Pipelining vs. Parallel processing

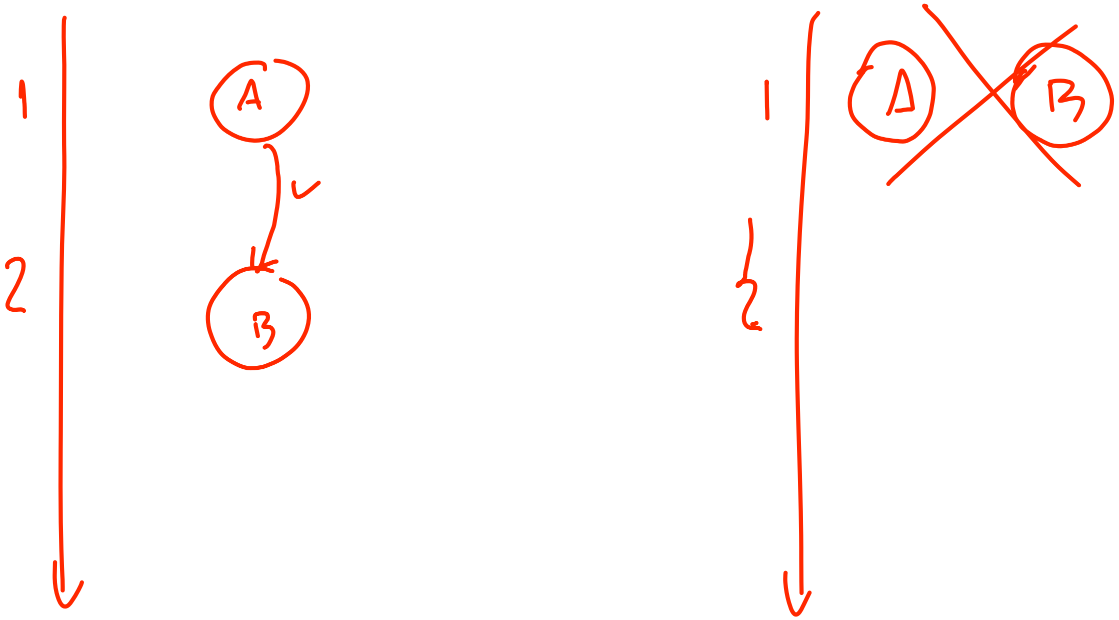
- In both cases, multiple "things" processed by multiple "functional units"

  Pipelining: each thing is broken into a **sequence of pieces**, where each piece is handled by a **different** (specialized) functional unit

  Parallel processing: each thing is processed **entirely** by a single functional unit

- We will briefly introduce the key ideas behind parallel processing
  - instruction level parallelism
  - thread-level parallelism

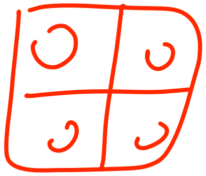
It is all about dependences!



Exploiting Parallelism

- Of the computing problems for which performance is important, many have inherent parallelism

- Best example: computer games
  - Graphics, physics, sound, AI etc. can be done separately
  - Furthermore, there is often parallelism within each of these:
    - Each pixel on the screen's color can be computed independently
    - Non-contacting objects can be updated/simulated independently
    - Artificial intelligence of non-human entities done independently



- Another example: Google queries
  - Every query is independent
  - Google is read-only!!

Parallelism at the Instruction Level

```
         w              r
add $2 <- $3, $6
     w         R
or $2 <- $2, $4
      w
lw $6 <- 0($4)
              r
addi $7 <- $6, 0x5
sub $8 <- $8, $4
```

$6 ←

Dependences?
RAW ←
WAW ←
WAR

When can we reorder instructions?

obey dependences,

When should we reorder instructions?

→ fixing latency

→ exploit parallelism

- •
- •

Surperscalar Processors:
Multiple instructions executing in parallel at *same* stage

```
add $2 <- $3, $6
or $5 <- $2, $4
lw $6 <- 0($4)
sub $8 <- $8, $4
addi $7 <- $6, 0x5
```

OoO Execution Hardware

Fetch/Decode

EX

x

I_1

I_2

I_3

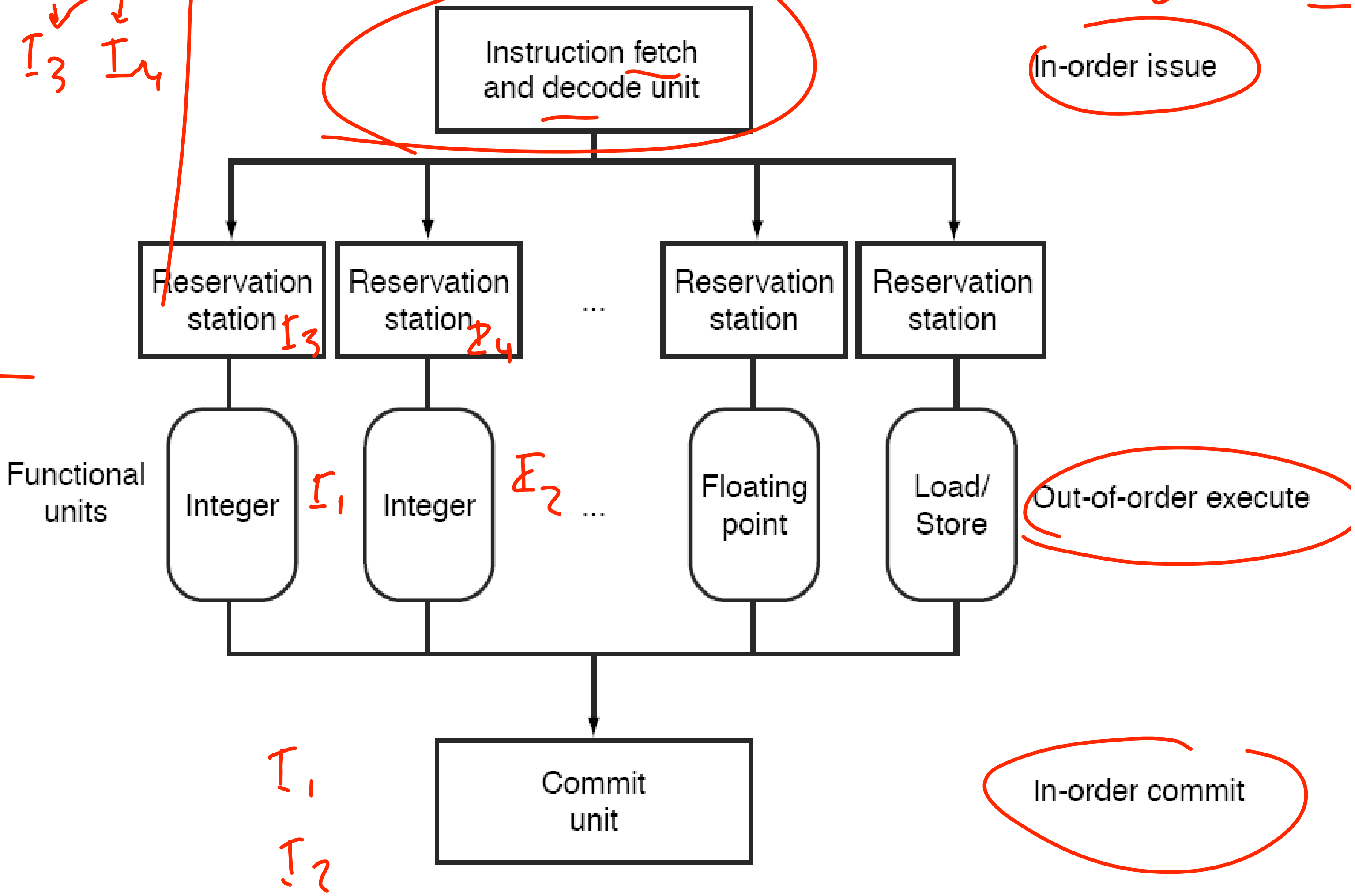
I_4

I_1 I_2

I_3 I_4

compile

depen



Instruction fetch and decode unit

Reservation station

Reservation station

...

Reservation station

Reservation station

Functional units

Integer

Integer

...

Floating point

Load/ Store

Commit unit

In-order issue

Out-of-order execute

In-order commit

I_3

I_4

I_1

I_2

I_1

I_2

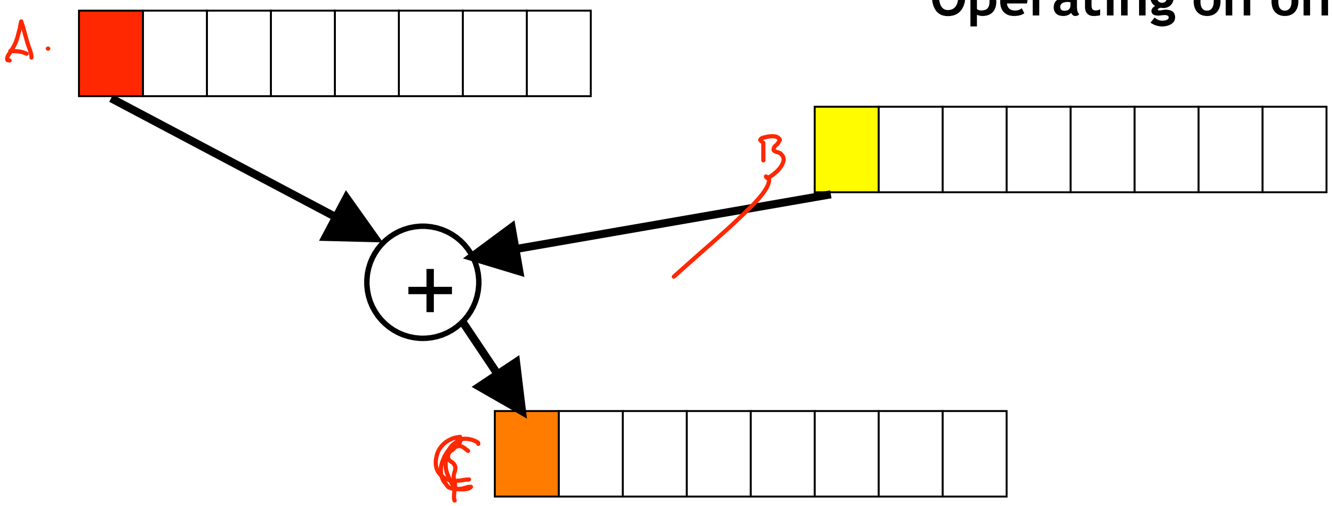
I_2

Exploiting Parallelism at the Data Level

- Consider adding together two arrays:

```
void
array_add(int A[], int B[], int C[], int length) {
   int i;
   for (i = 0 ; i < length ; ++ i) {
    C[i] = A[i] + B[i];
   }
}
```

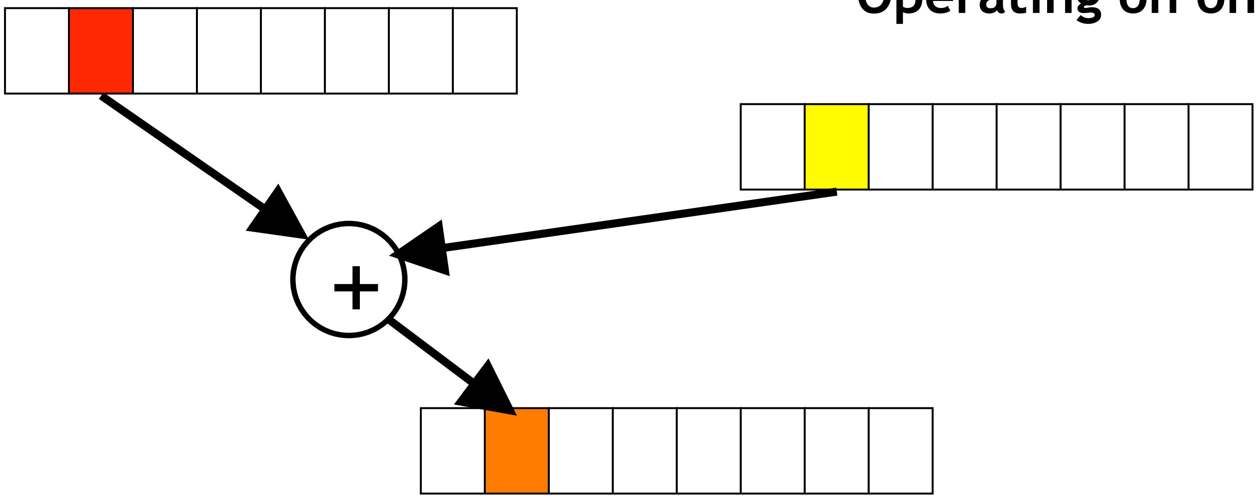
Operating on one element at a time



Exploiting Parallelism at the Data Level

- Consider adding together two arrays:

```
void
array_add(int A[], int B[], int C[], int length) {
  int i;
  for (i = 0 ; i < length ; ++ i) {
   C[i] = A[i] + B[i];
  }
}
```



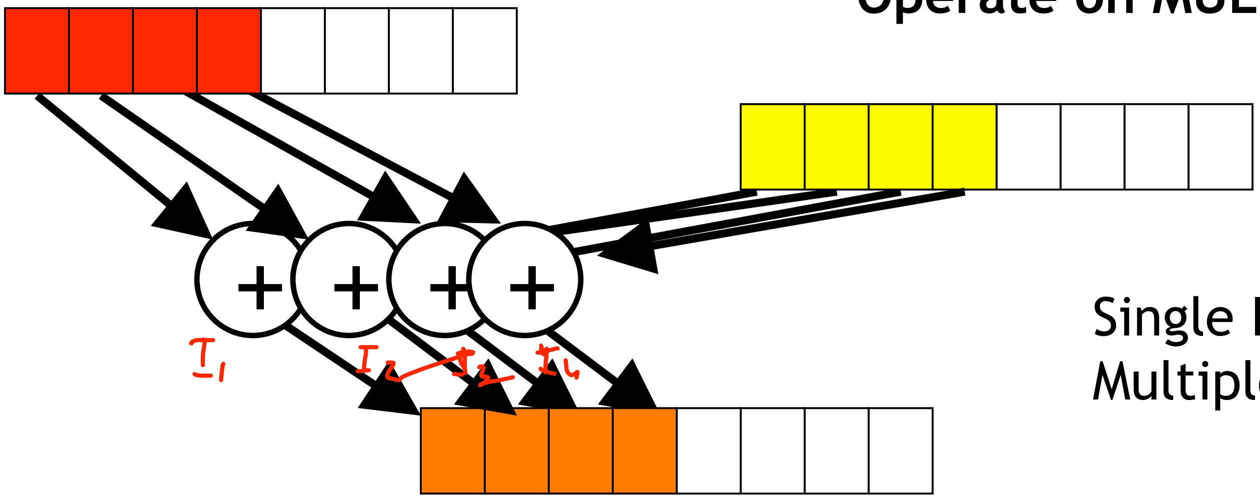
Operating on one element at a time

Exploiting Parallelism at the Data Level (SIMD)

- Consider adding together two arrays:

```
void
array_add(int A[], int B[], int C[], int length) {
  int i;
  for (i = 0 ; i < length ; ++ i) {
   C[i] = A[i] + B[i];
  }
}
```

Operate on MULTIPLE elements



I_1

I_2

I_3

I_4

Single Instruction,
Multiple Data (**SIMD**)

Intel SSE/SSE2 as an example of SIMD

- Added new 128 bit registers (XMM0 – XMM7), each can store
  - 4 single precision FP values (SSE) 4 * 32b
  - 2 double precision FP values (SSE2) 2 * 64b
  - 16 byte values (SSE2) 16 * 8b
  - 8 word values (SSE2) 8 * 16b
  - 4 double word values (SSE2) 4 * 32b
  - 1 128-bit integer value (SSE2) 1 * 128b

	4.0 (32 bits)	4.0 (32 bits)	3.5 (32 bits)	-2.0 (32 bits)
+	-1.5 (32 bits)	2.0 (32 bits)	1.7 (32 bits)	2.3 (32 bits)
	2.5 (32 bits)	6.0 (32 bits)	5.2 (32 bits)	0.3 (32 bits)

Is it always that easy?

- Not always… a more challenging example:

```
unsigned
sum_array(unsigned *array, int length) {
  int total = 0;
  for (int i = 0 ; i < length ; ++ i) {
         total += array[i];
  }
  return total;
}
```

- Is there parallelism here?

We first need to restructure the code

```
unsigned
sum_array2(unsigned *array, int length) {
  unsigned total, i;
  unsigned temp[4] = {0, 0, 0, 0};
  for (i = 0 ; i < length & ~0x3 ; i += 4) {
    temp[0] += array[i];
    temp[1] += array[i+1];
    temp[2] += array[i+2];
    temp[3] += array[i+3];
  }
  total = temp[0] + temp[1] + temp[2] + temp[3];
  for ( ; i < length ; ++ i) {
    total += array[i];
  }
  return total;
}
```

0011

1100

6

xx

00

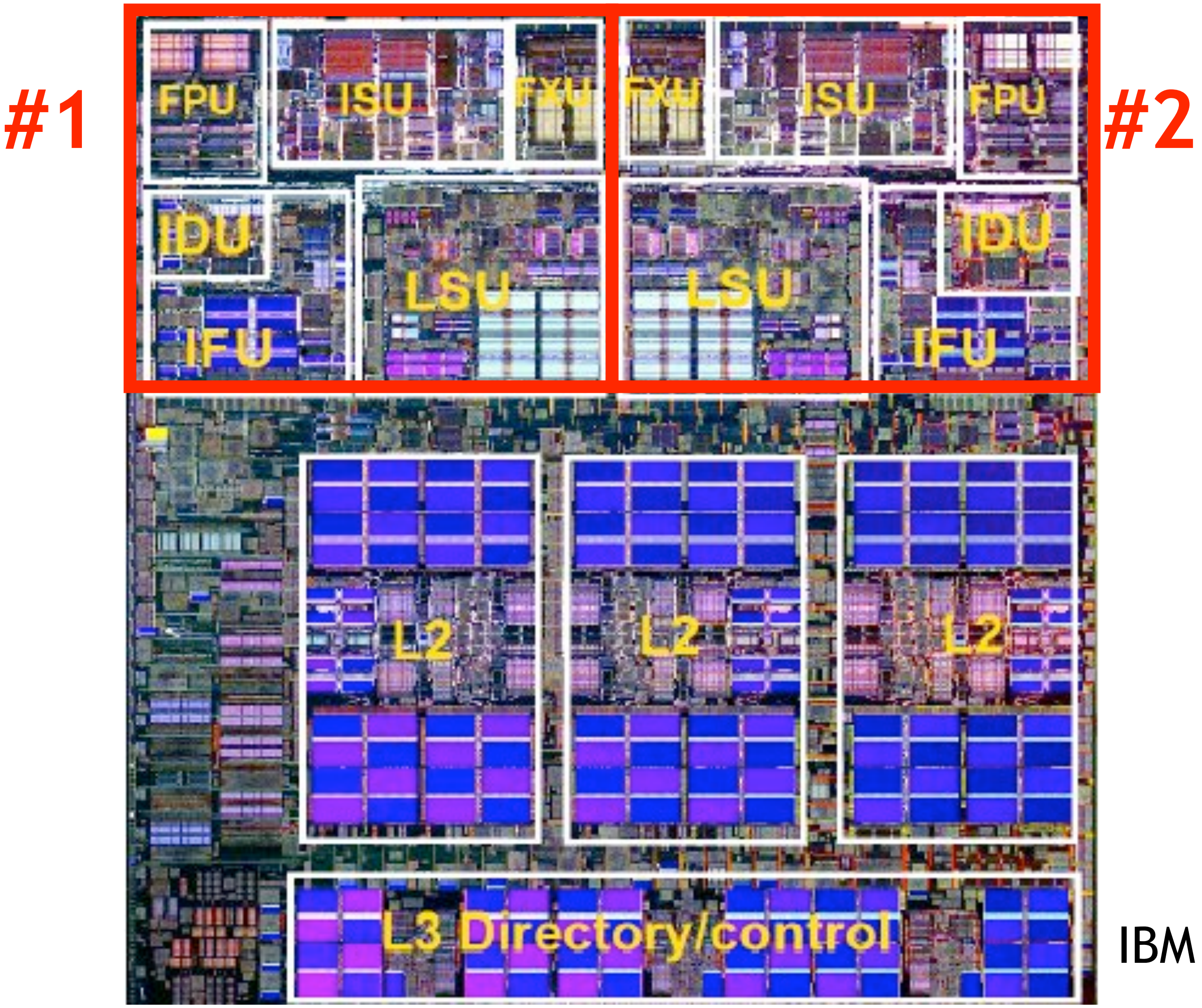
≤ 3

Then we can write SIMD code for the hot part

```
unsigned
sum_array2(unsigned *array, int length) {
  unsigned total, i;
  unsigned temp[4] = {0, 0, 0, 0};
  for (i = 0 ; i < length & ~0x3 ; i += 4) {
    temp[0] += array[i];
    temp[1] += array[i+1];
    temp[2] += array[i+2];
    temp[3] += array[i+3];
  }
  total = temp[0] + temp[1] + temp[2] + temp[3];
  for ( ; i < length ; ++ i) {
    total += array[i];
  }
  return total;
}
```

Thread level parallelism: Multi-Core Processors

- Two (or more) complete processors, fabricated on the same silicon chip
- Execute instructions from two (or more) programs/threads at same time



IBM Power5

Multi-Cores are Everywhere

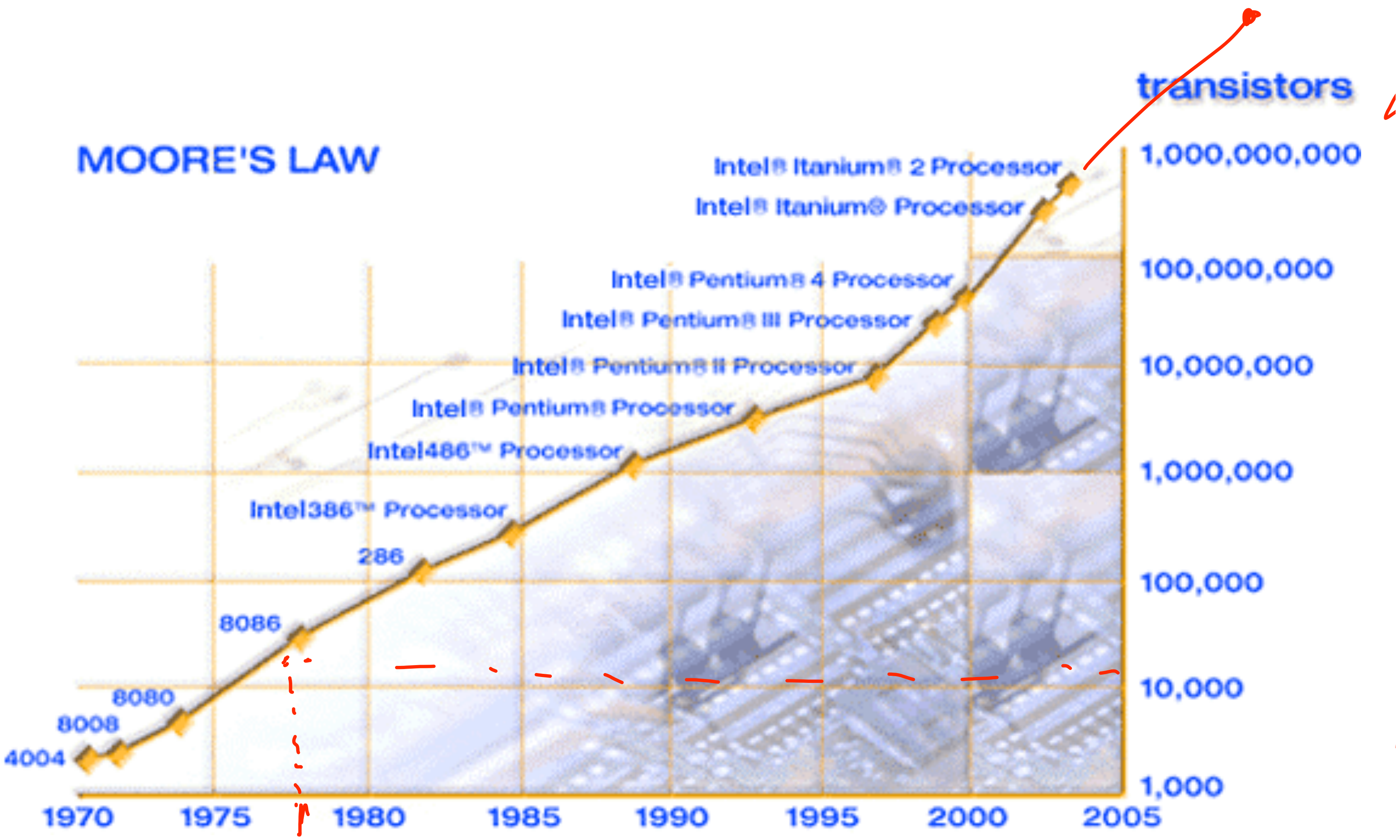
Intel Core Duo in new Macs: 2 x86 processors on same chip

XBox360: 3 PowerPC cores

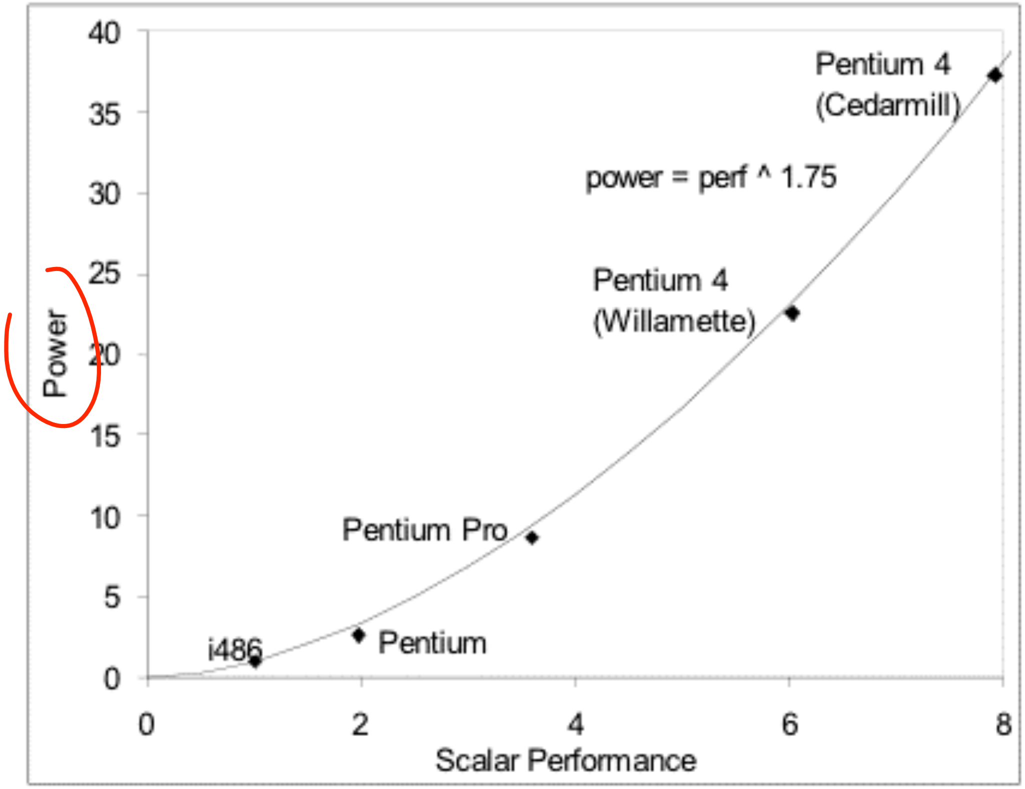
Sony Playstation 3: Cell processor, an asymmetric multi-core with 9 cores (1 general-purpose, 8 special purpose SIMD processors)

Why Multi-cores Now?

- Number of transistors we can put on a chip growing exponentially…



… and performance growing too…

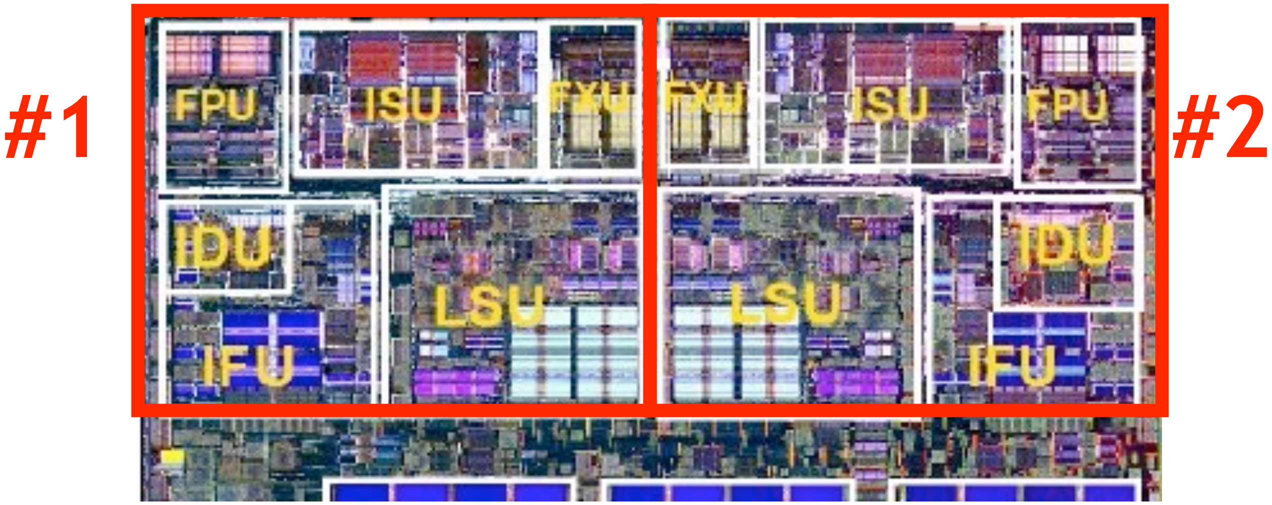


- But power is growing even faster!!
  - Power has become limiting factor in current chips

As programmers, do we care?

- What happens if we run a program on a multi-core?

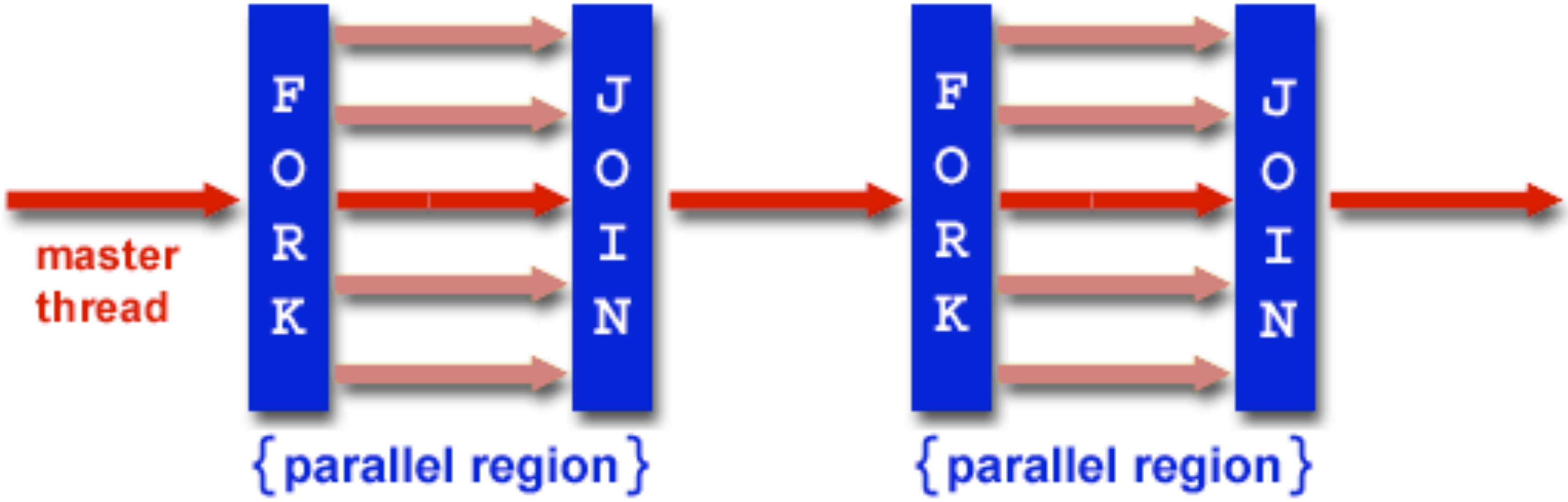
```
void
array_add(int A[], int B[], int C[], int length) {
  int i;
  for (i = 0 ; i < length ; ++i) {
   C[i] = A[i] + B[i];
  }
}
```



What if we want a program to run on both processors

- We have to explicitly tell the machine exactly how to do this
  - This is called parallel programming or concurrent programming

- There are many parallel/concurrent programming models
  - We will look at a relatively simple one: **fork-join parallelism**
  - Posix threads and explicit synchronization

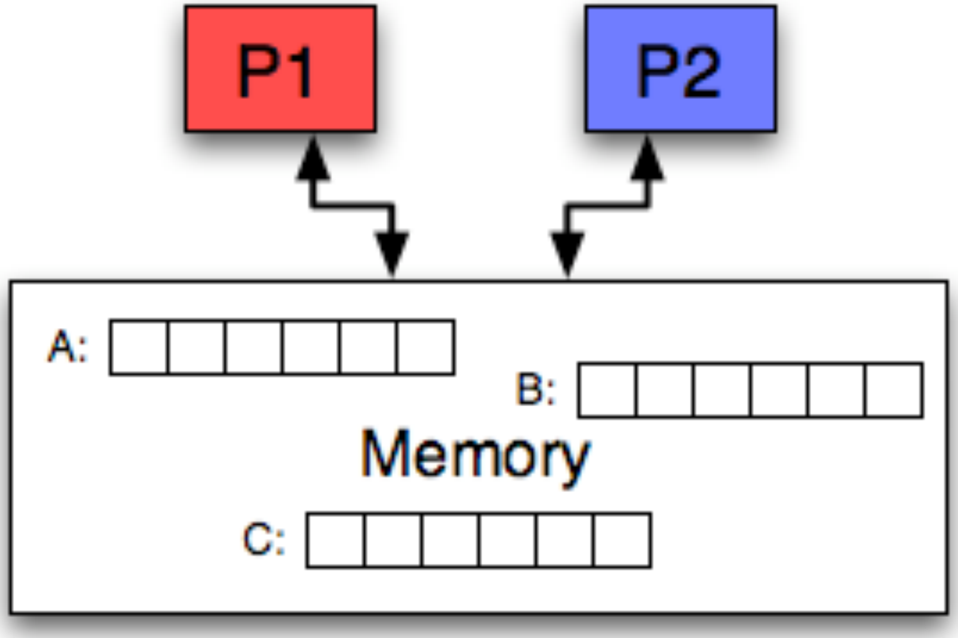


Fork/Join Logical Example

- Fork N-1 threads
- Break work into N pieces (and do it)
- Join (N-1) threads

```
void
array_add(int A[], int B[], int C[], int length) {
    cpu_num = fork(N-1);
   int i;
   for (i = cpu_num ; i < length ; i += N) {
     C[i] = A[i] + B[i];
   }
    join();
}
```

How good is this with caches?

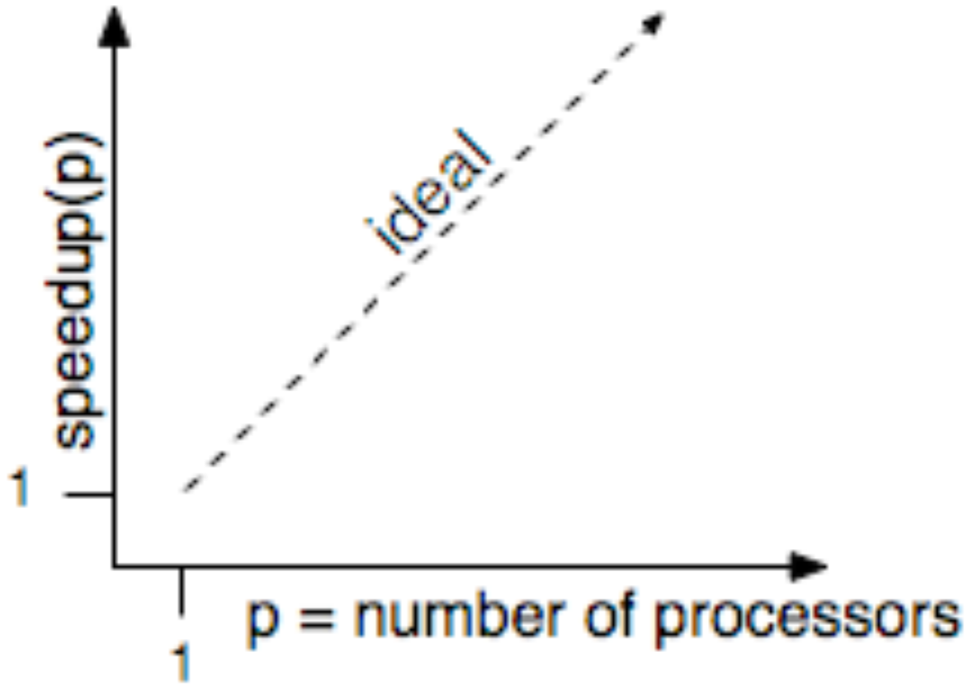


How does this help performance?

- Parallel **speedup** measures improvement from parallelization:

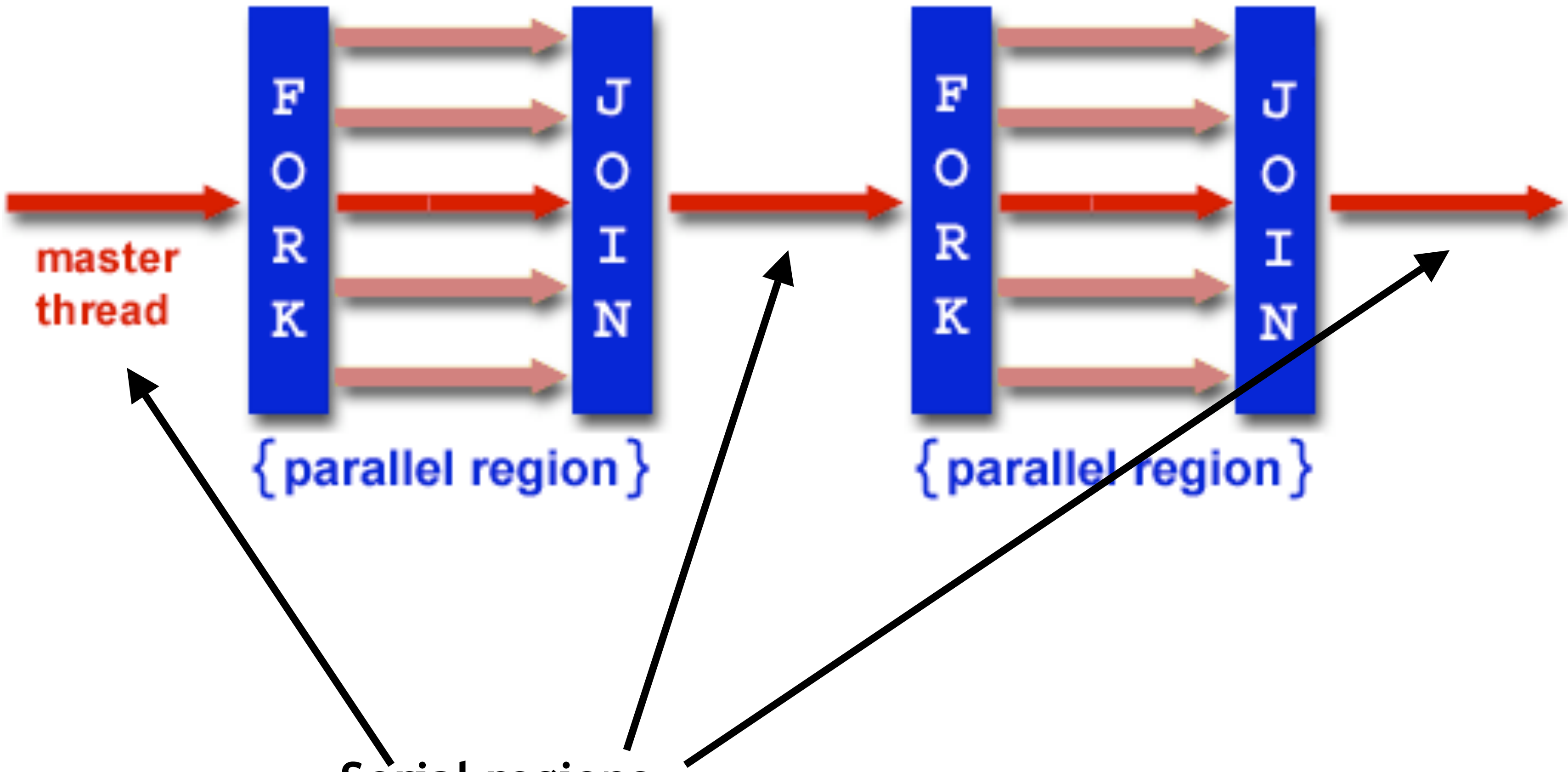
$$\text{speedup}(\mathbf{p}) = \frac{\text{time for best serial version}}{\text{time for version with } \mathbf{p} \text{ processors}}$$

- What can we realistically expect?



Reason #1: Amdahl’s Law

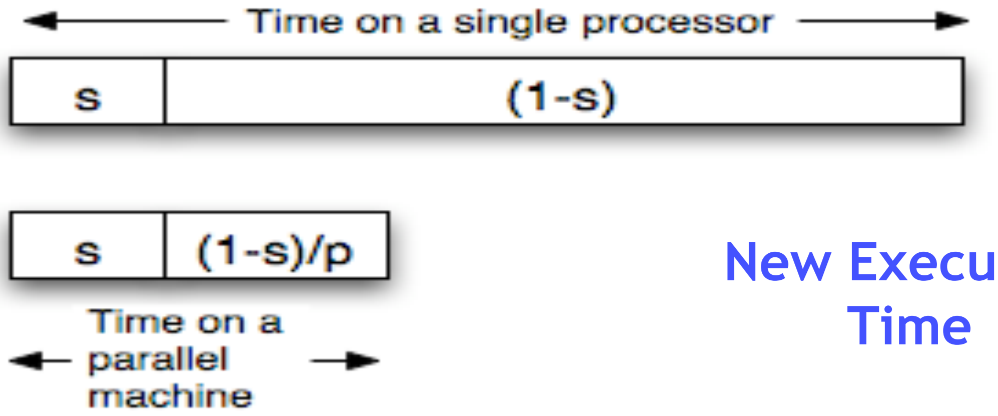
- In general, the whole computation is not (easily) parallelizable



Serial regions

Reason #1: Amdahl's Law

- Suppose a program takes 1 unit of time to execute serially
- A fraction of the program, **s**, is inherently serial (unparallelizable)



$$\text{New Execution Time} = \frac{1-s}{P} + s$$

- For example, consider a program that, when executing on one processor, spen
  10% of its time in a non-parallelizable region. How much faster will this progra
  run on a 3-processor system?

$$\textbf{New Execution Time} = \frac{.9T}{3} + .1T =$$

Speedup =

- What is the maximum speedup from parallelization?

Reason #2: Overhead

```
void
array_add(int A[], int B[], int C[], int length) {
   cpu_num = fork(N-1);
  int i;
  for (i = cpu_num ; i < length ; i += N) {
    C[i] = A[i] + B[i];
  }
   join();
}
```

- Forking and joining is not instantaneous
  - Involves communicating between processors
  - May involve calls into the operating system
    - Depends on the implementation

$$\text{New Execution Time} = \frac{1-s}{P} + s + \text{overhead}(P)$$

Programming Explicit Thread-level Parallelism

- As noted previously, the programmer must specify how to parallelize
- But, want path of least effort

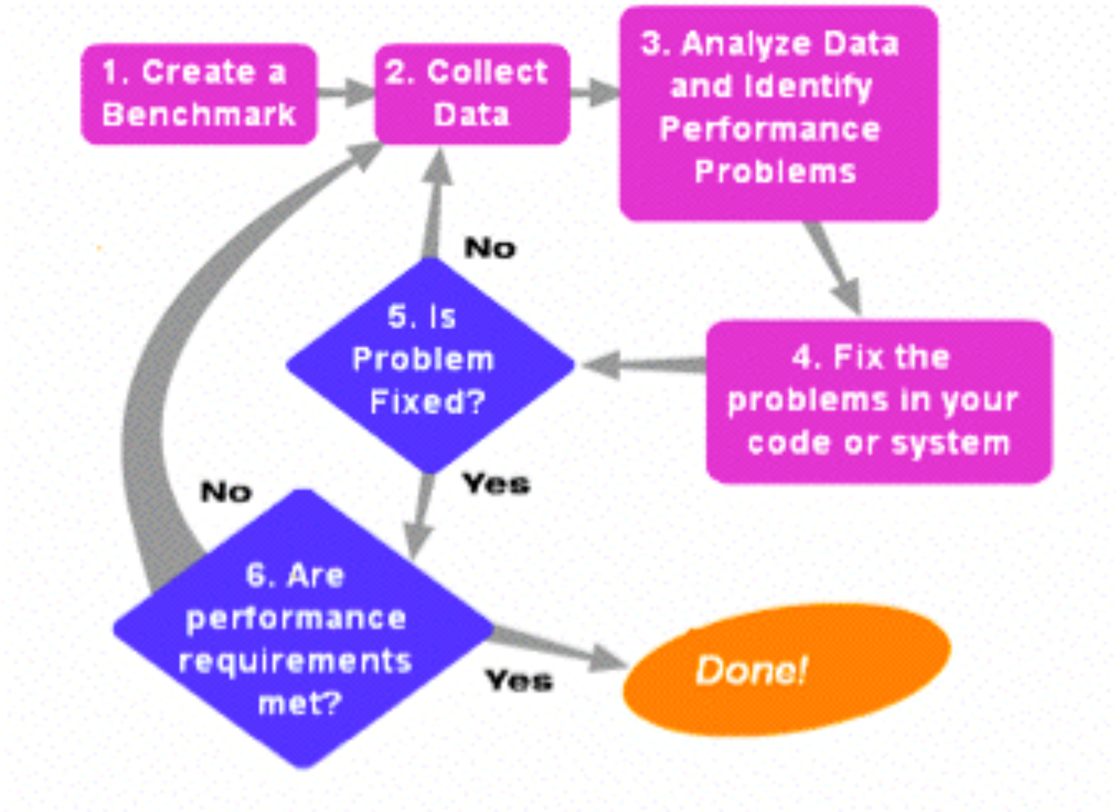
- Division of labor between the **Human** and the **Compiler**
  - **Humans: good at expressing parallelism**, bad at bookkeeping
  - **Compilers:** bad at finding parallelism, **good at bookkeeping**

- Want a way to take serial code and say “Do this in parallel!” without:
  - Having to manage the synchronization between processors
  - Having to know a priori how many processors the system has
  - Deciding exactly which processor does what
  - Replicate the private state of each thread

- OpenMP: an industry standard set of compiler extensions
  - Works very well for programs with structured parallelism.

Performance Optimization

- Until you are an expert, first write a working version of the program
- Then, and only then, begin tuning, first collecting data, and iterate
  - Otherwise, you will likely optimize what doesn't matter



"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." -- *Sir Tony Hoare*

Summary

- Multi-core is having more than one processor on the same chip.
  - Soon most PCs/servers and game consoles will be multi-core
  - Results from Moore's law and power constraint

- Exploiting multi-core requires parallel programming
  - Automatically extracting parallelism too hard for compiler, in general
  - But, can have compiler do much of the bookkeeping for us
  - OpenMP

- Fork-Join model of parallelism
  - At parallel region, fork a bunch of threads, do the work in parallel, an then join, continuing with just one thread
  - Expect a speedup of less than P on P processors
    - Amdahl's Law: speedup limited by serial portion of program
    - Overhead: forking and joining are not free