
  - All computations happen in (private) register



# Improved Code for Computing a Sum

```
static int s = 0;  
static lock lk;
```

Why not do lock  
inside the loop?

## Thread 1

```
local_s1= 0  
for i = 0, n/2-1  
    local_s1 = local_s1 + f(A[i])  
    lock(lk);  
    s = s + local_s1  
    unlock(lk);
```

## Thread 2

```
local_s2 = 0  
for i = n/2, n-1  
    local_s2= local_s2 + f(A[i])  
    lock(lk);  
    s = s +local_s2  
    unlock(lk);
```

- Since addition is associative, it's OK to rearrange order
- Most computation is on private variables
  - Sharing frequency is also reduced, which might improve speed
  - But there is still a race condition on the update of shared s
  - The race condition can be fixed by adding locks (only one thread can hold a lock at a time; others wait for it)



# POSIX Threads

---

- POSIX: Portable Operating System Interface for UNIX
  - ◆ Interface to Operating System utilities
- Pthreads: The POSIX threading interface
  - ◆ System calls to create and synchronize threads
  - ◆ Should be relatively uniform across UNIX-like OS platforms
  - ◆ Originally IEEE POSIX 1003.1c
- Pthreads contain support for
  - ◆ Creating parallelism
  - ◆ Synchronizing
  - ◆ No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread
    - ◆ Only for static variables and HEAP! Stacks not shared



# Pthreads Example: Hello World

---

```
void* SayHello(void *foo) {  
    printf( "Hello, world!\n" );  
    return NULL;  
}  
  
int main() {  
    pthread_t threads[16];  
    int tn;  
    for(tn=0; tn<16; tn++) {  
        pthread_create(&threads[tn], NULL, SayHello, NULL);  
    }  
    for(tn=0; tn<16 ; tn++) {  
        pthread_join(threads[tn], NULL);  
    }  
    return 0;  
}
```



# Shared Data and Threads

---

- Variables declared outside of main are shared
- Objects allocated on the heap may be shared (if pointer is passed)
- Variables on the stack are private: passing pointer to these around to other threads can cause problems
- Often done by creating a large “thread data” struct, which is passed into all threads as argument

```
char *message = "Hello World!\n";
pthread_create(&thread1, NULL,
                print_fun, (void*) message);
```



# OpenMP

C\$OMP FLUSH

#pragma omp critical

## OpenMP: An API for Writing Multithreaded Applications



- A set of compiler directives and library routines for parallel application programmers
- Makes writing multi-threaded applications in Fortran, C and C++ as easy as we can make it
- Standardizes last 20 years of SMP practice

C\$OMP PARALLEL COPYIN(/blk/)

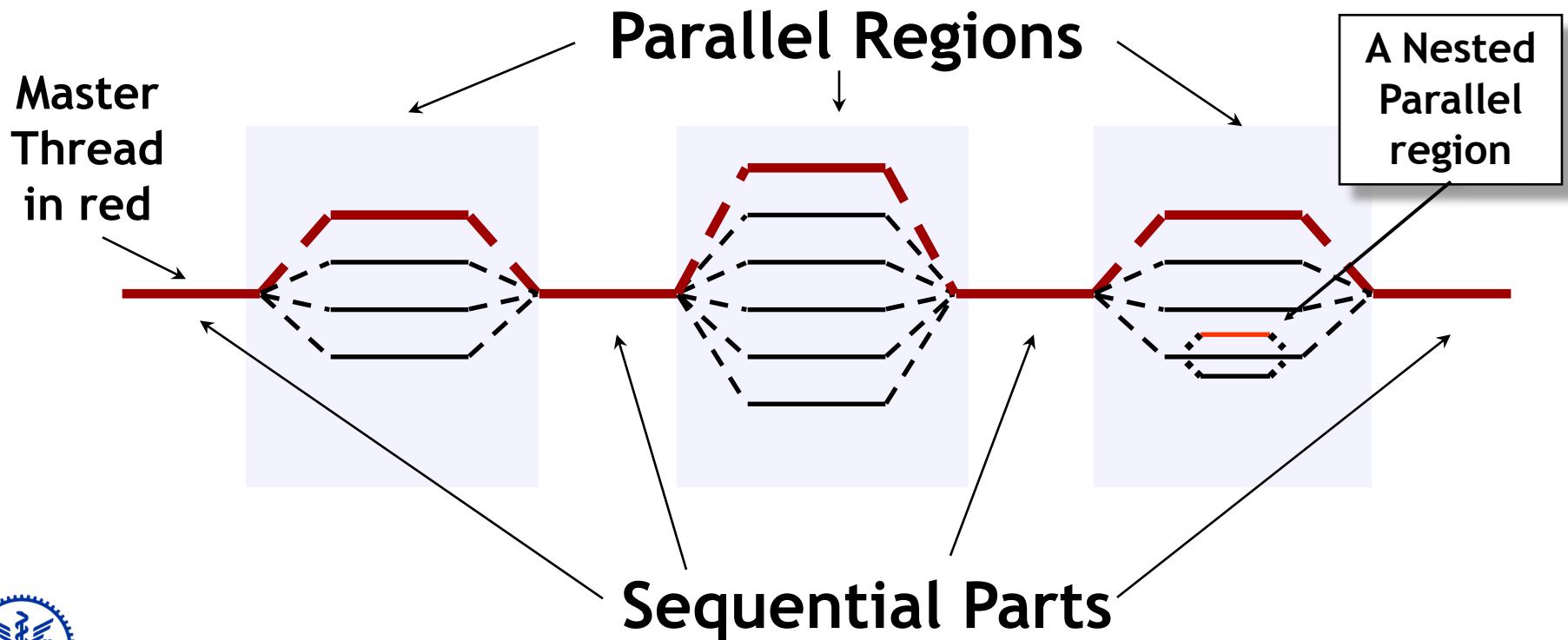
C\$OMP DO lastprivate(XX)



# OpenMP Execution Model:

## Fork-Join Parallelism:

- Master thread spawns a team of threads as needed
- Parallelism added incrementally until performance are met: i.e. the sequential program evolves into a parallel program



# OpenMP Example: Hello World

- Write a multithreaded program where each thread prints “hello world”

```
void main() {  
  
    printf(" hello(%d) ", ID);  
    printf(" world(%d) \n", ID);  
  
}
```



# OpenMP Example: Hello World

- Tell the compiler to pack code into a function, fork the threads, and join when done ...

```
#include "omp.h"
void main() {
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
}
}
```

Sample Output:  
hello(1) hello(0) world(1)  
world(0)  
hello(3) hello(2) world(3)  
world(2)



# Outline

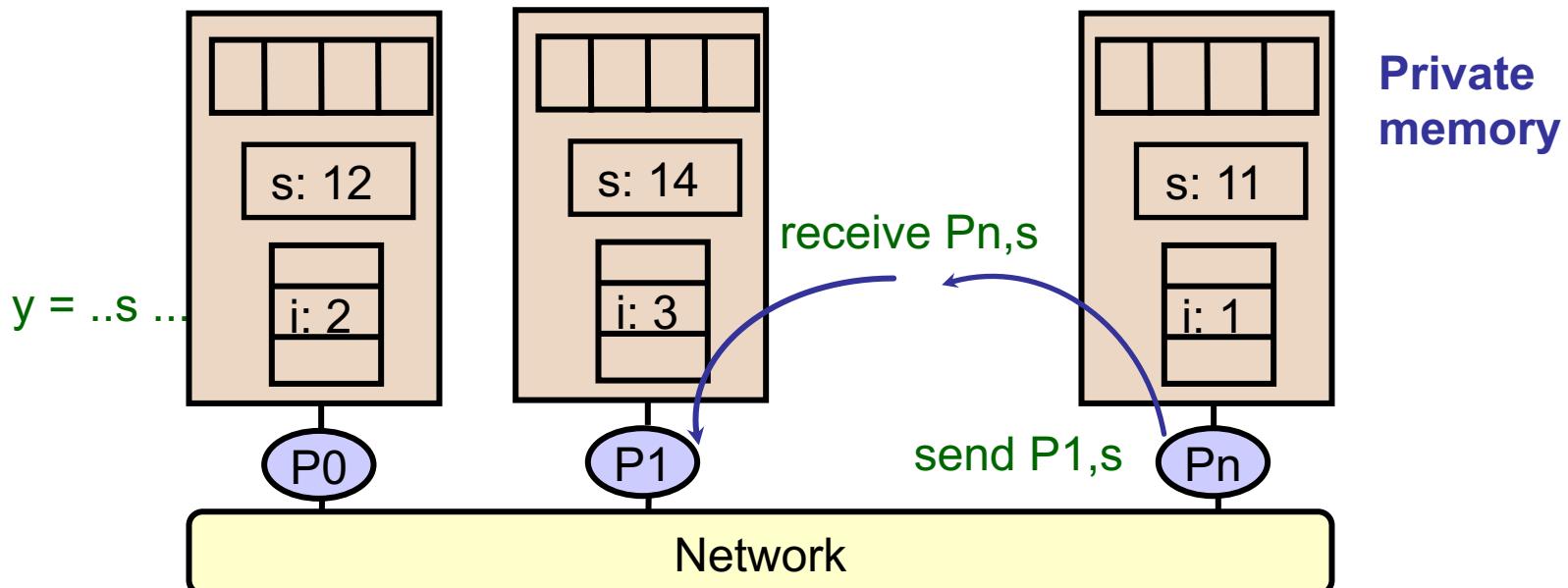
---

- The What and Why of Parallel Computing
- The Why and How of Parallel Programming
- Parallel Programming Models
  - ✿ Shared-memory model
  - ✿ **Distributed-memory model**
  - ✿ Data-parallel model
- Performance Evaluation



# Programming Model 2: Message Passing

- Program consists of a collection of **named** processes
  - Usually fixed at program startup time
  - Thread of control plus local address space -- NO shared data
  - Logically shared data is partitioned over local processes
- Processes communicate by explicit send/receive pairs
  - Coordination is implicit in every communication event
  - MPI (Message Passing Interface) is the most commonly used SW



# Computing $s = f(A[1]) + f(A[2])$ on each processor

- ° First possible solution - what could go wrong?

**Processor 1**

```
xlocal = f(A[1])
send xlocal, proc2
receive xremote, proc2
s = xlocal + xremote
```

**Processor 2**

```
xlocal = f(A[2])
send xlocal, proc1
receive xremote, proc1
s = xlocal + xremote
```

- ° If send/receive acts like the telephone system? The post office?
- ° Second possible solution

**Processor 1**

```
xlocal = f(A[1])
send xlocal, proc2
receive xremote, proc2
s = xlocal + xremote
```

**Processor 2**

```
xlocal = f(A[2])
receive xremote, proc1
send xlocal, proc1
s = xlocal + xremote
```



# MPI - the de facto standard

- MPI has become the de facto standard for parallel computing using message passing
- Example:

```
for (i=1;i<numprocs;i++) {  
    sprintf(buff, "Hello %d!", i);  
    MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);  
}  
for(i=1;i<numprocs;i++) {  
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);  
    printf("%d: %s\n", myid, buff);  
}
```

- Pros and Cons of standards
  - ⊕ MPI created finally a standard for applications development in the HPC community => portability
  - ⊕ The MPI standard is a least common denominator building on mid-80s technology, so may discourage innovation



# Which is better? SM or MP?

## ■ Which is better, Shared Memory or Message Passing?

- ⊕ Depends on the program!
- ⊕ Both are “communication Turing complete”
  - ◆ i.e. can build Shared Memory with Message Passing and vice-versa

## ■ Advantages of Shared Memory

- ⊕ Implicit communication (loads/stores)
- ⊕ Low overhead when cached

## ■ Disadvantages of Shared Memory

- ⊕ Complex to build in way that scales well
- ⊕ Requires synchronization operations
- ⊕ Hard to control data placement within caching system

## ■ Advantages of Message Passing

- ⊕ Explicit Communication (sending/receiving of messages)
- ⊕ Easier to control data placement (no automatic caching)

## ■ Disadvantages of Message Passing

- ⊕ Message passing overhead can be quite high
- ⊕ More complex to program
- ⊕ Introduces question of reception technique (interrupts/polling)



# Outline

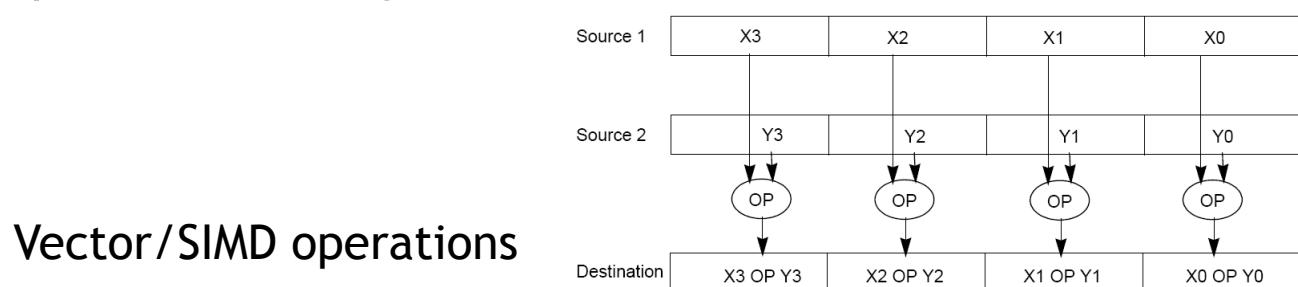
---

- The What and Why of Parallel Computing
- The Why and How of Parallel Programming
- Parallel Programming Models
  - ✿ Shared-memory model
  - ✿ Distributed-memory model
  - ✿ Data-parallel model
- Performance Evaluation



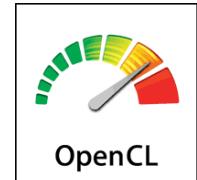
# Programming Model 3: Data Parallel

- Single thread of control consisting of **parallel operations**
  - ⊕ A = B + C could mean adding two arrays in parallel
- Parallel operations applied to all (or a defined subset) of a data structure, usually an array
  - ⊕ Communication is implicit in parallel operators
  - ⊕ Elegant and easy to understand and reason about
  - ⊕ Coordination is implicit - statements executed synchronously
  - ⊕ Similar to Matlab language for array operations
- Drawbacks:
  - ⊕ Not all problems fit this model
  - ⊕ Difficult to map onto coarse-grained machines

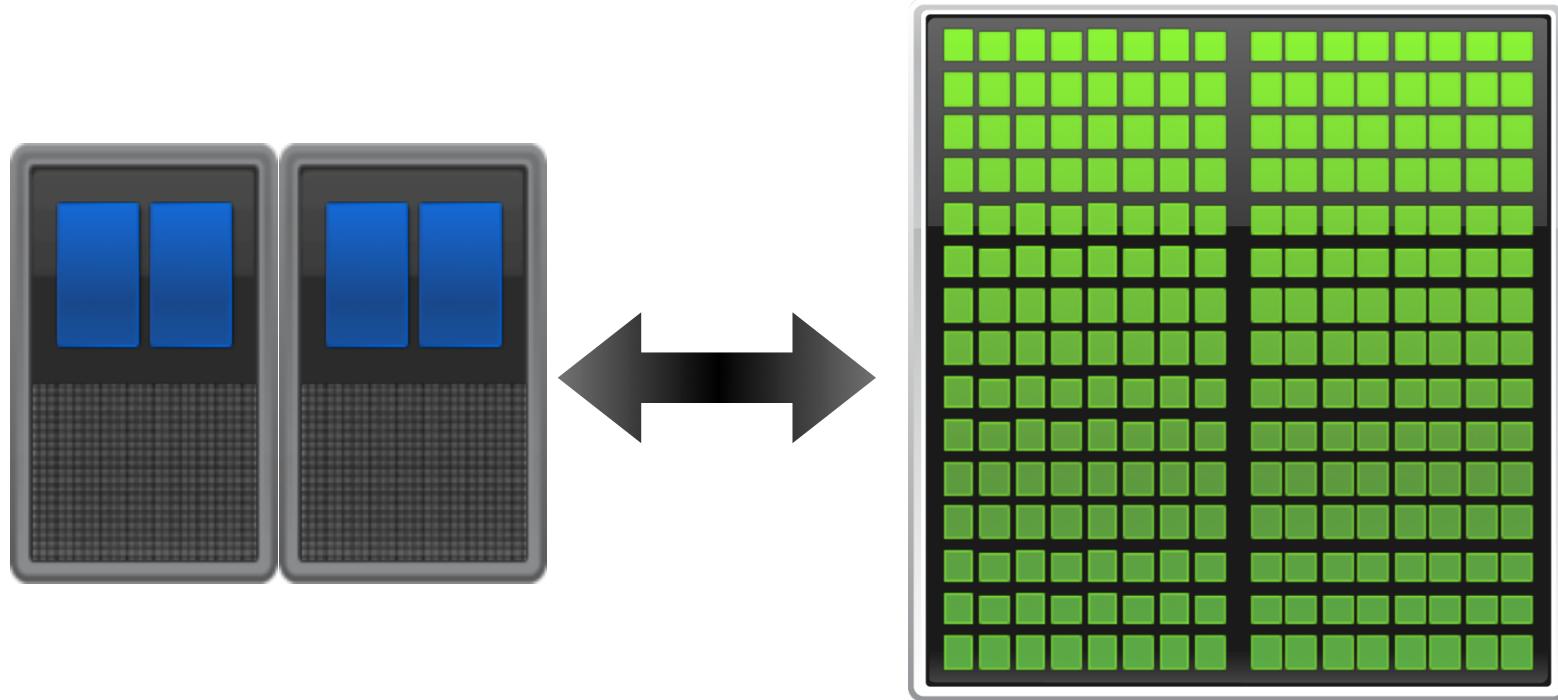


# What about GPU and Cloud?

- GPU's big performance opportunity is data parallelism
  - ⊕ Most programs have a mixture of highly parallel operations, and some not so parallel
  - ⊕ GPUs provide a threaded programming model (CUDA) for data parallelism to accommodate both
  - ⊕ Current research attempting to generalize programming model to other architectures, for portability (OpenCL)
- Cloud computing allows large numbers of people easily share  $O(10^5)$  machines
  - ⊕ MapReduce was first programming model: data parallel on distributed memory
  - ⊕ More flexible models (Hadoop...) invented since then



# CPU-GPU: Heterogeneous Parallel Computing



Latency-  
optimized CPU

Fast serial  
processing

Throughput-  
optimized GPU

Scalable parallel  
processing



# CUDA (Compute Unified Device Architecture)

---

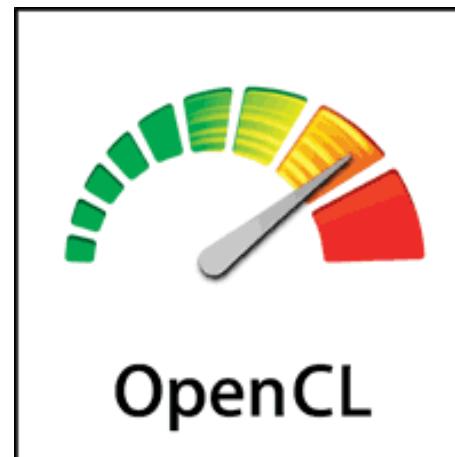
- ***Data-parallel*** programming interface to GPU
  - ⊕ Data to be operated on is discretized into independent partition of memory
  - ⊕ Each thread performs roughly same computation to different partition of data
- Programmer expresses
  - ⊕ Thread programs to be launched on GPU, and how to launch
  - ⊕ Data organization and movement between host and GPU
  - ⊕ Synchronization, memory management, ...
- CUDA is one of first to support ***heterogeneous*** architectures (more later in the semester)



# OpenCL

---

- Open Computing Language
- A framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs and other processors



# CUDA Example: Vector Add

```
int main() {
    int A[ ] = {...};
    int B[ ] = {...};

    for (i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
    }
}
```

Sequential version

```
__global__ void vecAdd(int A[ ], int B[ ], int C[ ]) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

```
int main() {
    int hA[ ] = {...};
    int hB[ ] = {...};

    cudaMemcpy(dA, hA, sizeof(hA), HostToDevice);
    cudaMemcpy(dB, hB, sizeof(hB), HostToDevice);

    vecAdd<<<1, N>>>(dA, dB, dC);

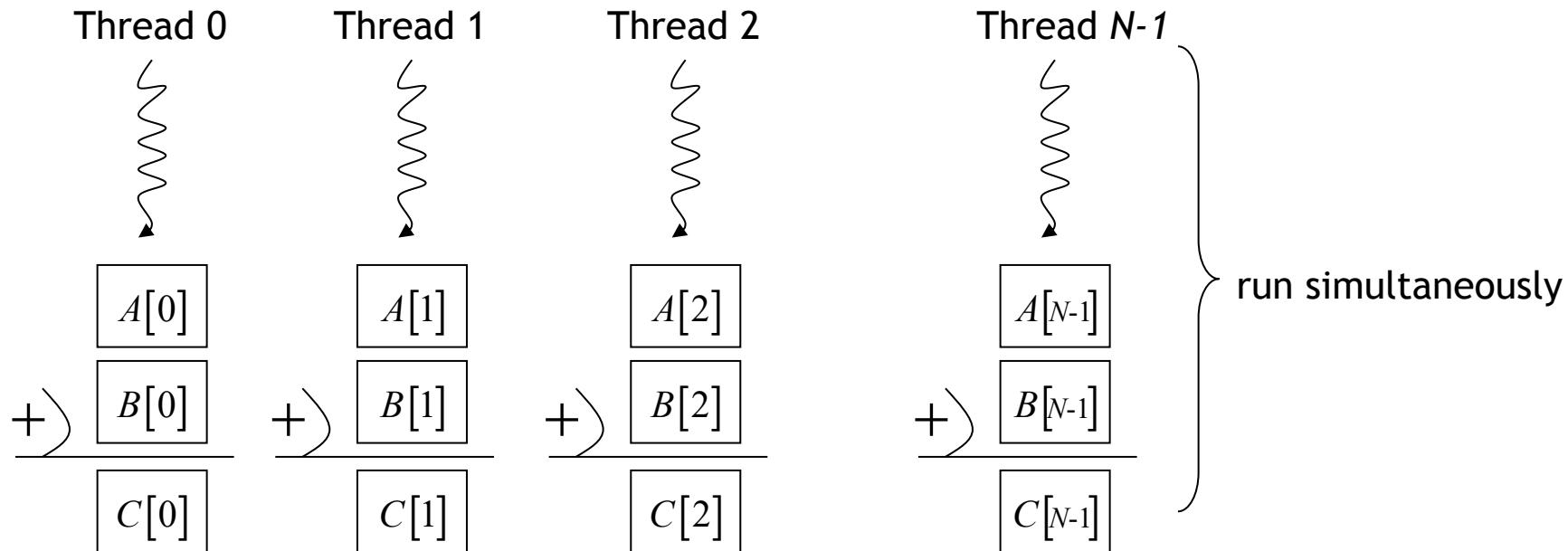
    cudaMemcpy(dC, hC, sizeof(hC), DeviceToHost);
}
```

Parallel version



# CUDA Example: Vector Add

```
__global__ void vecAdd(int A[ ], int B[ ], int C[ ]) {  
    int i = threadIdx.x;  
    C[i] = A[i] + B[i];  
}
```



# MapReduce

---

- MapReduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster
- The model is inspired by the map and reduce functions commonly used in functional programming



# Map and Reduce

## Map

- ⊕ The master node takes the input, divides it into smaller sub-problems, and distributes them to worker nodes
- ⊕ The worker node processes the smaller problem, and passes the answer back to its master node

Map function:

$$(K_{in}, V_{in}) \rightarrow \text{list} <(K_{inter}, V_{inter})>$$

## Reduce

- ⊕ The master node then collects the answers to all the sub-problems and combines them in some way to form the output, the answer to the problem it was originally trying to solve

Reduce function:

$$(K_{inter}, \text{list} < V_{inter} >) \rightarrow \text{list} < (K_{out}, V_{out}) >$$



# Typical Problem

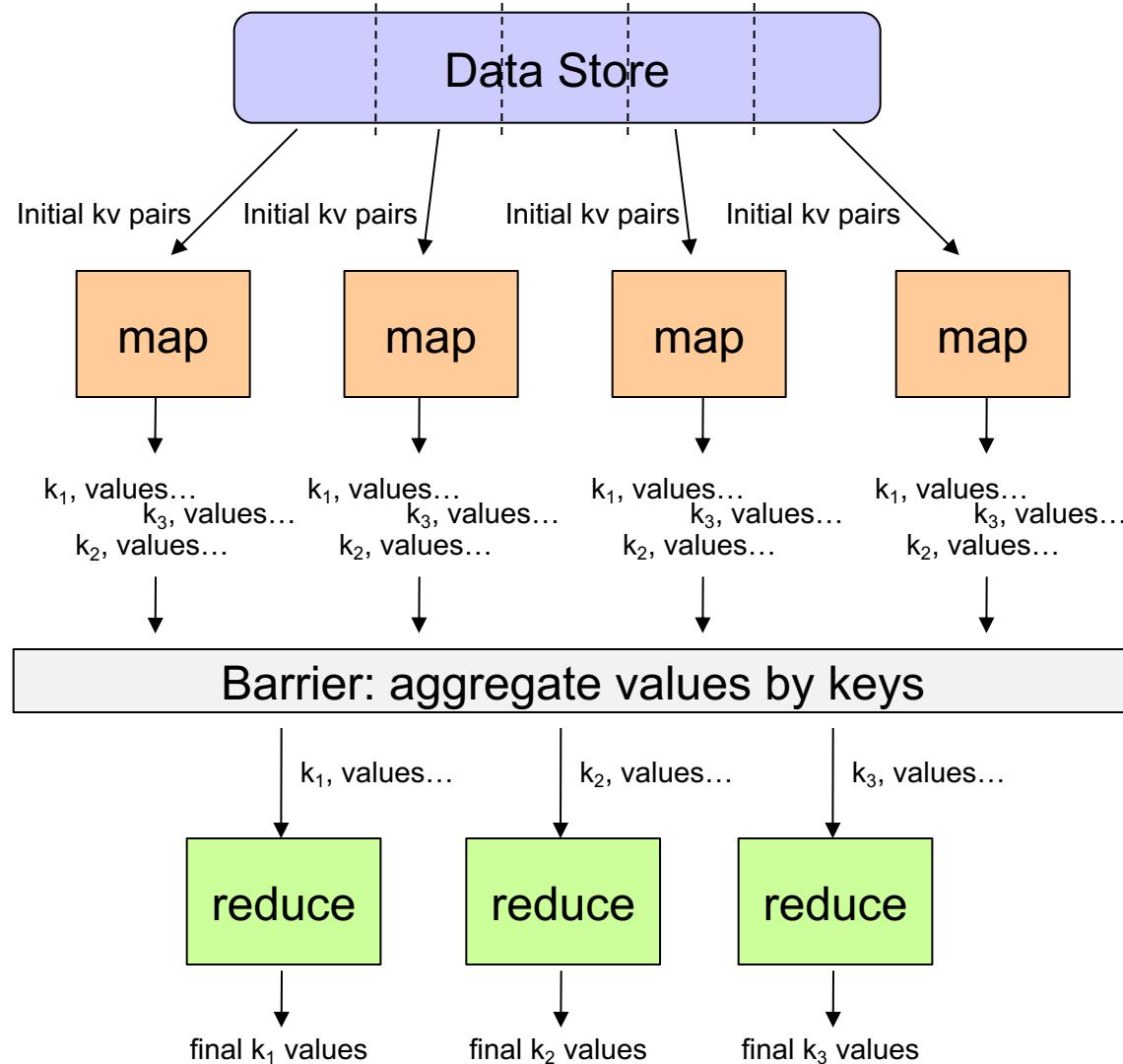
---

- Iterate over a large number of records
- **Map**: extract something of interest from each
- Shuffle and sort intermediate results
- **Reduce**: aggregate intermediate results
- Generate final output

**Key idea:** provide an abstraction at the point of these two operations



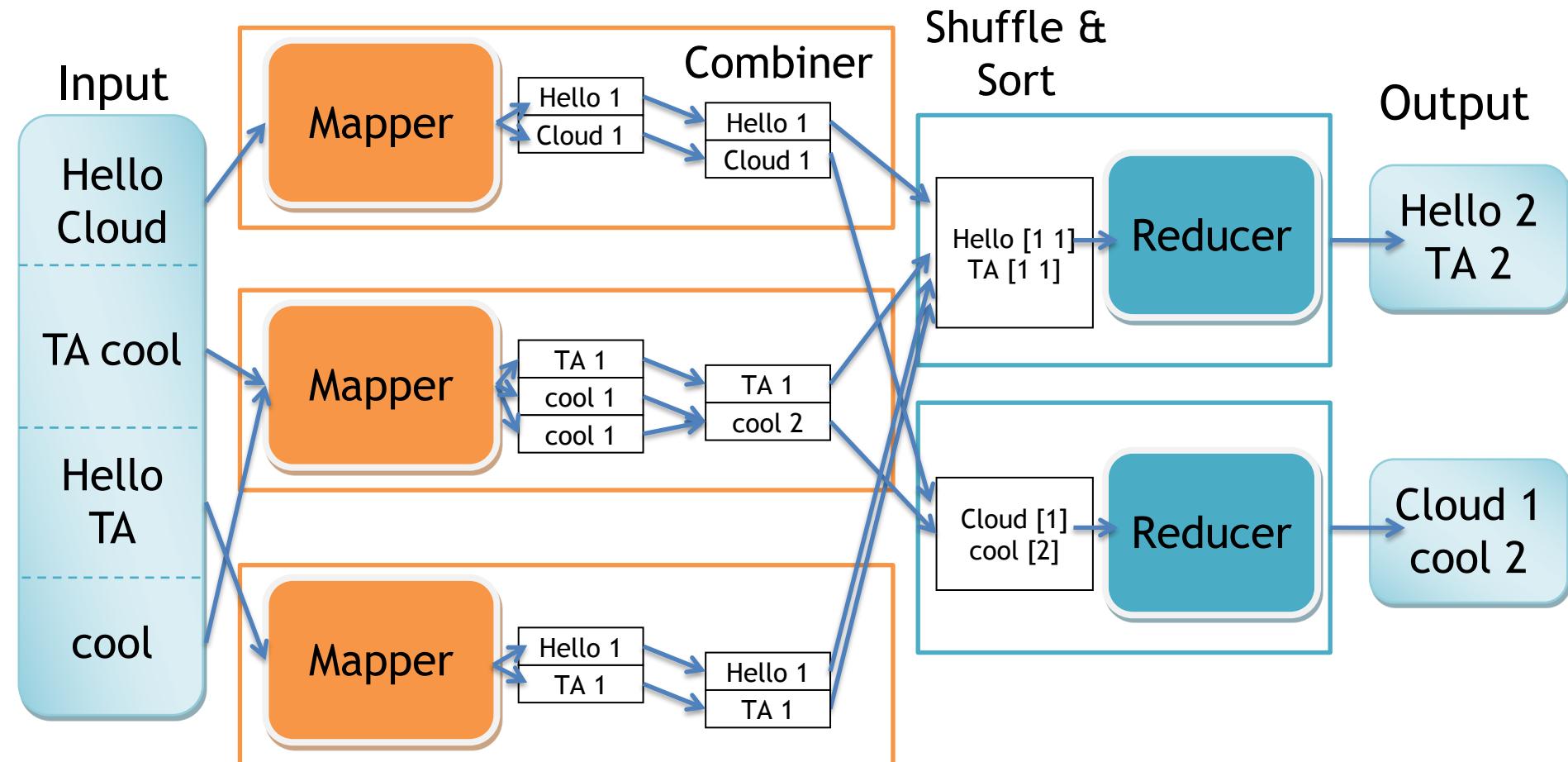
# It's just divide and conquer!



# Example: Word Count

```
def map(line_num, line):
    foreach word in line.split():
        output(word, 1)
```

```
def reduce(word, counts):
    output(word, sum(counts))
```



# Hadoop

---

- Hadoop (High-availability distributed object-oriented platform)
- An open-source software framework that supports data-intensive distributed applications
- Distributed File System (HDFS)
  - Runs on same machines
  - Replicates data 3x for fault-tolerance
- It supports the running of applications on large clusters of commodity hardware



# Hadoop

---

- Hadoop was derived from Google's MapReduce and Google File System (GFS) papers
- Hadoop is written in the Java programming language



# Remark

---

- Three basic conceptual models
  - ❖ Shared memory
  - ❖ Distributed memory
  - ❖ Data parallel

and hybrids of these models
- All of these models rely on dividing up work into parts that are:
  - ❖ Mostly independent (little synchronization)
  - ❖ About same size (load balanced)
  - ❖ Have good locality (little communication)



# Outline

---

- The What and Why of Parallel Computing
- The Why and How of Parallel Programming
- Parallel Programming Models
- **Performance Evaluation**



# Performance

---

- Of course our main purpose in writing parallel programs is usually increased performance
- So what can we expect? And how can we evaluate our programs?
  - ◆ Speedup
  - ◆ Efficiency
  - ◆ Scalability



# Speedup

---

- We run our program with  $p$  cores, one thread or process on each core, then our parallel program will run  $p$  times faster than the serial program
- When this happens, we say that our parallel program has **linear speedup**



# Speedup

---

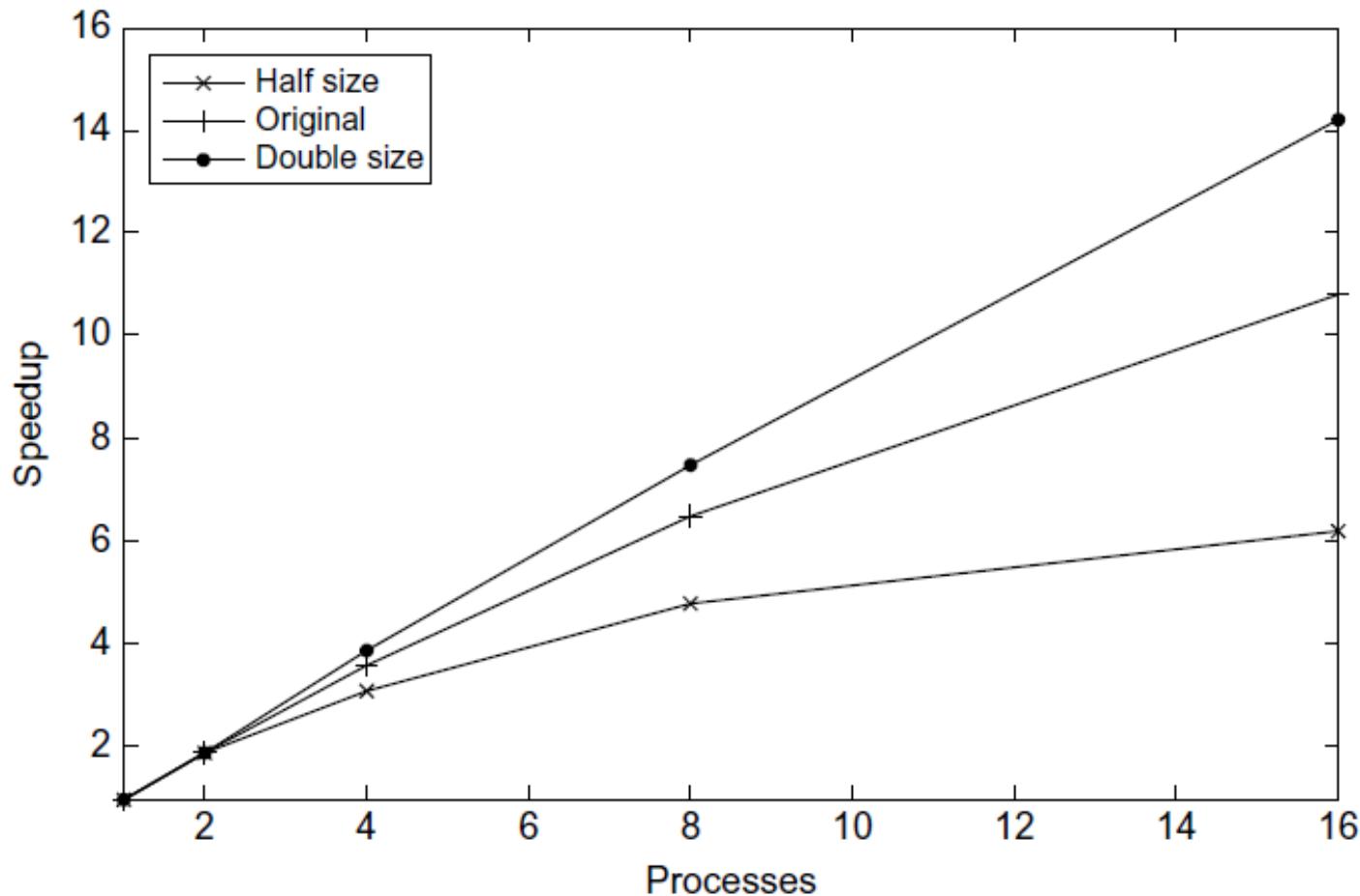
- In practice, we're unlikely to get linear speedup because the use of multiple processes almost invariably introduces some overhead
- We define the speedup of a parallel program to be:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

- Then linear speedup has  $S = p$



# Speedup



Speedups of parallel program on different problem sizes



# Efficiency

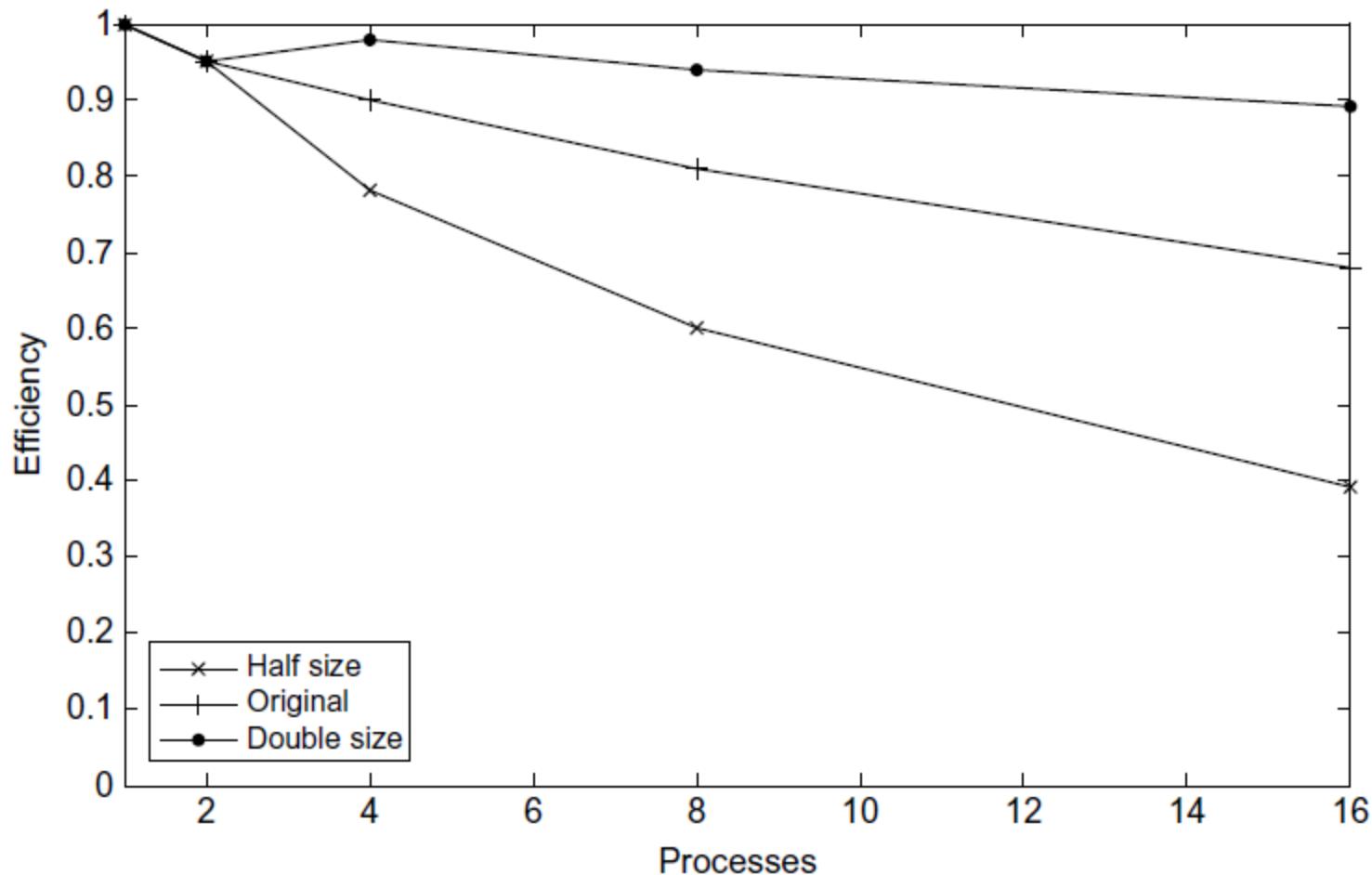
---

- $S/p$ , is sometimes called the **efficiency** of the parallel program

$$E = \frac{S}{p} = \frac{\left( \frac{T_{\text{serial}}}{T_{\text{parallel}}} \right)}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}$$



# Efficiency



Efficiencies of parallel program on different problem sizes



# Scalability

- If we increase the problem size at the same rate that we increase the number of processes/threads, then the efficiency will be unchanged, and our program is scalable



AUTO SCALE →



Grows without requiring developers to re-architect their algorithms/applications



# Strongly Scalable and Weakly Scalable

---

- There are a couple of cases that have special names:
  - 1) **Strongly scalable**
    - When we increase the number of processes/threads, we can keep the efficiency fixed *without* increasing the problem size
  - 2) **Weakly scalable**
    - We can keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads



# A Word of Warning

---

- Every parallel program contains at least one serial program
- Since we almost always need to coordinate the actions of multiple cores, writing parallel programs is almost always more complex than writing a serial program that solves the same problem



# Amdahl's Law

- The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program

$$\text{Speedup} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}$$

500

100

100

100

100

400 → 1.25

100

50 50

100

50 50

100

350 → 1.43

100

25 25 25 25

100

25 25 25 25

100

300 → 1.67

100

100

100



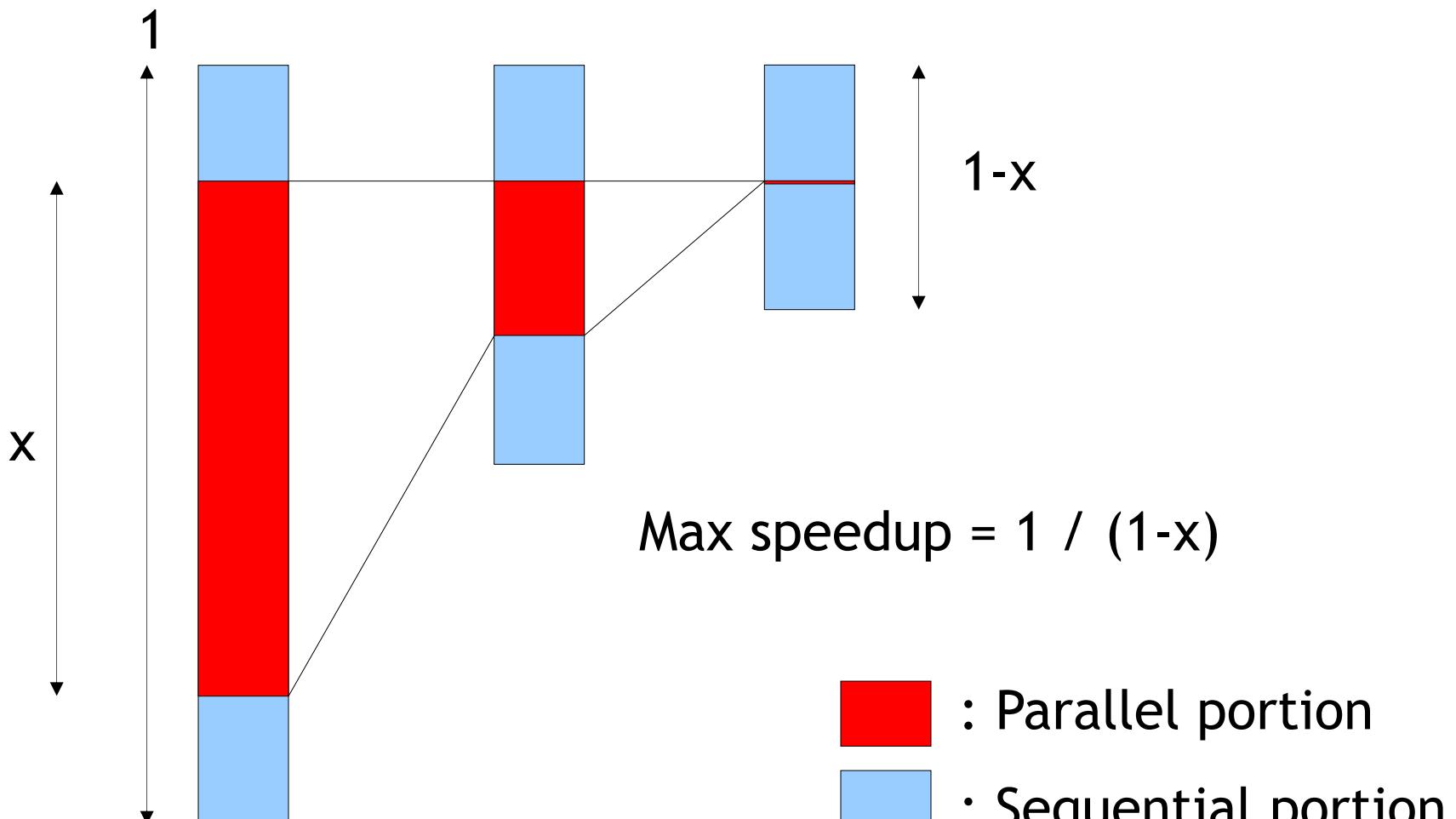
: Parallel portion



: Sequential portion



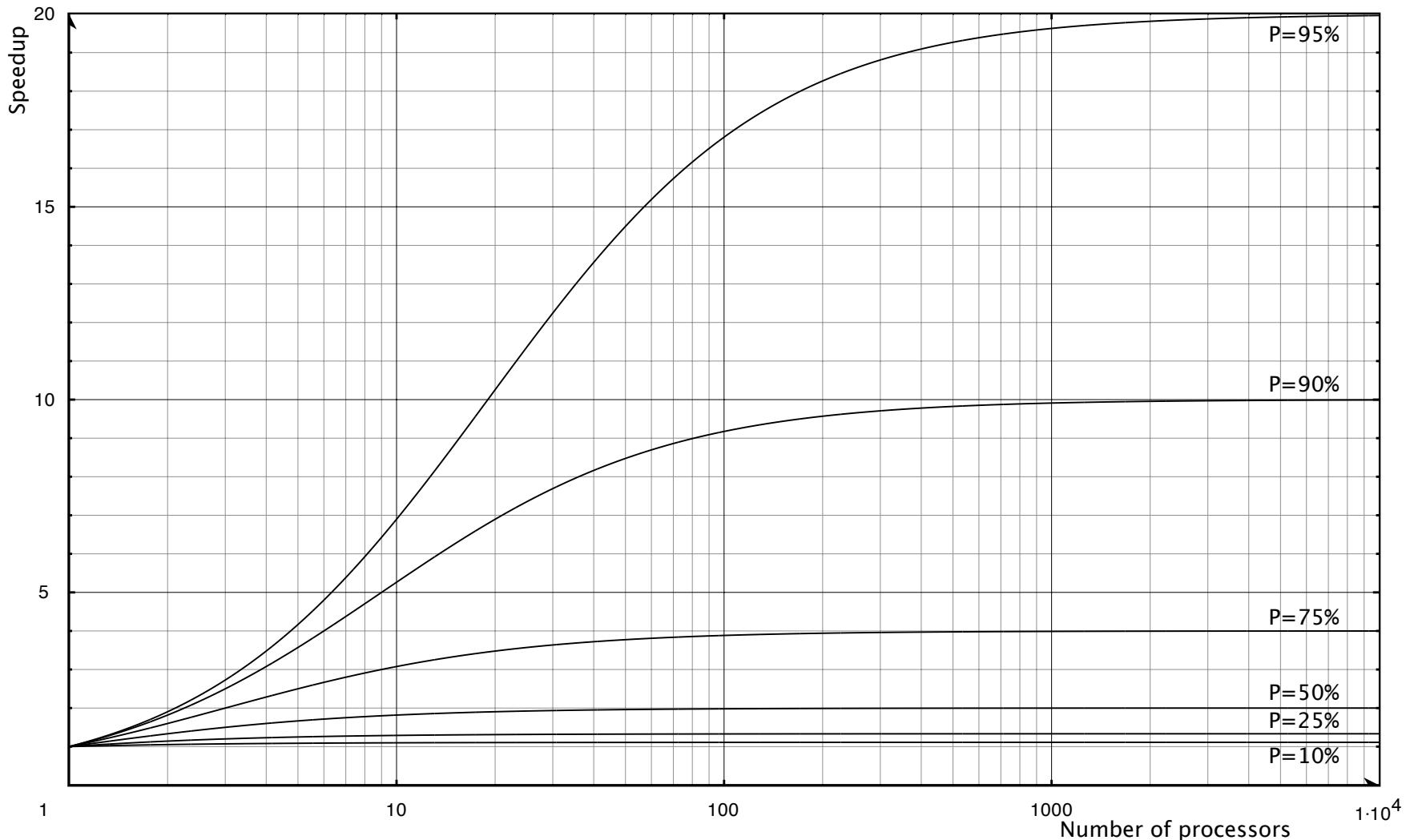
# Amdahl's Law (Cont'd)



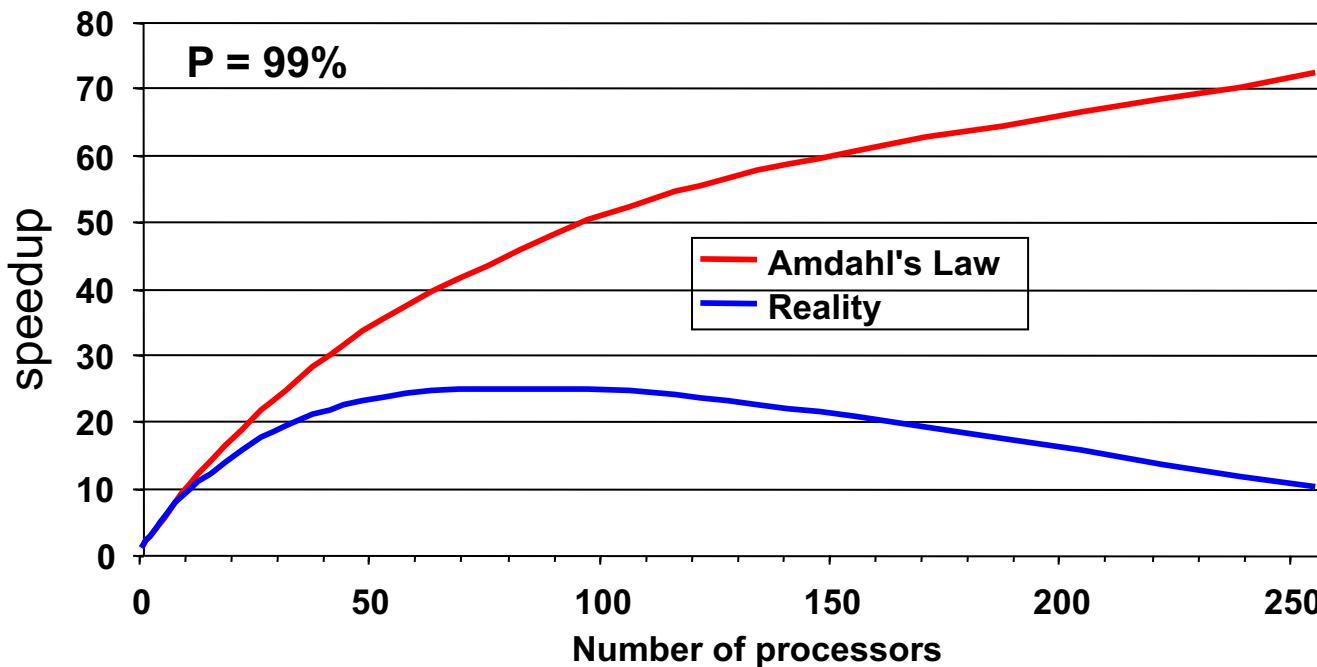
$$S(n) = \frac{s + p}{s + p/n} = \frac{1}{(1-p) + p/n}$$



# Amdahl's Law (Cont'd)



# Amdahl's Law vs Reality



- Amdahl's Law provides a theoretical upper limit on parallel speedup assuming that there are no costs for communications
  - ✿ In reality, communications will result in a further degradation of performance



# More on Amdahl's Law

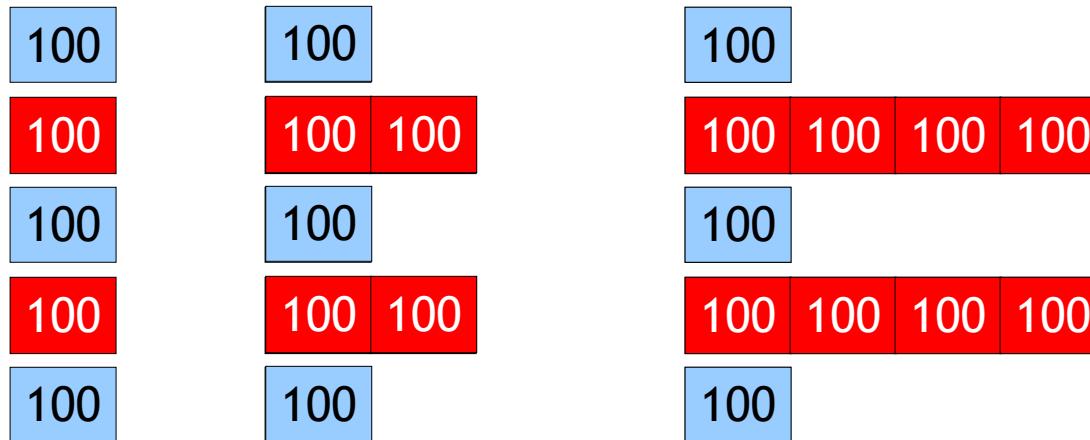
---

- Amdahl's Law works on a *fixed* problem size
  - ✿ This is reasonable if your only goal is to solve a problem faster
  - ✿ What if you also want to solve a larger problem?
- Gustafson's Law:
  - ✿ The proportion of the computations that are sequential normally decreases as the problem size increases



# Gustafson's Law

- Rather than assume that the problem size is fixed, assume that the parallel execution time is fixed
  - ⊕ In increasing the problem size, Gustafson also makes the case that the serial section of the code does not increase as the problem size



- ⊕ Scaled speedup

$$S_s(n) = \frac{s + np}{s + p} = s + np = (1 - p) + np = 1 + (n - 1)p$$



# Example of Gustafson's Law

- Suppose a parallel section of 95% and 20 processors; the speedup according to the formula is 19.05 instead of 10.26 according to Amdahl's law
  - ❖ Note, however, the different assumptions

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}} = \frac{1}{0.05 + \frac{0.95}{20}} \approx 10.26$$

$$S_s(n) = 1 + (n-1)p = 1 + (20-1)*0.95 = 19.05$$



# Summary

---

- In order to increase the power of processors, chipmakers have turned to **multicore** integrated circuits
- Ordinary **serial** programs usually cannot exploit the presence of multiple cores



# Summary

---

- When we write parallel programs, we usually need to **coordinate** the work of the cores. This can involve **communication** among the cores, **load balancing**, and **synchronization** of the cores
- Parallel programs are usually **very complex**. So it's even more important to use good program development techniques with parallel programs



# References

---

- Peter Pacheco, An Introduction to Parallel Programming, Morgan Kaufmann; 1 edition (January 21, 2011).
- MIT Course 6.189, “Multi-core Programming Primer: Learn and Compete in Programming the PLAYSTATION®3 Cell Processor”
- UCB Course CS267, “Applications of Parallel Computers”
- “Overview of Multicore Programming” - Prof. Tien-Fu Chen
- Blaise Barney, Lawrence Livermore National Laboratory, “Introduction to Parallel Computing”  
([https://computing.llnl.gov/tutorials/parallel\\_comp/](https://computing.llnl.gov/tutorials/parallel_comp/))

