Parallel Programming

Introduction to Parallel Computing

Slides adapted from the lecture notes by Peter Pacheco

Roadmap

- Why we need ever-increasing performance.
- Why we’re building parallel systems.
- Why we need to write parallel programs.
- How do we write parallel programs?
- Concurrent, parallel, distributed!

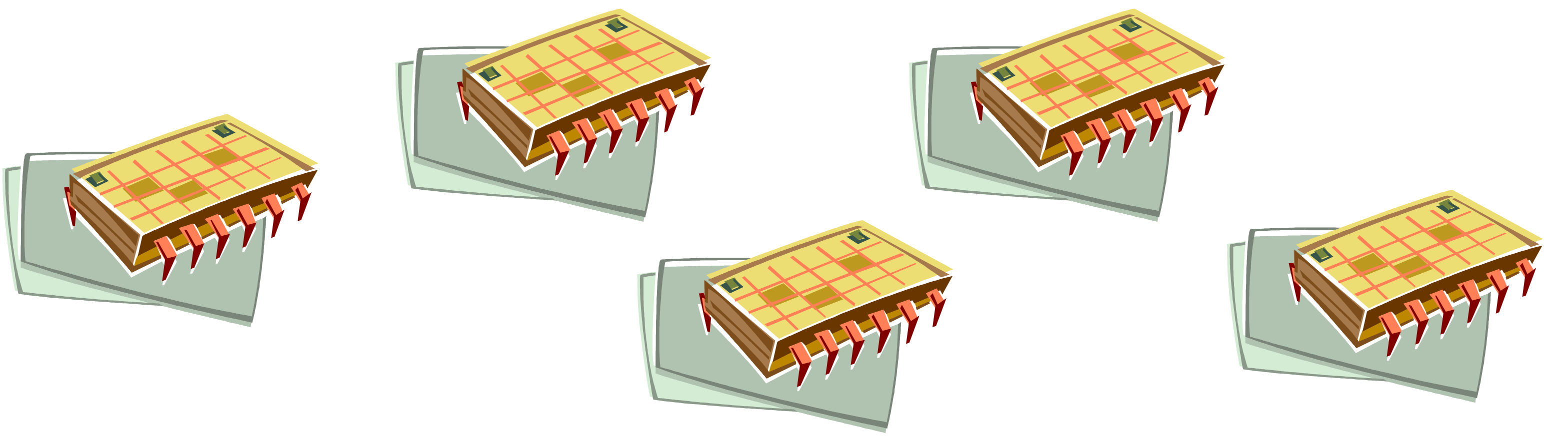
Changing times

- From 1986 – 2002, microprocessors were speeding like a rocket, increasing in performance an average of 50% per year.

- Since then, it’s dropped to about 20% increase per year.

An intelligent solution

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



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.

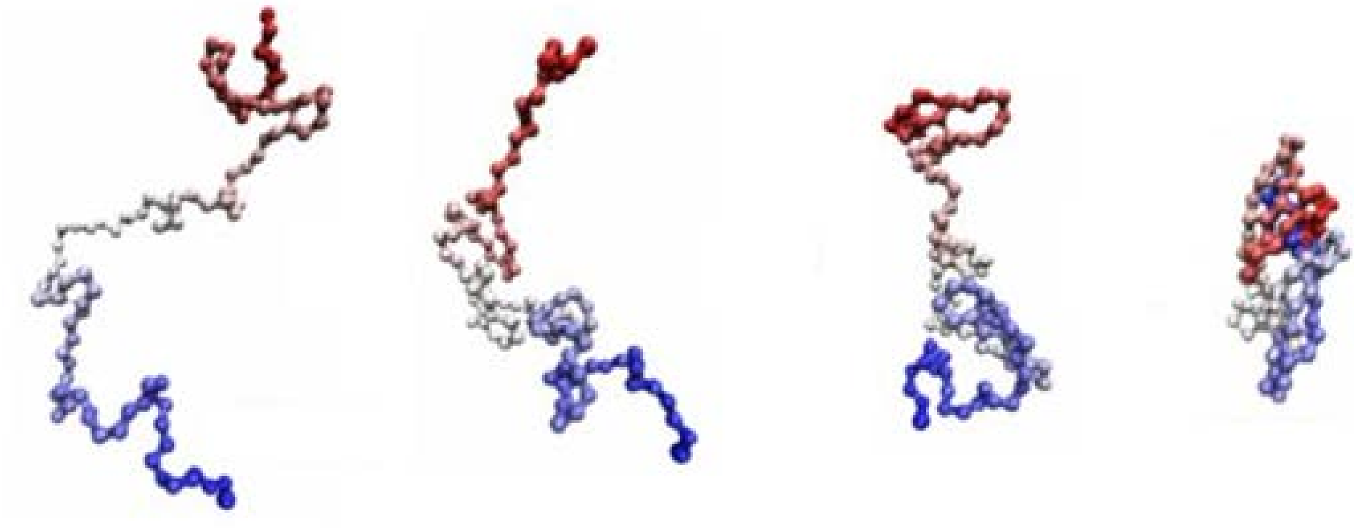
- Serial programs don’t benefit from this approach (in most cases).

Why we need ever-increasing performance

- Computational power is increasing, but so are our computation problems and needs.
- Problems we never dreamed of have been solved, such as decoding the human genome.
- More complex problems are still waiting to be solved.

Climate modeling

Protein folding



Drug discovery

Energy research

Data analysis

Why we’re building parallel systems

- Up to now, performance increases have been attributable to increasing density of transistors.

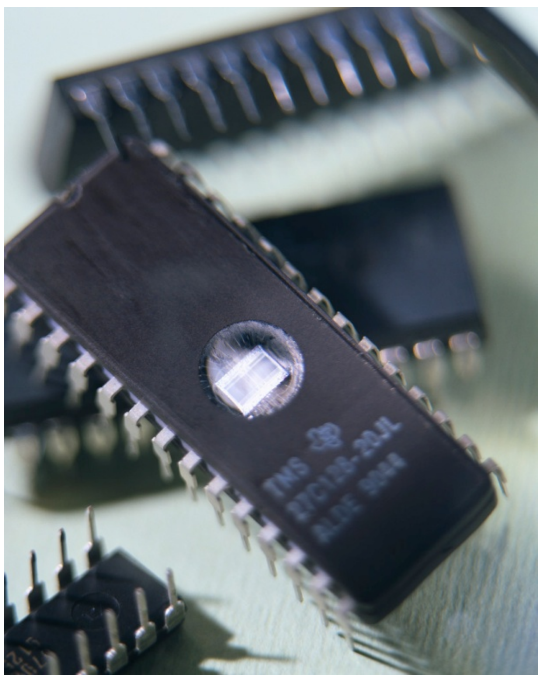
- But there are inherent problems.

Problem

- Denser transistors -> faster processors.
- Faster processors -> increased power consumption.
- Increased power consumption -> increased heat.
- Increased heat -> unreliable processors.

Solution

- Move away from single-core systems to multicore processors.
- “core” = central processing unit (CPU)



- Introducing parallelism!!!

Why we need to write parallel programs

- Running multiple instances of a serial program often isn’t very useful.
- Think of running multiple instances of your favorite game.

- What you really want is for it to run faster.

Approaches to the serial problem

- Rewrite serial programs so that they're parallel.

- Write translation programs that automatically convert serial programs into parallel programs.
  - This is very difficult to do.
  - Success has been limited.

More problems about translation

- Some coding constructs can be recognized by an automatic program generator, and converted to a parallel construct.
- However, it’s likely that the result will be a very inefficient program.
- Sometimes the best parallel solution is to step back and devise an entirely new algorithm.

Example

- Compute n values and add them together.
- Serial solution:

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

Example (cont.)

- We have p cores, p much smaller than n.
- Each core performs 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 its own private variables and executes this block of code independently of the other cores.

Example (cont.)

- After each core completes execution of the code, a private variable my_sum contains the sum of the values computed by its calls to Compute_next_value.

- Ex., 8 cores, n = 24, then the calls to Compute_next_value return:

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

Example (cont.)

- Once all the cores are done computing their private my_sum, they form a global sum by sending results to a designated “master” core which adds the final result.

Example (cont.)

```
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 (cont.)

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

Global sum

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

Core	0	1	2	3	4	5	6	7
my_sum	95	19	7	15	7	13	12	14

Better parallel algorithm

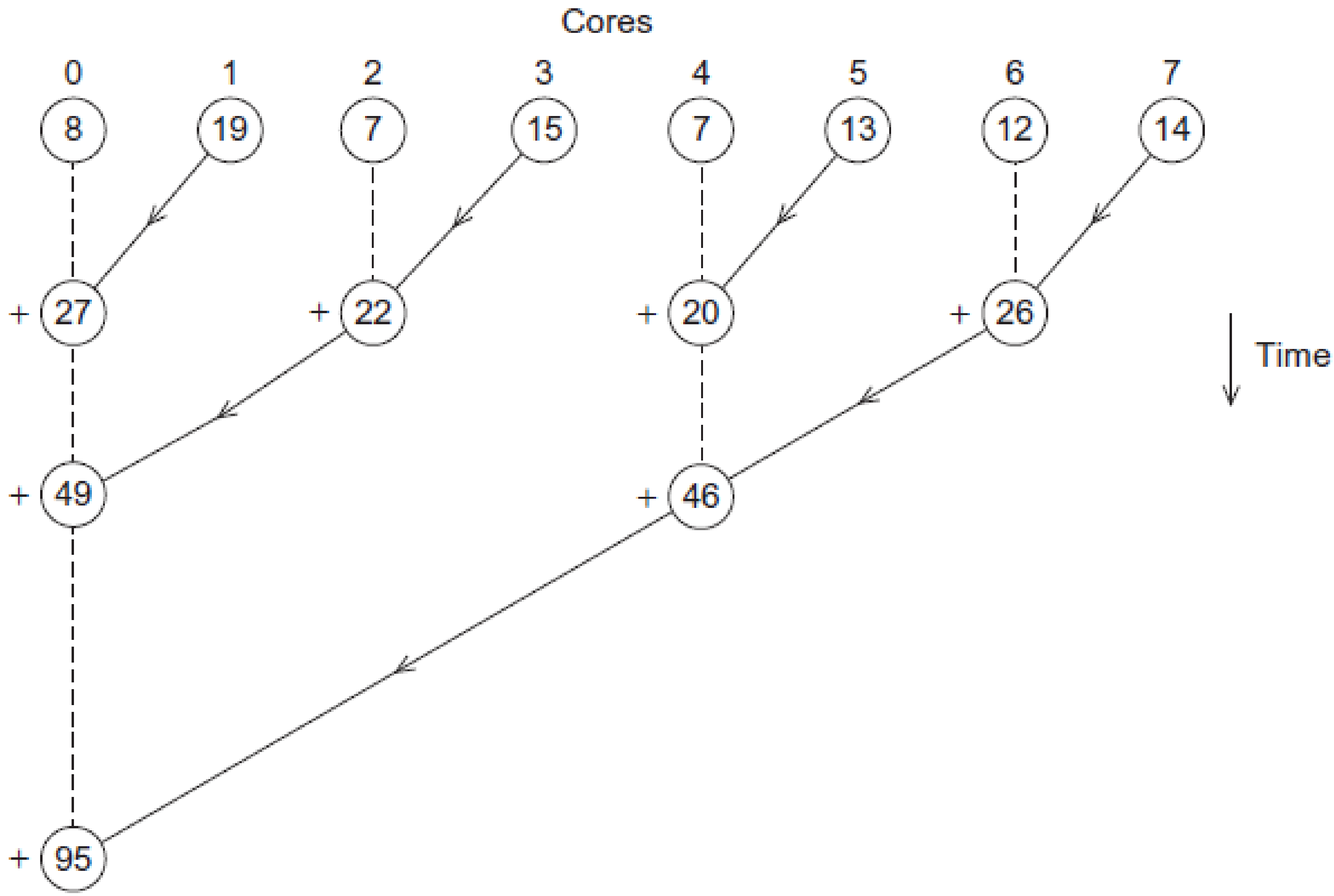
- Don’t make the master core do all the work.
- Share it among the other cores.
- Pair the cores so that core 0 adds its result with core 1’s result.
- Core 2 adds its result with core 3’s result, etc.
- Work with odd and even numbered pairs of cores.

Better parallel algorithm (cont.)

- Repeat the process now with only the evenly ranked cores.
- Core 0 adds result from core 2.
- Core 4 adds the result from core 6, etc.

- Now cores divisible by 4 repeat the process, and so forth, until core 0 has the final result.

Multiple cores forming a global sum



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!

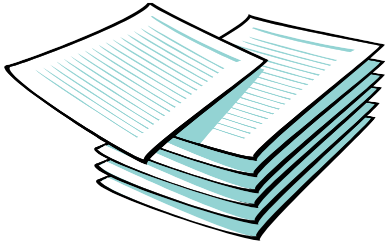
How do we write parallel programs?

- Task parallelism
  - Partition various tasks solving the problem among the cores.
- Data parallelism
  - Partition the data used in solving the problem among the cores.
  - Each core carries out similar operations on its part of the data.

Professor P

15 questions

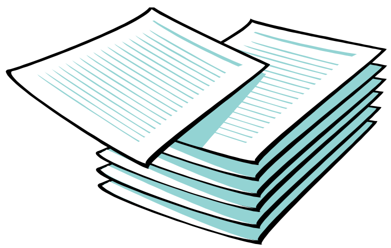
300 exams



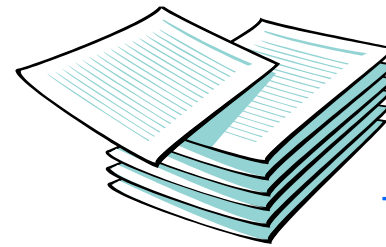
Professor P’s grading assistants

Division of work – data parallelism

TA#1

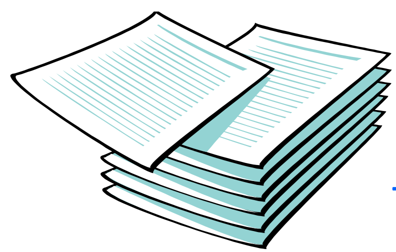


100 exams



TA#3

100 exams



TA#2

100 exams

Division of work – task parallelism

TA#1

TA#3

Questions 11 - 15

Questions 1 - 5

TA#2

Questions 6 - 10

Division of work – data parallelism

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

Division of work – task parallelism

```
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;
}
```

Tasks

1) Receiving
2) Addition

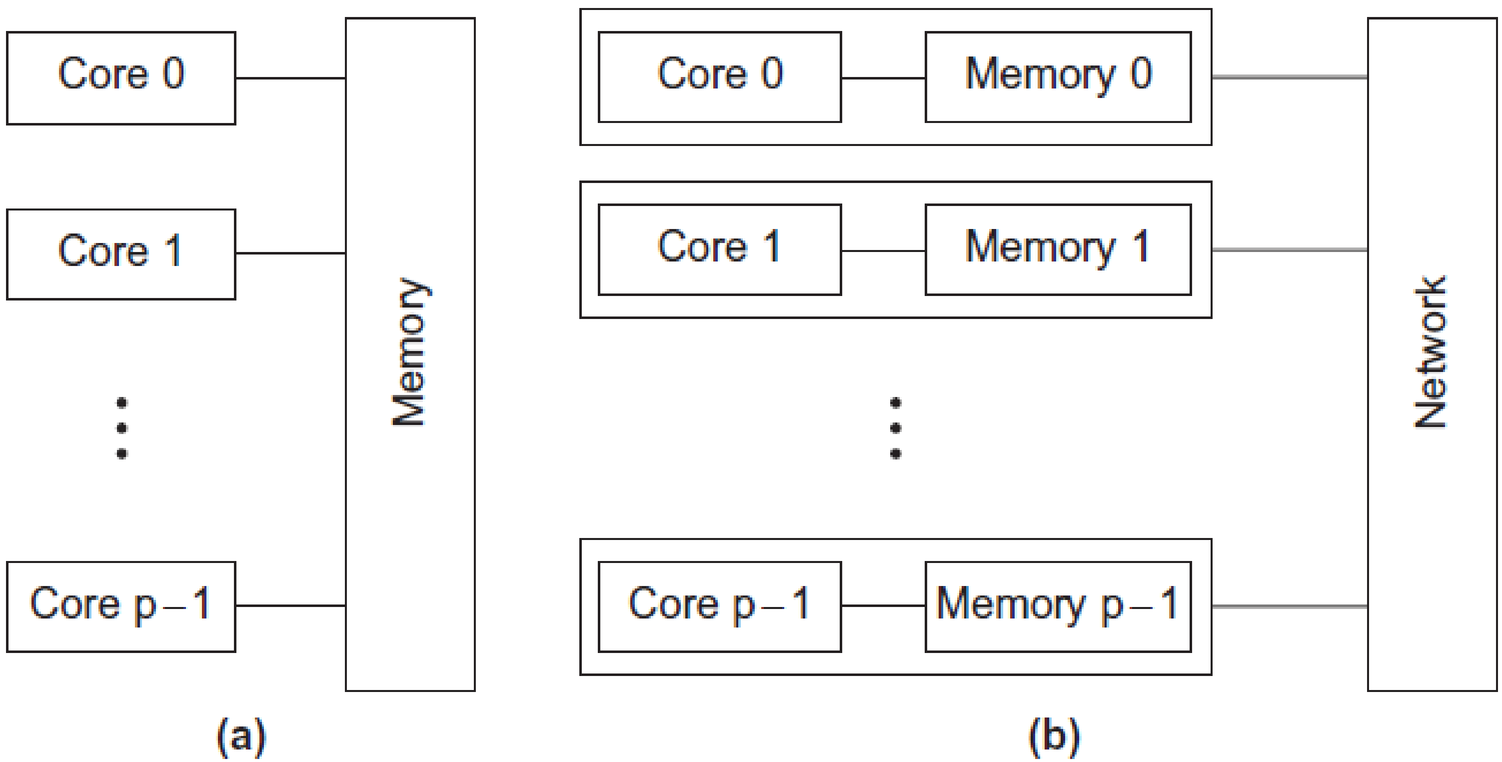
Coordination

- Cores usually need to coordinate their work.
- Communication – one or more cores send their current partial sums to another core.
- Load balancing – share the work evenly among the cores so that one is not heavily loaded.
- Synchronization – because each core works at its own pace, make sure cores do not get too far ahead of the rest.

Type of parallel systems

- Shared-memory
  - The cores can share access to the computer's memory.
  - Coordinate the cores by having them examine and update shared memory locations.
- Distributed-memory
  - Each core has its own, private memory.
  - The cores must communicate explicitly by sending messages across a network.

Type of parallel systems



Shared-memory

Distributed-memory

Terminology

- Concurrent computing – In a program multiple tasks can be in progress at any instant.
- Parallel computing – In a program multiple tasks cooperate closely to solve a problem.
- Distributed computing – A program may need to cooperate with other programs to solve a problem.

Concluding Remarks

- Parallel systems are the trend of computing.
- Serial programs typically don’t benefit from multiple cores.
- 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.