ECE 432/532
Programming for Parallel Processors

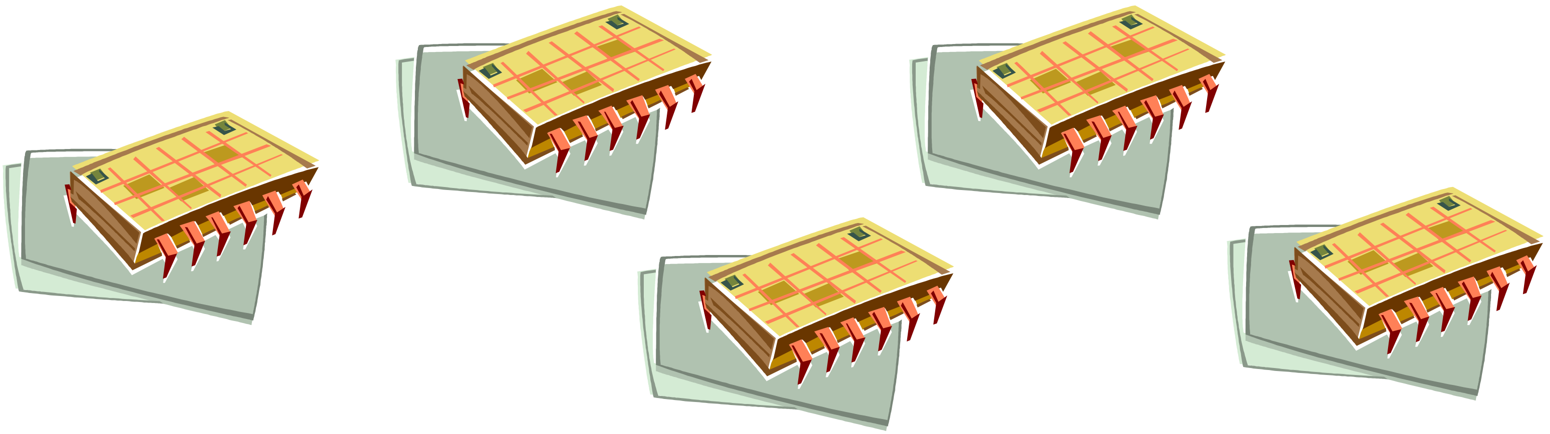
So far…

- <u>1986-2002</u>: The performance of microprocessors increased, on average, 50% per year
- <u>2002</u>: Single-processor performance improvement has slowed to about 20% per year
- <u>2005</u>: Most of the major manufacturers decided that the road to rapidly increasing performance lay in the direction of parallelism
  - Rather than trying to continue to develop ever-faster monolithic processors, manufacturers started putting *multiple* complete processors on a single integrated circuit.

Simply adding more processors will **not** magically improve the performance of the vast majority of **serial programs**, that is,
programs that were written to run on a single processor.

An intelligent solution

- Instead of designing and building faster microprocessors, put <u>multiple</u> processors on a single integrated circuit.



All of this raises a number of questions

- Why do we care? Aren’t single processor systems fast enough?

- Why can’t microprocessor manufacturers continue to develop much faster single processor systems? Why build **parallel systems**?

- Why can’t we write programs that will automatically convert serial programs into **parallel programs?**

Now it's up to the programmers

- Adding more processors doesn't help much if programmers aren't aware of them…
- … or don't know how to use them.

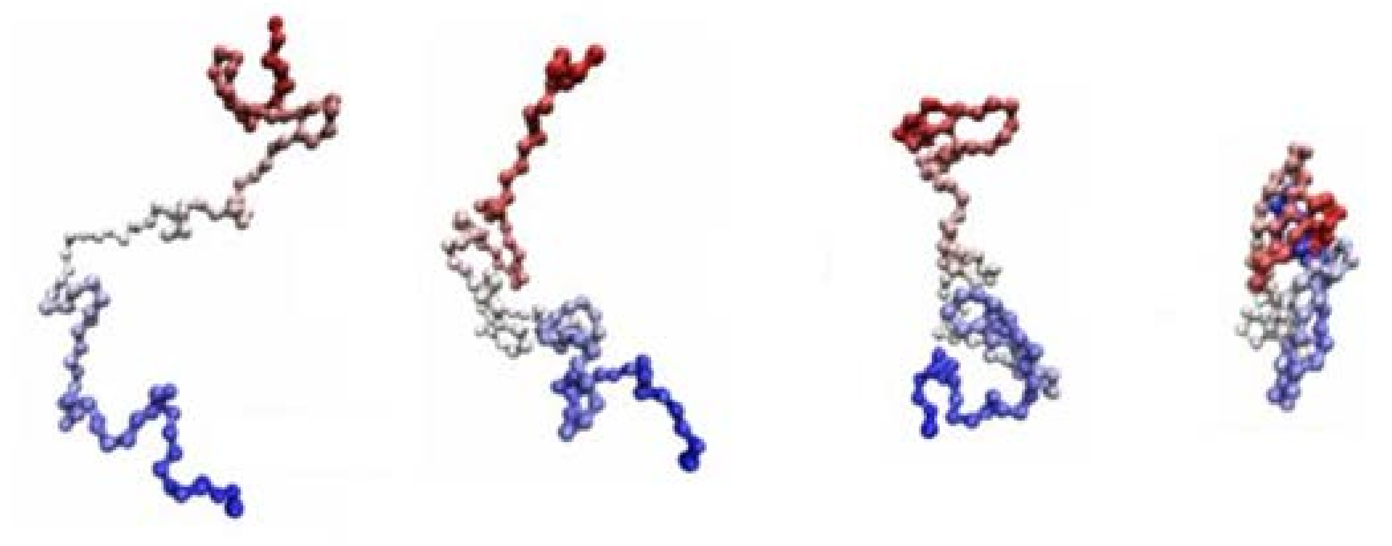
- Serial programs don't benefit from this approach (in most cases).

Why do we care?

- Application needs:
  - *Climate modeling*
  - *Protein folding*
  - *Drug discovery*
  - *Data analysis*
  - *DNA analysis*
  - *…*

Climate modeling

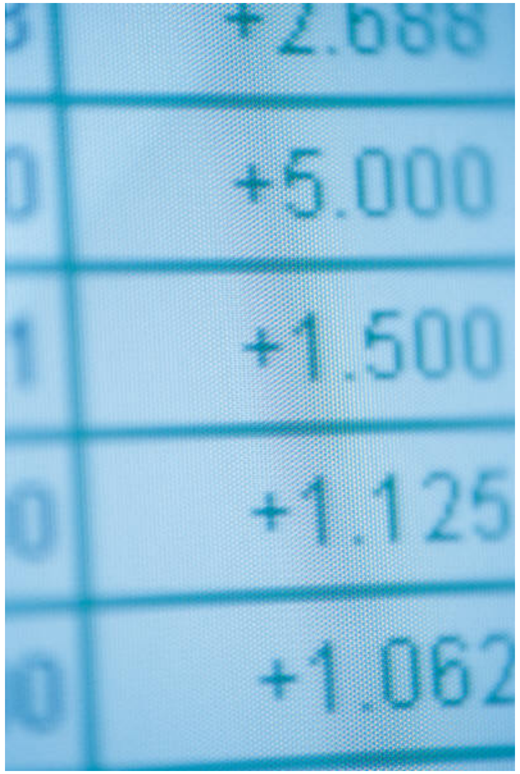
Protein folding



Drug discovery

Energy research

Data analysis



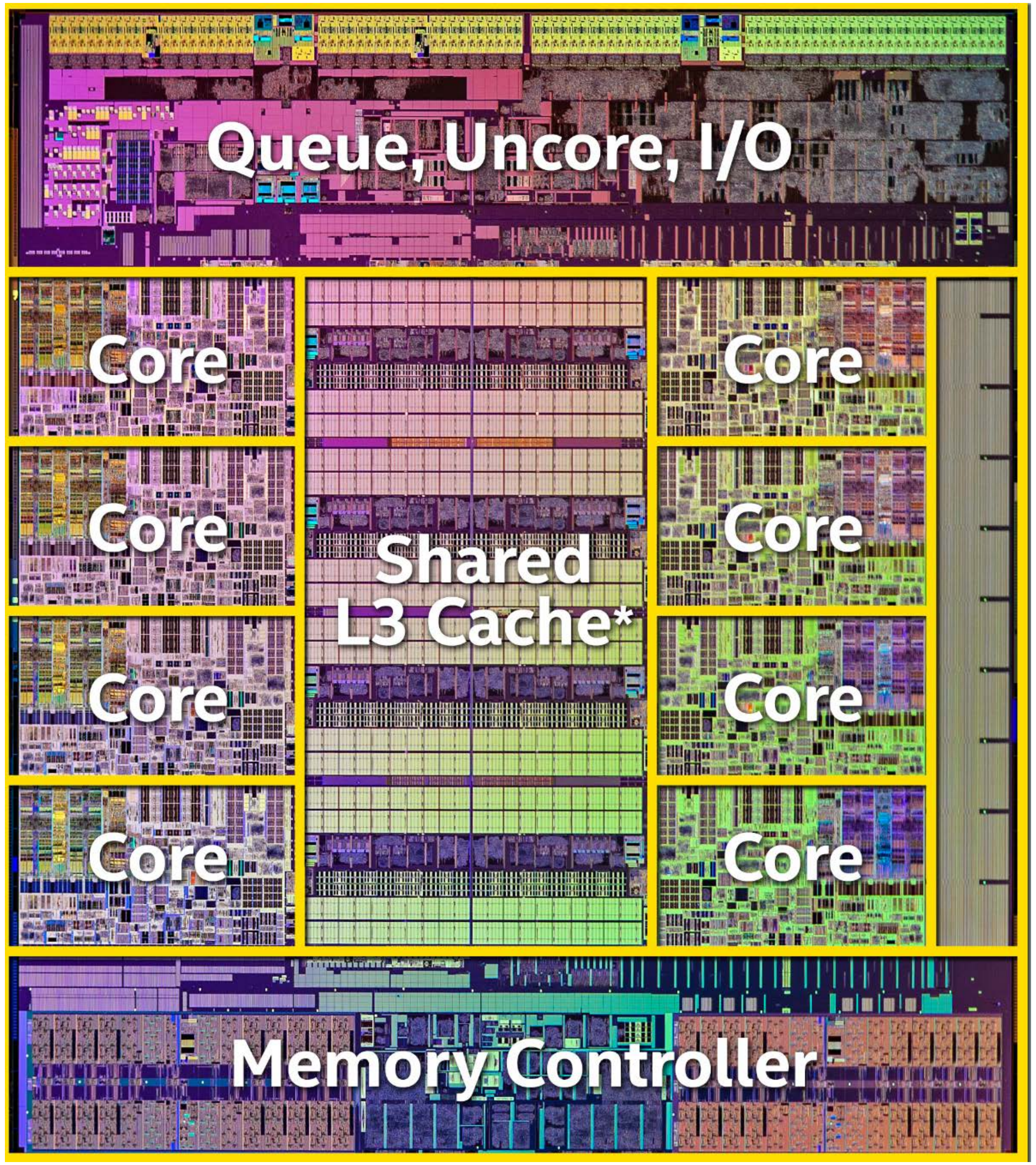
Why build parallel systems?

- Much of the increase in single processor performance has been driven by the ever-increasing density of transistors

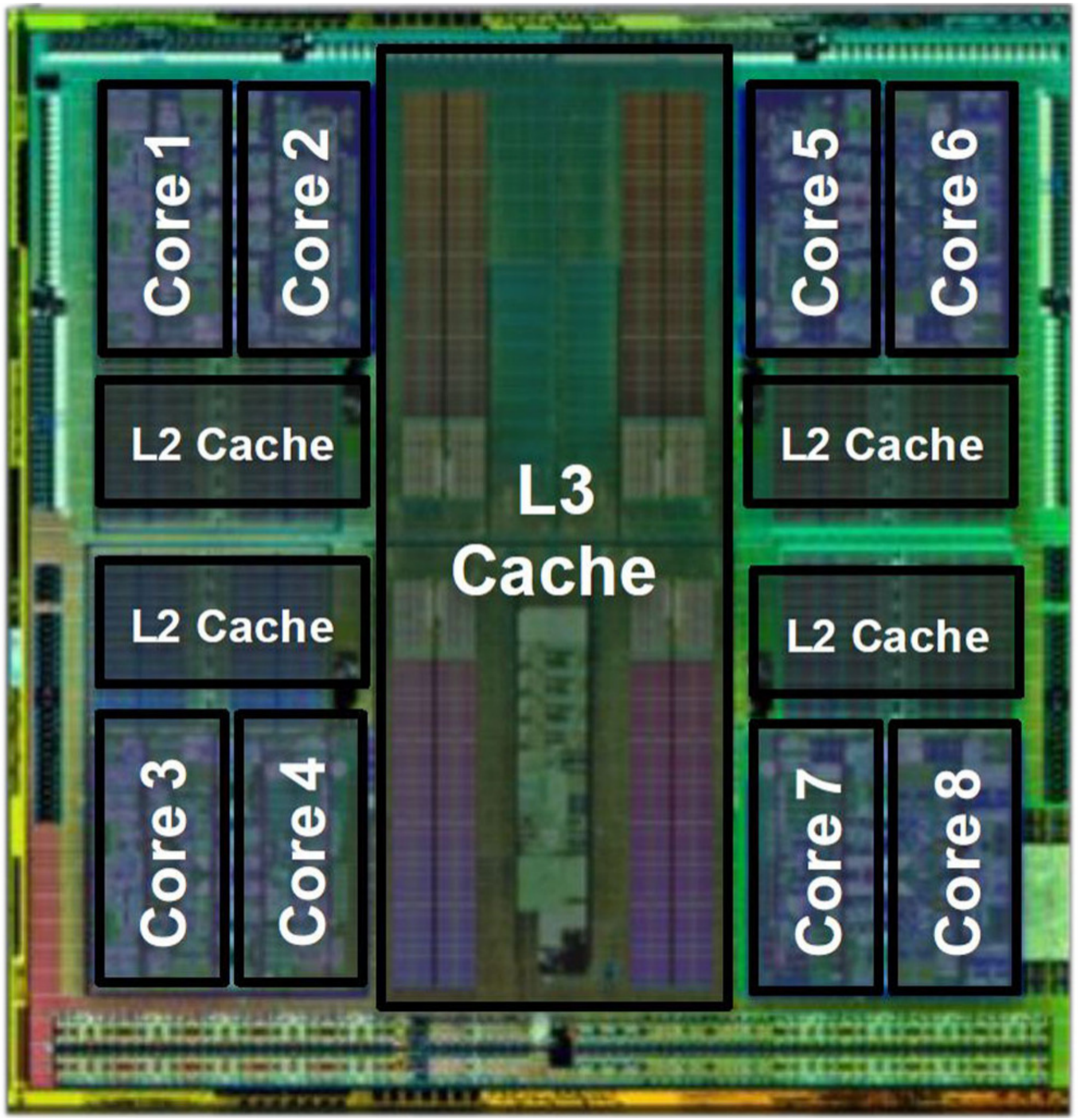
- As the size of transistors decreases, their speed can be increased
  - Smaller transistors = faster processors
  - Faster processors = increased power consumption.

- However, as the speed of transistors increases, their power consumption also increases → Heat problems, unreliability

- *Parallelism:* put multiple, relatively simple, complete processors on a single chip. Such integrated circuits are called multicore processors intel


Queue, Uncore, I/O
Core
Core
Core
Core
Shared
L3 Cache*
Core
Core
Core
Core
Memory Controller

AMD


Core 1
Core 2
L2 Cache
L2 Cache
Core 3
Core 4
L3
Cache
Core 5
Core 6
L2 Cache
L2 Cache
Core 7
Core 8

Fun facts

- **1978**: Intel introduces the 16-bit 8086 microprocessor. It will become an industry standard
- **2003:** AMD introduces the x86-64, a 64-bit superset of the x86 instruction set
- **2004:** AMD demonstrates an x86 dual-core processor chip
- **2005:** Intel ships its first dual-core processor chip

Why we need to write parallel programs?

- Most programs that have been written for single-core systems
  - We can run multiple instances of a program on a multicore system, but this is often of little help

- Example: Video games
  - We can run multiple instances of our favorite game program

or

  - run faster with more realistic graphics

Why we need to write parallel programs?

- Solutions:
  - Rewrite the serial programs so that they're *parallel*
  - write translation programs → automatically convert serial to parallel code

Why we need to write parallel programs?

- Solutions:
  - Rewrite the serial programs so that they're *parallel*
  - ~~write translation programs → automatically convert serial to parallel code~~

Researchers have had very limited success writing programs that convert serial programs in languages such as C and C++ into parallel programs

Serial to parallel

- We can view the multiplication of two *n x n* matrices as a sequence of dot products

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}_{C}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

Serial to parallel

- We can view the multiplication of two *n x n* matrices as a sequence of dot products

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}_{C}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

but parallelizing a matrix multiplication as a sequence of parallel dot products is likely to be very slow on many systems

Serial to parallel

- An efficient parallel implementation of a serial program may not be obtained by finding efficient parallelizations of each of its steps.

- Rather, the best parallelization may be obtained by stepping back and devising an entirely new algorithm

Example

- Compute *n* values and add them together:

Example

- Compute *n* values and add them together:

Serial code

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

Example

- Compute *n* values and add them together:

Serial code

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

What about the parallel version?

Example

- Compute *n* values and add them together:

- Parallel code → We have *p* cores and *p* << *n* →

each core forms 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;
}
```

Each core uses it's own private variables and executes this block of code independently of the other cores.

Example

- Compute *n* values and add them together:

- Parallel code → We have *p* cores and *p* << *n* →

each core forms 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;
}
```

Each core uses it's own private variables and executes this block of code independently of the other cores.

Example

- For *p=8* and *n=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,

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

Example

- How do you find the final sum?
  - When the cores are done, they can form a global sum by sending their results to a designated "master" core, which can add their results

```
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;
}
```

Example

- If the master core is core 0, it would add the values:

8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95.

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

```
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;
}
```

Example

- If the master core is core 0, it would add the values:

8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95.

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

```
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;
}
```

Is this an effective way?

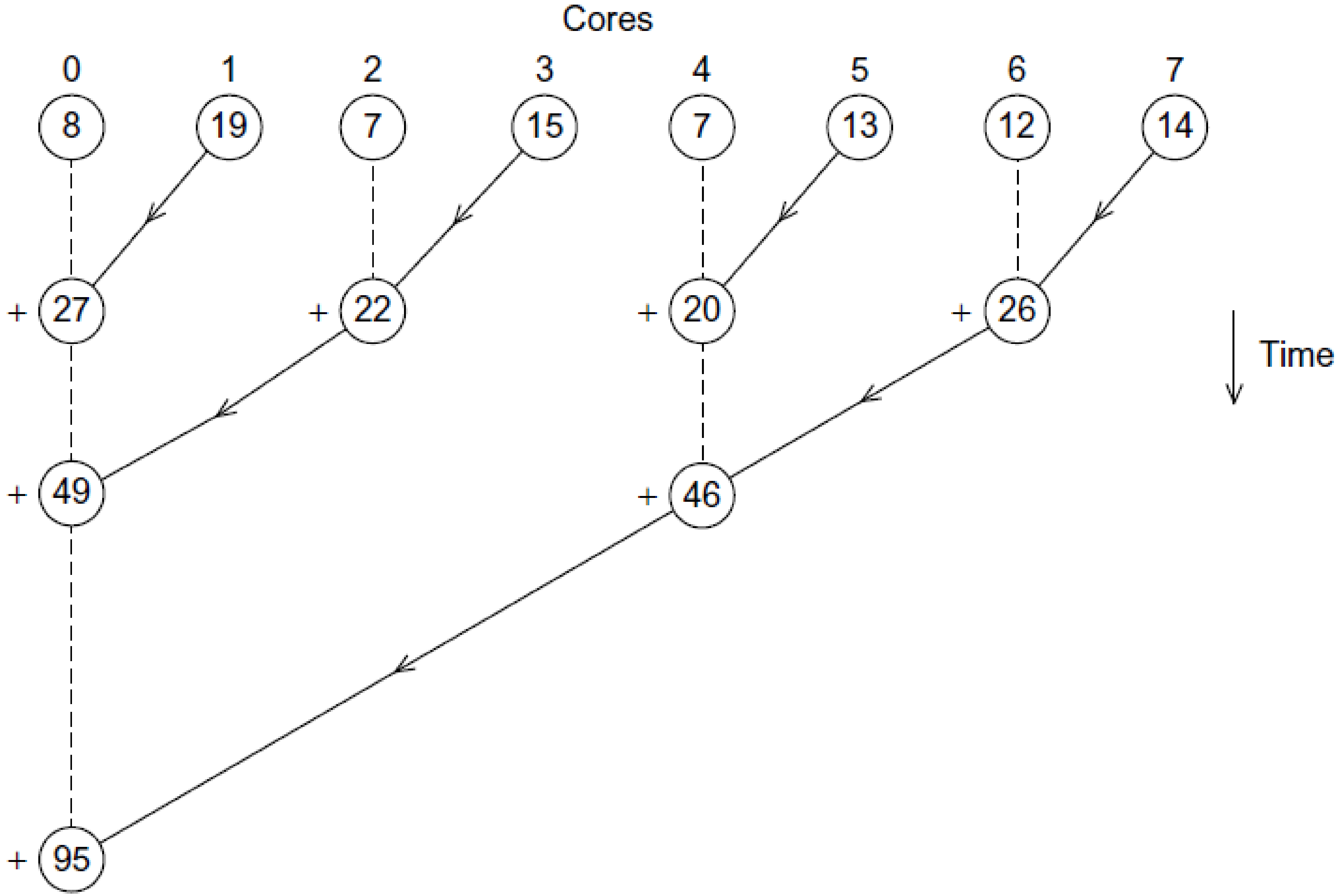
What happens when p >> 0

Example

- Don't make the master core do all the work.

- Alternative solution: pair the cores so that
  - core 0 adds in the result of core 1
  - core 2 can in the result of core 3
  - core 4 can add in the result of core 5
  - and so on.

Example



Analysis

- In the first example, the master core performs 7 receives and 7 additions.

- In the second example, the master core performs 3 receives and 3 additions.

- The improvement is more than a factor of 2!

Analysis (cont.)

- The difference is more dramatic with a larger number of cores.
- If we have 1000 cores:
  - The first example would require the master to perform 999 receives and 999 additions.
  - The second example would only require 10 receives and 10 additions.

- That's an improvement of almost a factor of 100!

Conclusions

- The first global sum is a fairly obvious generalization of the serial global sum
- The point here is that it's unlikely that a translation program would "discover" the second global sum
- Rather there would more likely be a predefined efficient global sum that the translation program would have access to
  - It could "recognize" the original serial loop and replace it with a precoded, efficient, parallel global sum.

How do we write parallel programs?

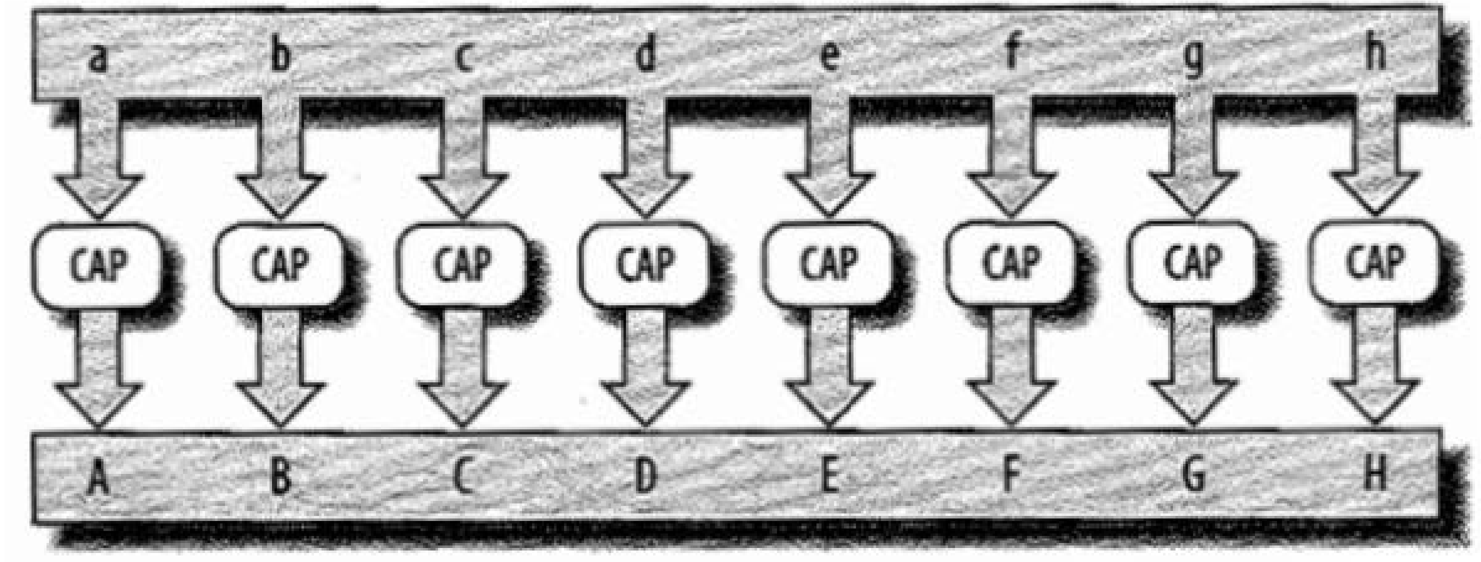
- Partition the work to be done among the cores:
  - **task-parallelism**
  - **data-parallelism**

- Task parallelism:
  - Different tasks running on the same data

- Data parallelism:
  - The same task runs on different data in parallel

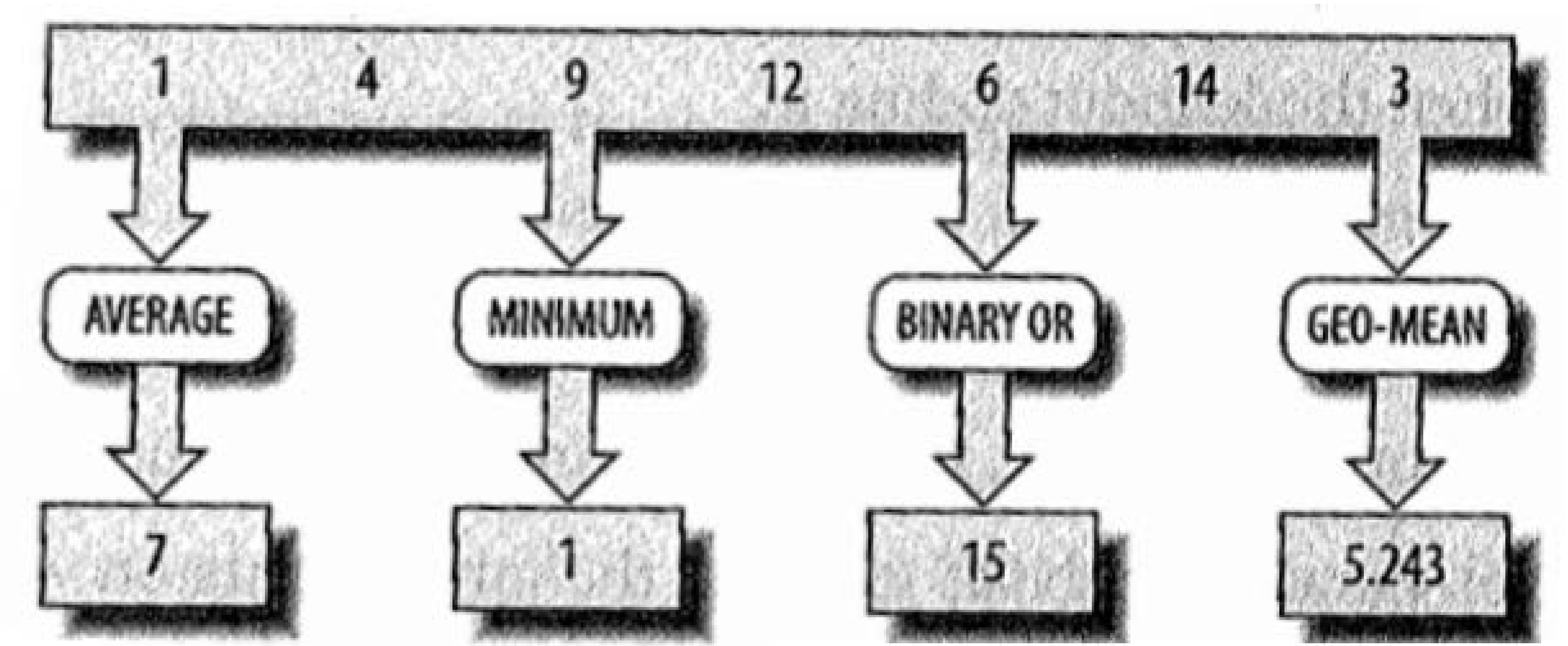
Data parallelism

- Example: convert all characters in an array to upper-case
  - Can divide parts of the data between different tasks and perform the tasks in parallel
  - Key: no dependencies between the tasks that cause their results to be ordered



Task parallelism

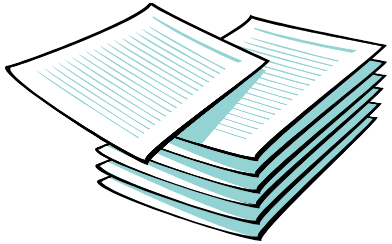
- Example:
  - Several functions on the same data: average, minimum, binary or, geometric mean
  - No dependencies between the tasks, so all can run in parallel



Professor P

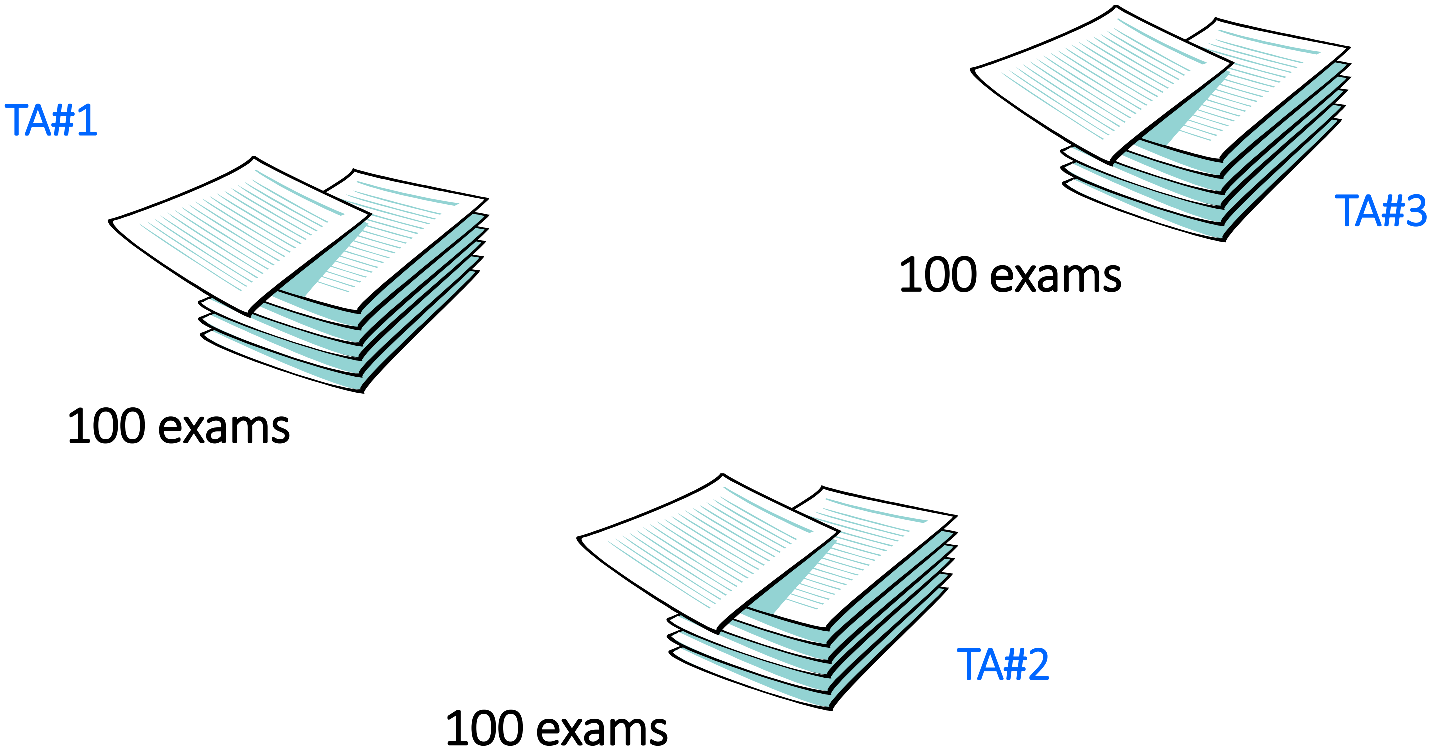
15 questions

300 exams



Professor P's grading assistants

Division of work – data parallelism



Division of work – task parallelism

TA#1

Questions 1 - 5

Questions 6 - 10

TA#2

TA#3

Questions 11 - 15

Coordination

- When the cores can work independently, writing a parallel program is much the same as writing a serial program

- Things get a good deal more complex when the cores need to coordinate their work

Coordination

- **Communication**: one or more cores send their current partial sums to another core

- **Load balancing**: we want the cores all to be assigned roughly the same number of values to compute

- **Synchronization**: we don’t want the other cores to race ahead

What we will see

- We’ll be focusing on learning to write programs that are *explicitly* parallel

- Learn the basics of programming parallel computers using the C/C++ language and three different extensions to C/C++:
  - **Message-Passing Interface** or **MPI**
  - **POSIX threads** or **Pthreads**
  - **OpenMP**

- MPI and Pthreads are libraries of type definitions, functions, and macros that can be used in C programs
- OpenMP consists of a library and some modifications to the C compiler.

What we will see

- Q: Why do we need 3 different extensions to C/C++ instead of just one?

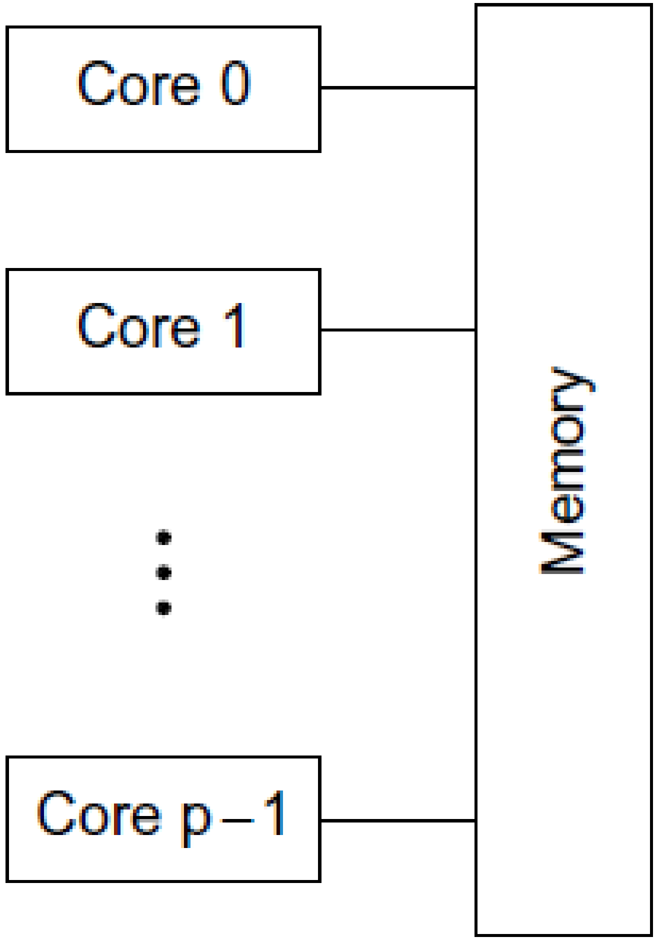
What we will see

- Q: Why do we need 3 different extensions to C/C++ instead of just one?
- A: Don’t forget hardware!!

- There are two main types of parallel systems:
  - **Shared memory** systems
  - **Distributed-memory** systems

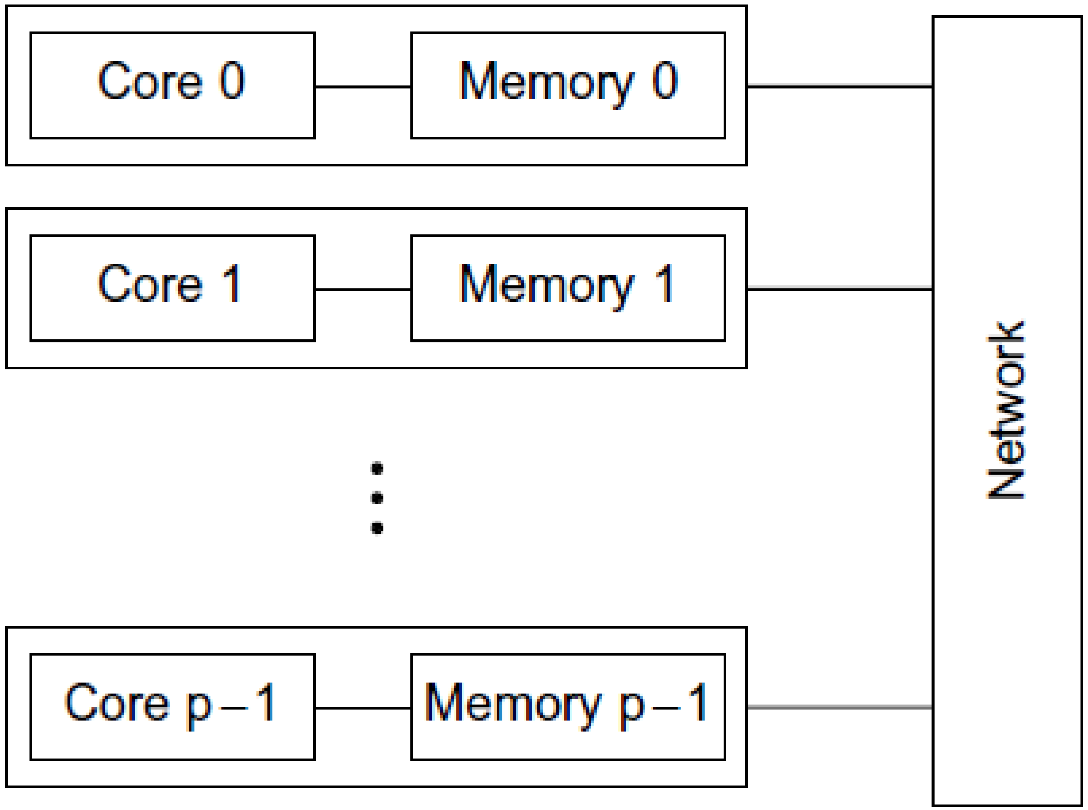
Shared memory

- In a **shared-memory** system, the cores can share access to the computer's memory
    - each core can read and write each memory location
- In a **shared-memory** system, we can coordinate the cores by having them examine and update shared-memory locations
- Pthreads and OpenMP were designed for programming shared-memory systems



Distributed memory

- In a **distributed-memory** system each core has its own, private memory
- Cores must communicate explicitly by doing something like sending messages across a network
- MPI was designed for programming distributed-memory systems.



Concurrent, parallel and distributed computing

- In **concurrent** computing, a program is one in which multiple tasks can be *in progress* at any instant
- In **parallel** computing, a program is one in which multiple tasks *cooperate closely* to solve a problem
- In **distributed** computing, a program may need to cooperate with other programs to solve a problem

Concurrent, parallel and distributed computing

- Parallel and distributed programs are concurrent
- A program such as a multitasking operating system is also concurrent, even when it is run on a machine with only one core, since multiple tasks can be *in progress* at any instant
- A parallel program usually runs multiple tasks simultaneously on cores
- On the other hand, distributed programs tend to be more “loosely coupled.” The tasks may be executed by multiple computers that are separated by large distances

Concluding Remarks (1)

- The laws of physics have brought us to the doorstep of multicore technology.
- Serial programs typically don’t benefit from multiple cores.
- Automatic parallel program generation from serial program code isn’t the most efficient approach to get high performance from multicore computers.

Concluding Remarks (2)

- Learning to write parallel programs involves learning how to coordinate the cores.
- Parallel programs are usually very complex and therefore, require sound program techniques and development.