Parallelism

Content based upon Dr. Bernhard Scholz

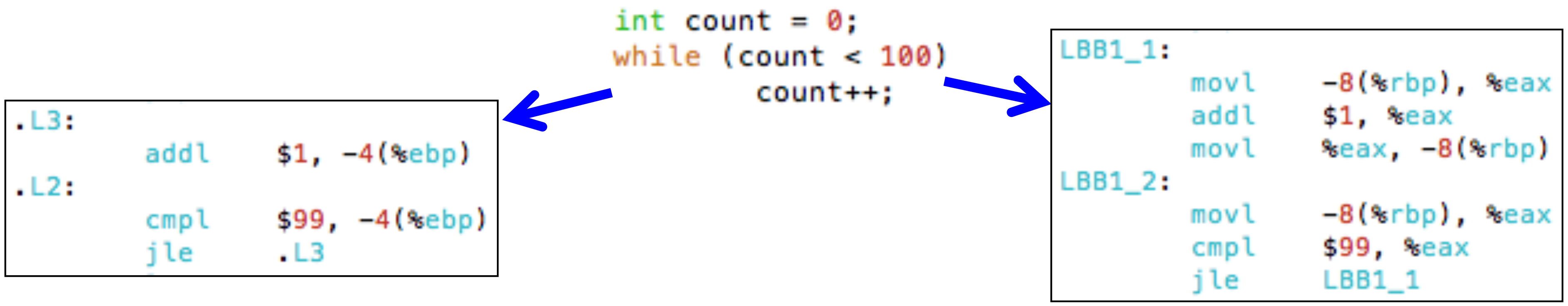
Outline

- Introduction to parallelism
  - Software development
  - Hardware development
- Forms of parallelism
  - Task-parallelism
  - Data-parallelism

Early Practices for Writing software

1960s and 1970s

- Hard coded instructions (machine language) is too difficult
- Abstract software constructs makes for easier development (hex codes -> C/Fortran)
- Creates portability, without losing performance.
- Abstractions assume uni-processor view
  - One flow of control
  - One memory image
  - Compiler does the work to allocate of abstract operations to low level hardware capabilities



```
int count = 0;
while (count < 100)
        count++;
```

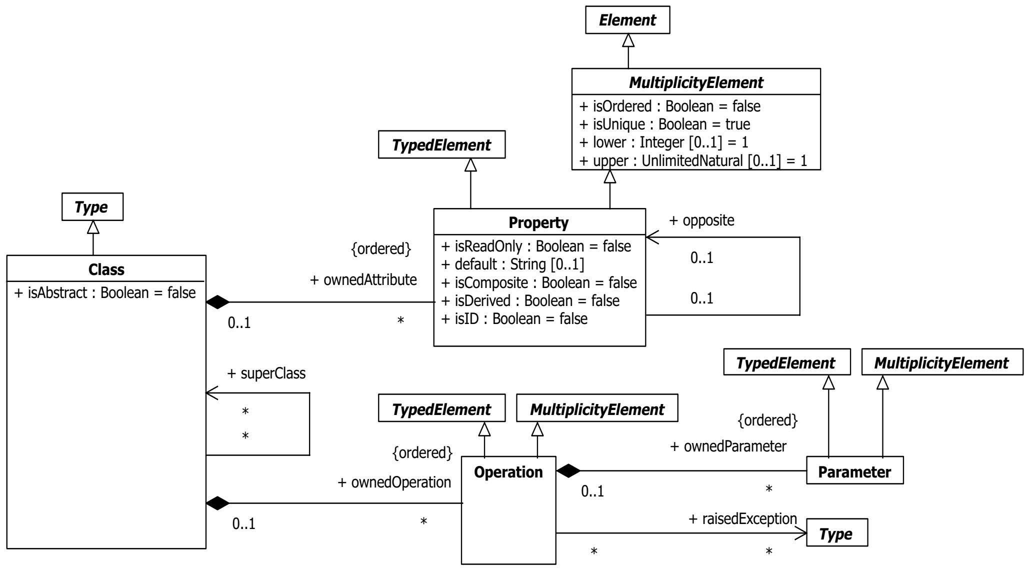
```
.L3:
        addl    $1, -4(%ebp)
.L2:
        cmpl    $99, -4(%ebp)
        jle     .L3
```

```
LBB1_1:
        movl    -8(%rbp), %eax
        addl    $1, %eax
        movl    %eax, -8(%rbp)
LBB1_2:
        movl    -8(%rbp), %eax
        cmpl    $99, %eax
        jle     LBB1_1
```

2nd Iteration: Complex software

1980s and 1990s

- Millions of lines of code
- Large software teams
- Higher levels of abstraction for composability, malleability and maintainability.
  - OO Design, IDE, Components, Frameworks, languages C++, C#, Java
- Trade a little performance for these levels of abstraction



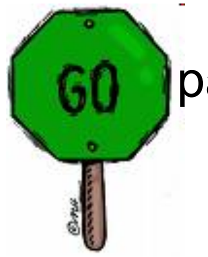
OMG UML

Class Diagram

The Parallel Programming Gap

- Time frame: 2005 to 20??
- Problem: no more performance gains for sequential programs. (see next slides).

- We need continuous and reasonable performance improvements
  - to support new software features
  - to process larger data-sets



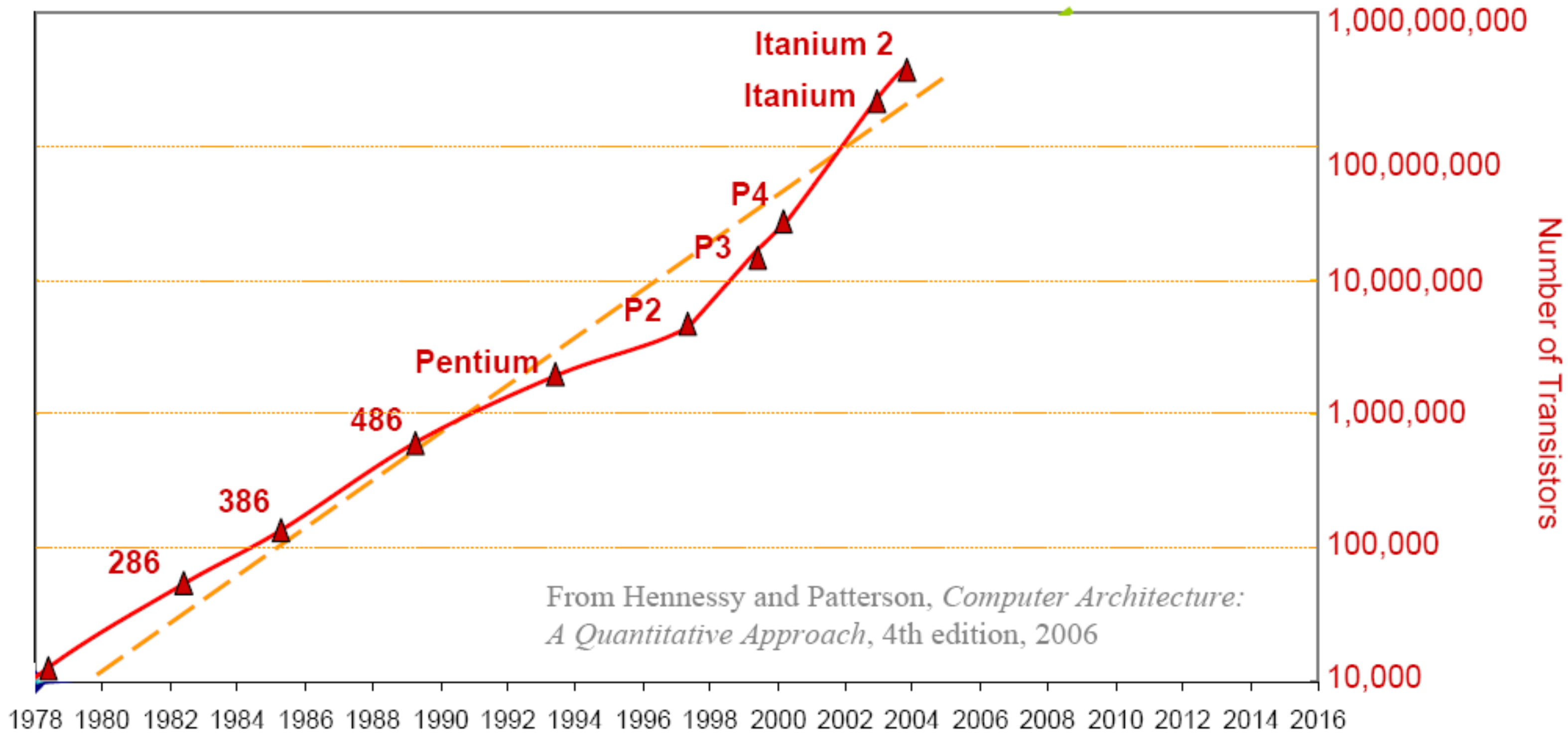
parallel, but...

- ➢ We need to keep portability, malleability and maintainability.
- ➢ We do not want to increase complexity faced by the programmer.

Moore’s Law

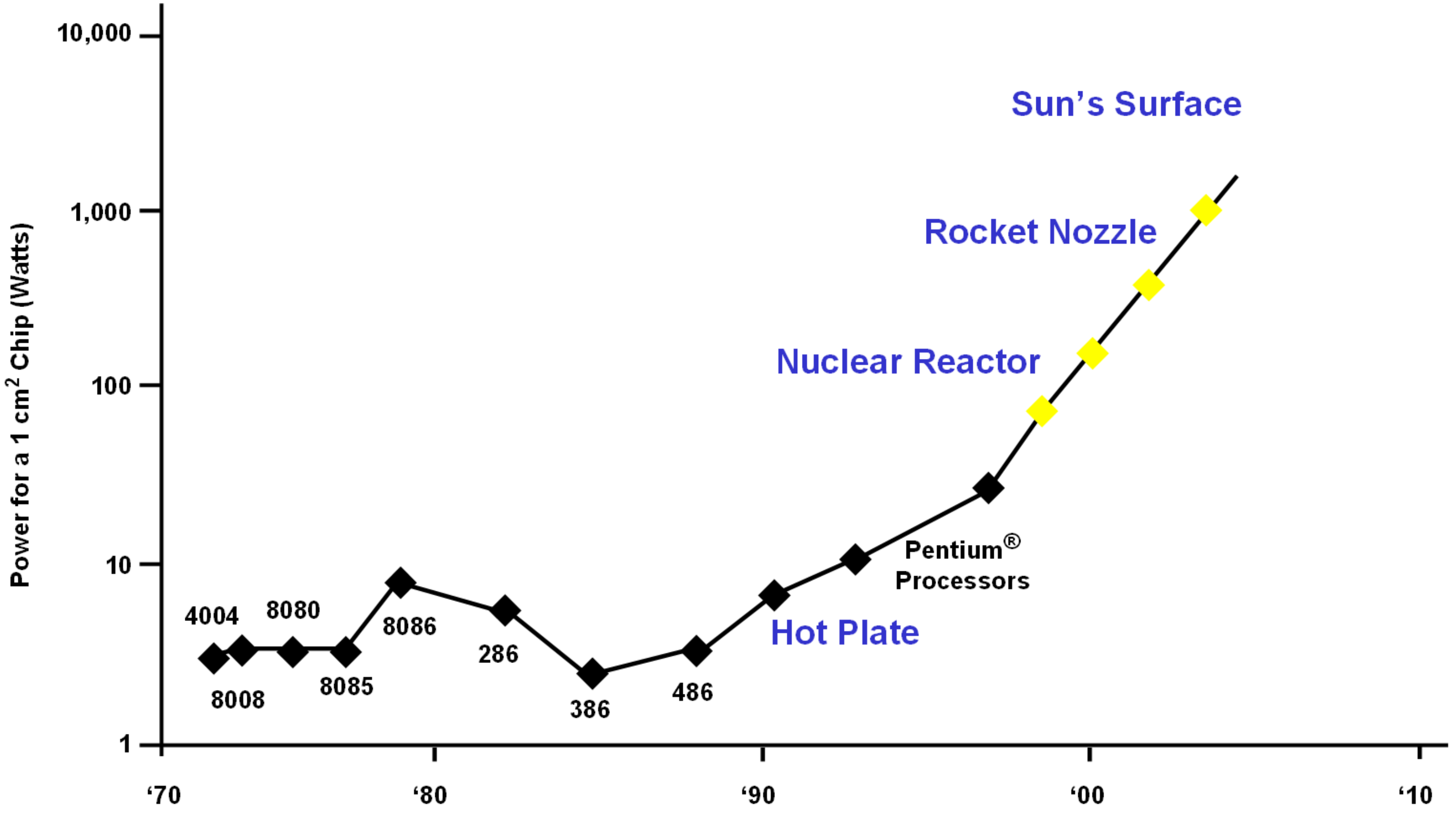
- Gordon Earle Moore, co-founder of Intel Cooperation, stated in an article published in Electronics Magazine in 1965, that

  “*the number of transistors that can be placed on an integrated circuit is doubling approximately every two years*”.

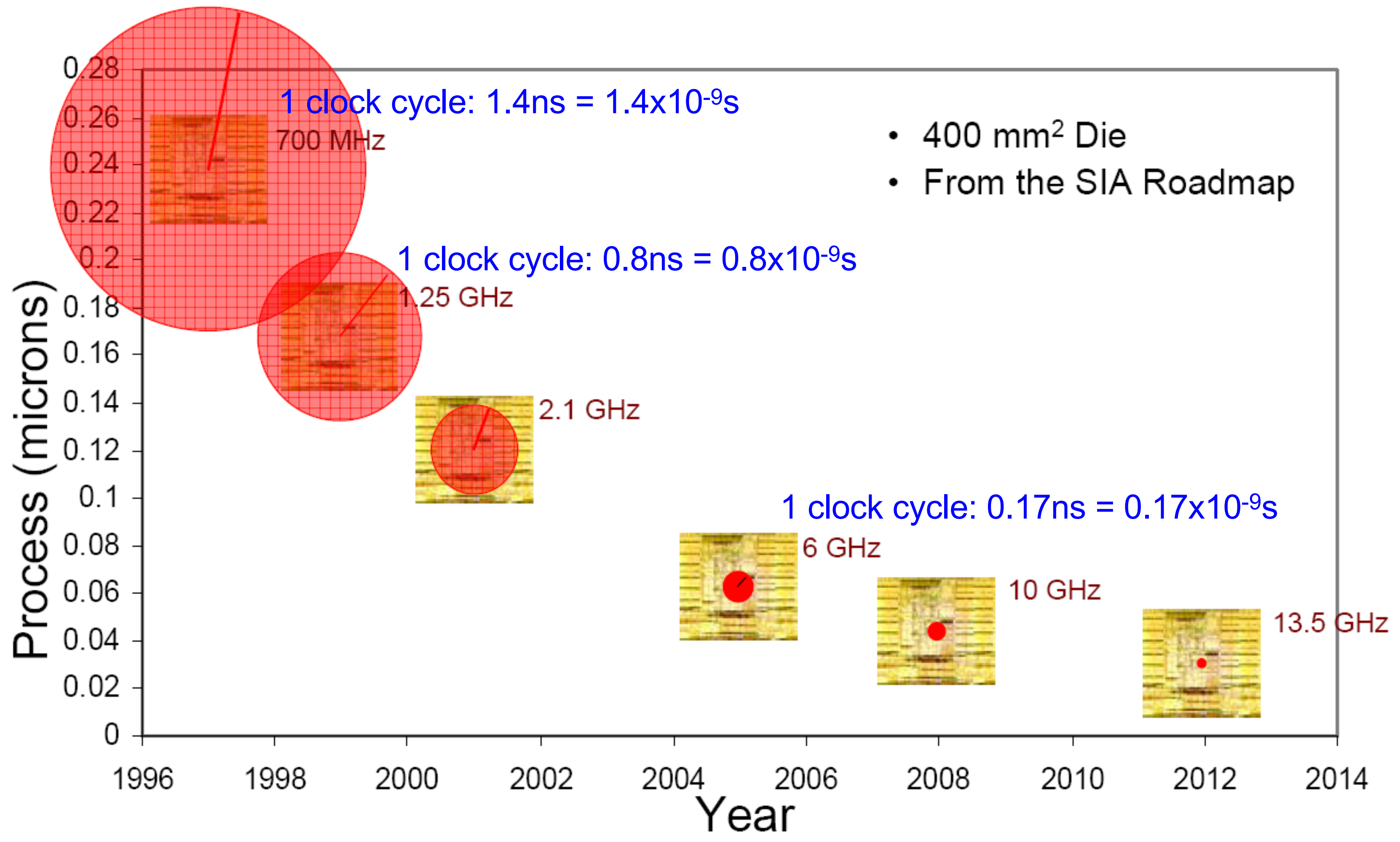


Wirth’s law: software becomes slower faster than hardware becomes faster.

Bottleneck: Power density



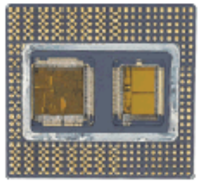
Bottleneck: Wire Delays



- Range ● that electrons can travel in one clock cycle decreases with higher clock frequencies.

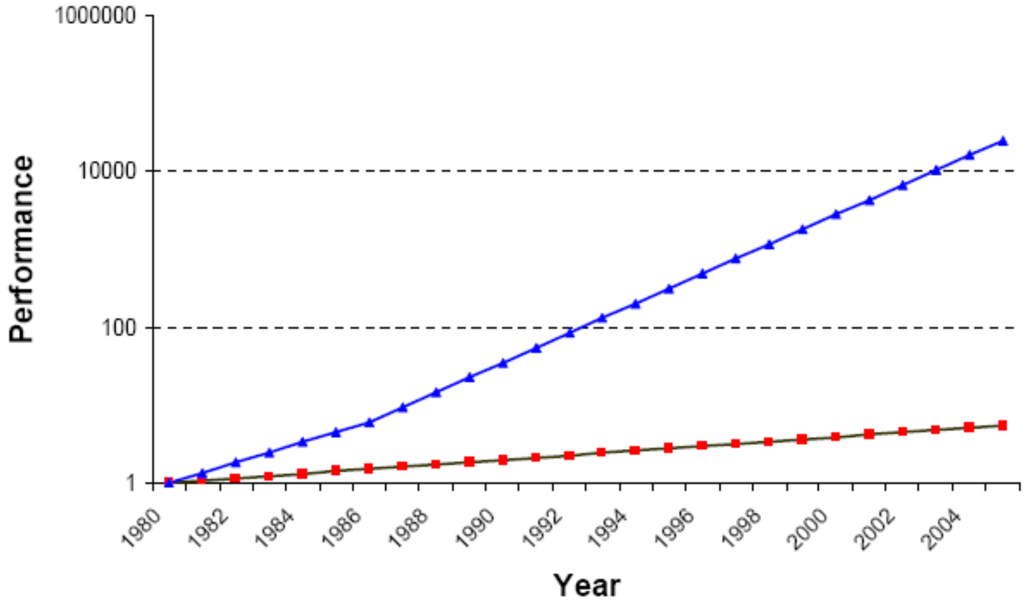
Bottleneck: DRAM Access Latency

- CPU's performance increases faster than DRAM.
- Memory hierarchies increasingly complex
  - L1,L2, L3 caches
- Power efficiency also becomes an issue.

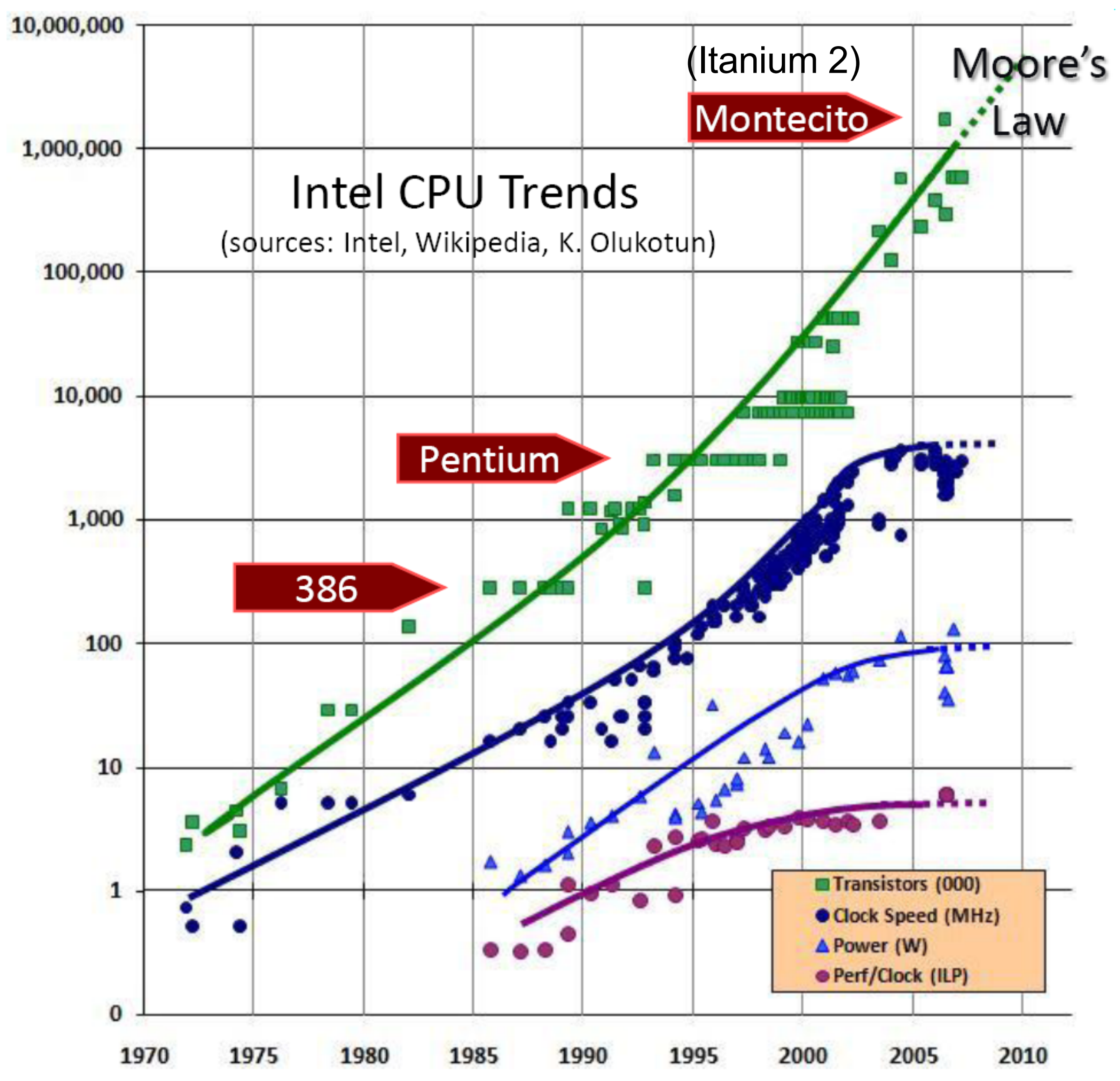


µProc
60%/yr.
(2X/1.5yr)

DRAM
9%/yr.
(2X/10 yrs)



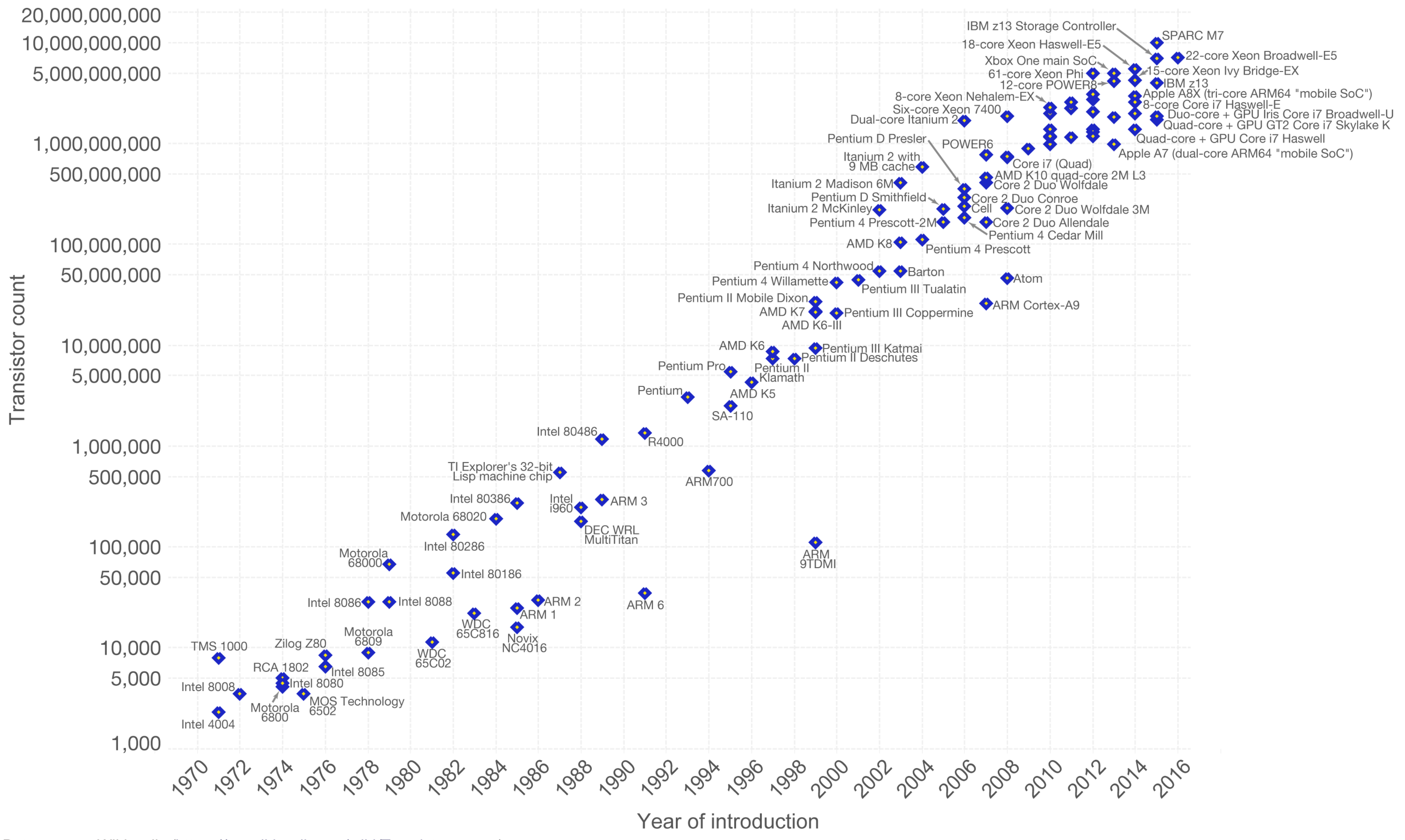
The Future



- Historically: use transistors to boost performance of single instruction streams (faster CPUs, ILP, caches).
- Now: deliver more cores per chip (multicores, GPUs).
- "Every year we get ~~faster~~ **more** processors."

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

The Future

- The free performance lunch is over for sequential applications.
- Transistors on a chip double every 18 months (Moore's Law), **however:**
  - Power consumption proportional to clock-frequency^2
  - Wire delays
  - Diminishing returns from instruction-level parallelism (ILP)
  - DRAM access latency

  → No substantial performance improvement of single core CPUs in sight.

  → No more speed-ups for sequential applications (see next slides).
- Hardware solution:
  - increase the number of cores per processor
  - new parallel computer architectures
    - GPGPUs (e.g., NVIDIA CUDA)
    - Cell architecture (heterogeneous multicore)

Outline

- Introduction to parallelism ✓
  - Software development ✓
  - Hardware development ✓
- Forms of parallelism
  - Task-parallelism
  - Data-parallelism
- Examples

Example: preparing a salad

+

+

=

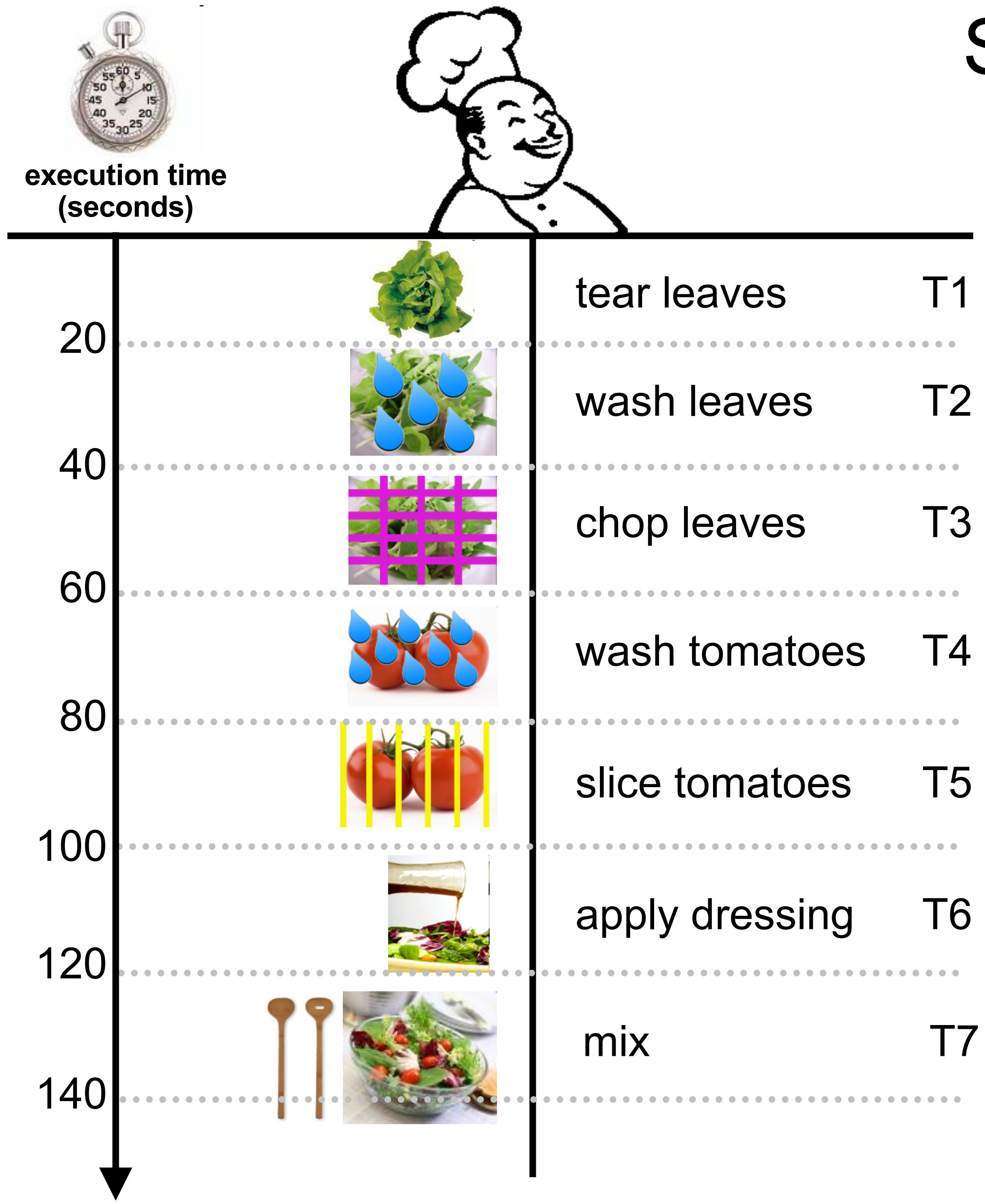
Tasks to do:

- tear leaves
- wash leaves
- chop leaves

- wash
- slice

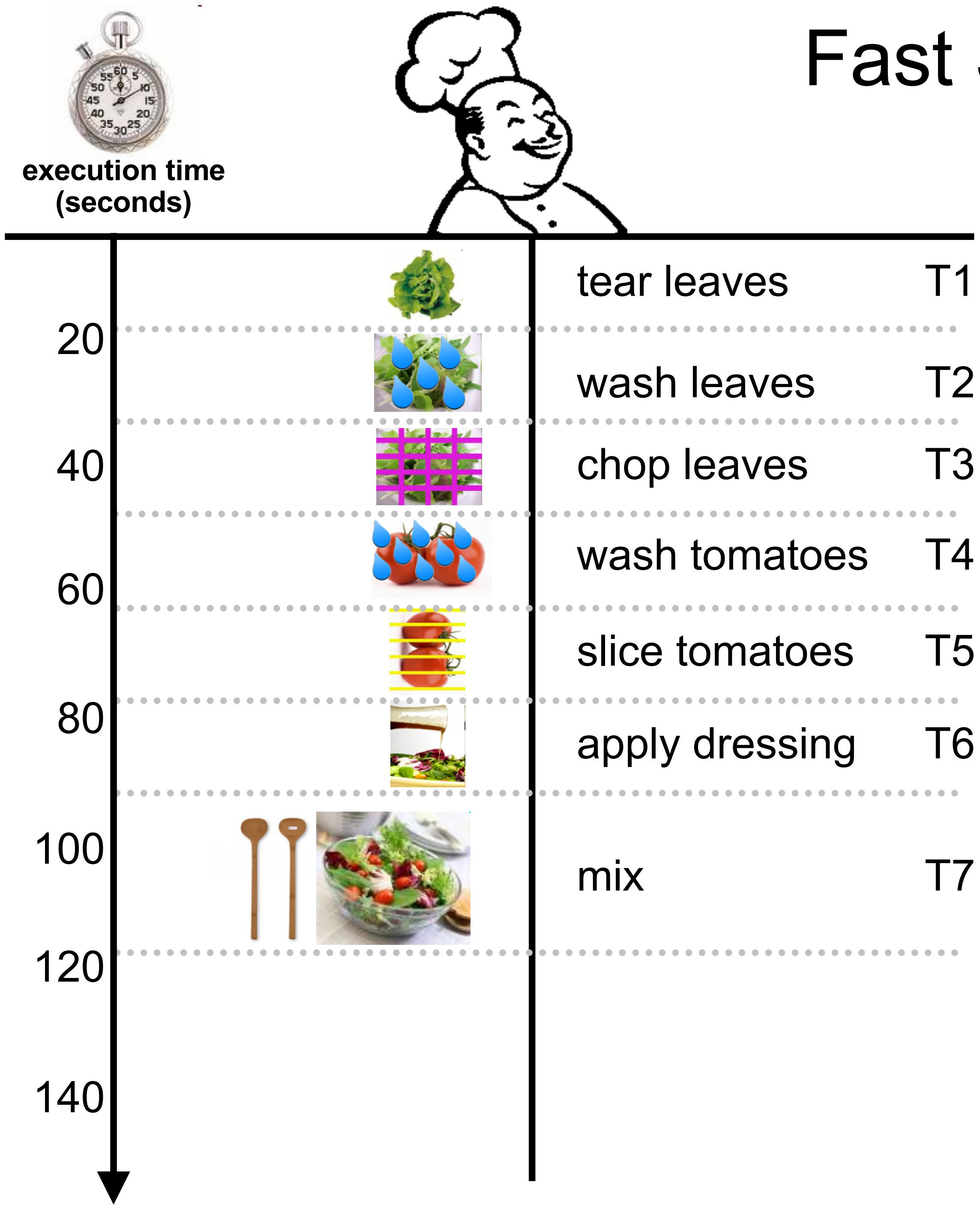
- apply dressing
- mix

Single Chef Salad (sequential)



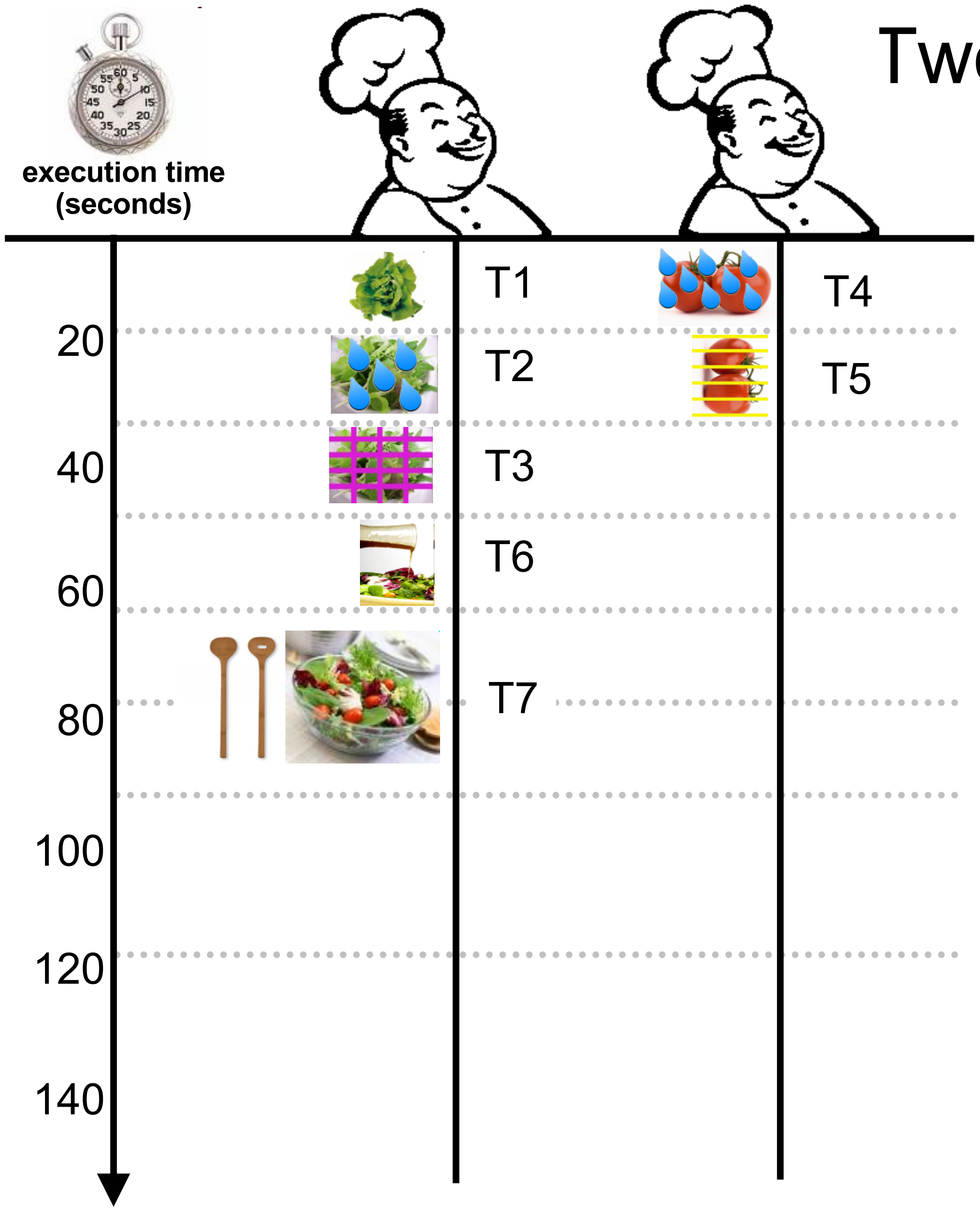
- Preparing a salad consists of 7 tasks (T1-T7).
- A single chef prepares a salad in processing the 7 tasks **sequentially** (one after the other).
- It takes a single chef 142 seconds to prepare a salad.
- We can speed up the process by making the chef work **faster**.

Fast Single Chef Salad (sequential)



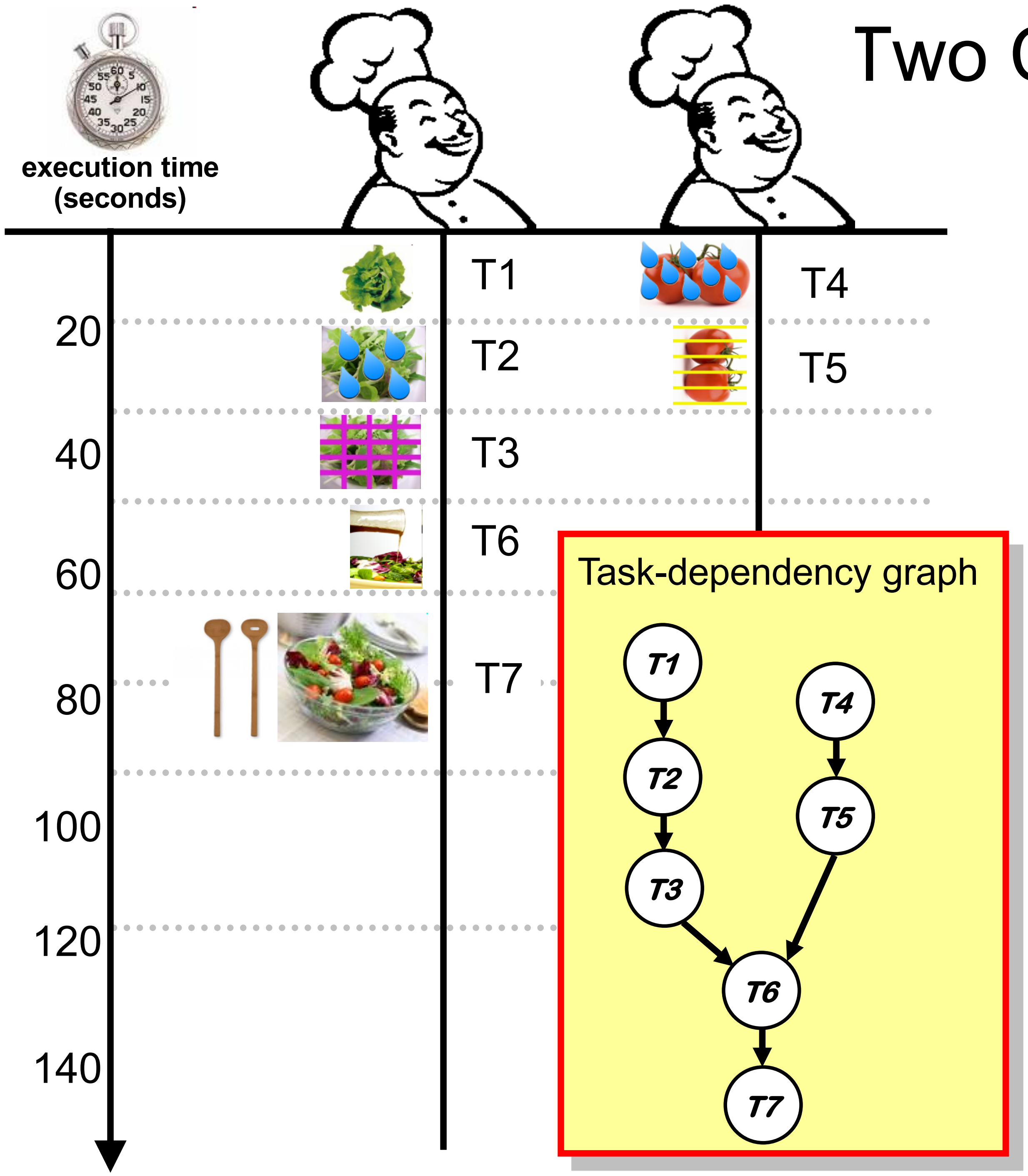
- If we can convince the Chef to finish every task in 18 instead of 20 seconds, we will reduce the process to 7x18=126 seconds.
- The chef agrees for Tasks T1-T6.
- However, he cannot speed up Task T7 without spoiling the entire kitchen.
- He refuses to work any faster than that.
- We can speed up the process by bringing in an additional Chef that works **in parallel** to Chef 1.

Two Chefs in Parallel Salad



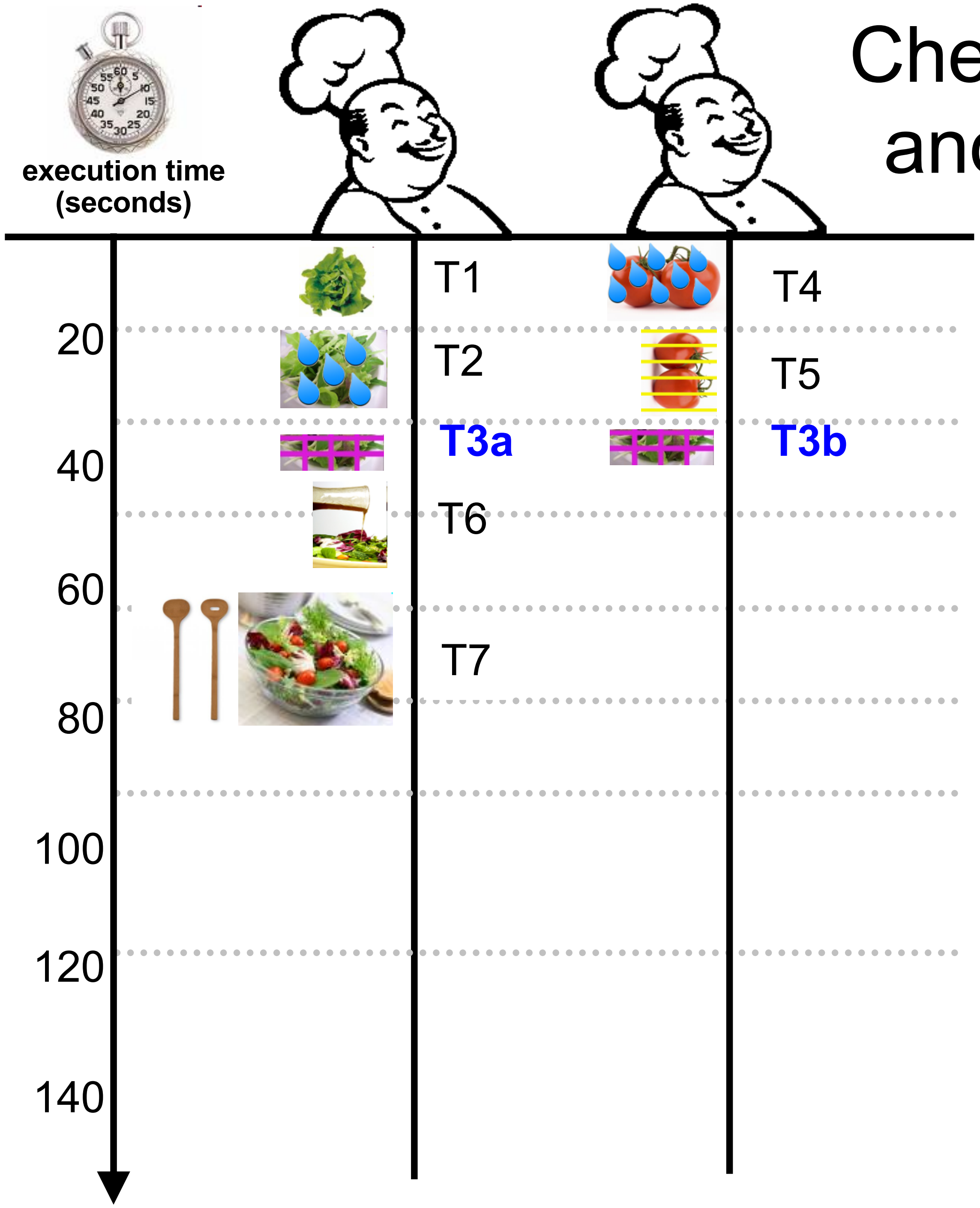
- Chef 2 washes the tomatoes (Task T4) while Chef 1 tears the lettuce (Task T1).
- Task T1 is processed in parallel to Task T4. Processing tasks in parallel is called **task-parallelism**.
- Another form of task-parallelism occurs between tasks T2 and T5.
- There exist **dependencies** between tasks:
  - Vegetables must be washed *before* they are chopped/sliced.
  - Mixing (T7) can only be done *after* the dressing has been applied (T6).

Two Chefs in Parallel Salad

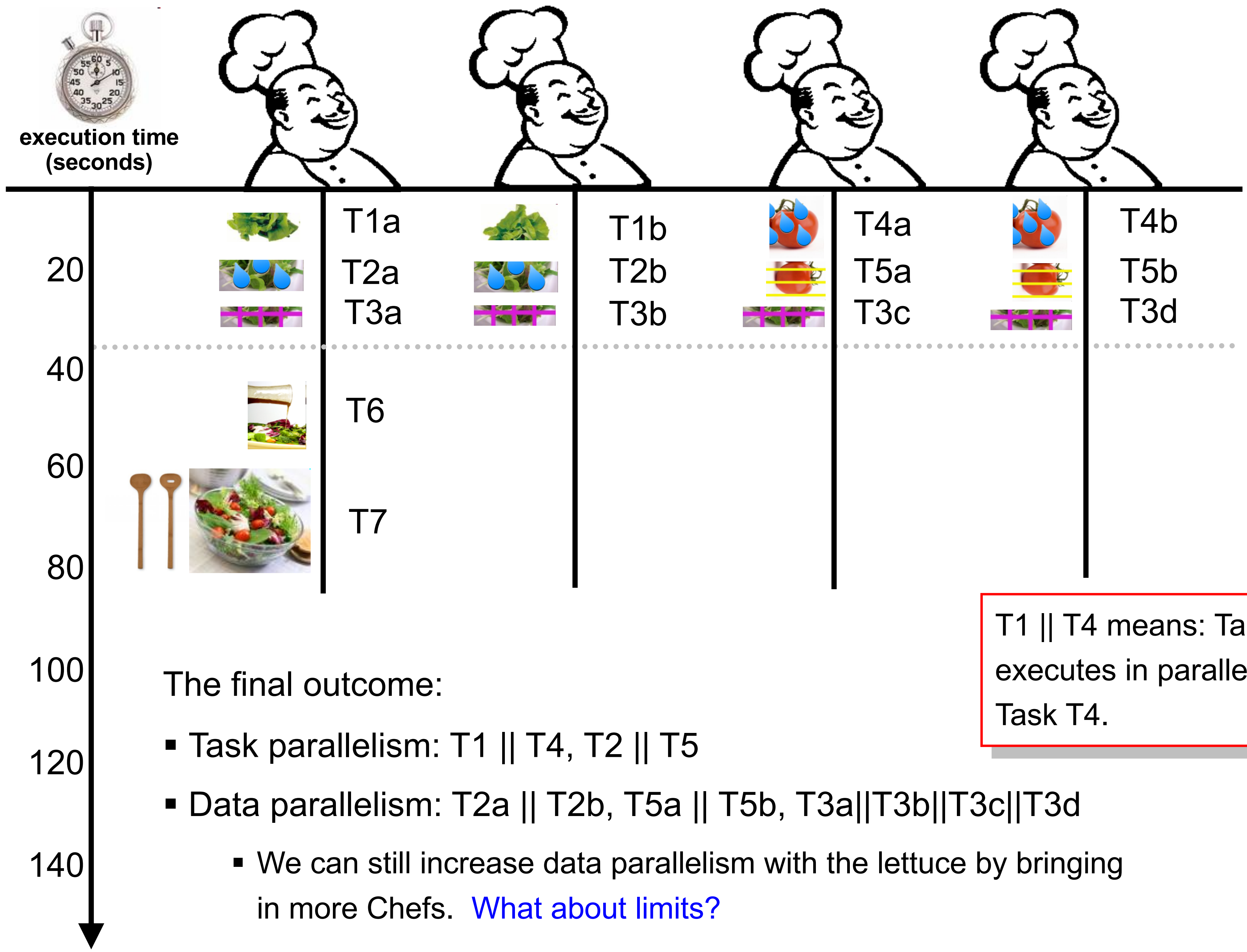


- Dependencies between tasks do not allow further task-parallelism with this example.
- A **task-dependency graph** shows dependencies between tasks.
  - directed acyclic graph (DAG)
  - graph nodes represent tasks
  - directed edge $T_a \rightarrow T_b$ indicates that T_b can only be processed after T_a has completed.
  - See example graph to the left.

Chefs in Parallel (task and data parallelism)



- Dependencies between tasks do not allow further task-parallelism with this example.
- However, we can have several Chefs work in parallel on the “data” of a task.
- This form of parallelism is called **data-parallelism**.
  - Example1: Chef 1 and Chef 2 chop one half of the lettuce each (Tasks **T3a** and **T3b**).
  - Example2: On the next slide more data-parallelism is introduced.



The final outcome:

- Task parallelism: T1 || T4, T2 || T5
- Data parallelism: T2a || T2b, T5a || T5b, T3a||T3b||T3c||T3d
  - We can still increase data parallelism with the lettuce by bringing in more Chefs. What about limits?

T1 || T4 means: Task T1 executes in parallel to Task T4.

Definitions

Task: a computation that consists of a sequence of instructions. The computation is a *distinct* part of a program or algorithm. (That is, programs and algorithms can be de-composed into tasks).

Examples: “washing lettuce”, “initialize data-structures”, “sort array”, ...

Task-parallelism: parallelism achieved from executing different tasks at the same time (i.e., in parallel).

Data-parallelism: performing the same task to different data-items at the same time.

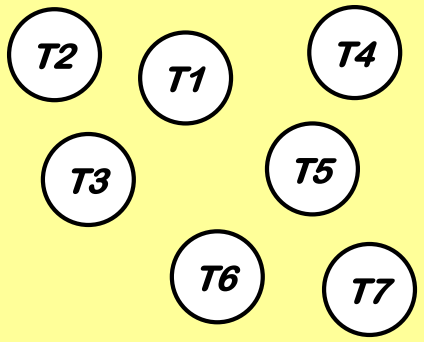
Example: 2 Chefs slicing 1 tomato each. (tomato = data, slicing ~ task).

Definitions (cont.)

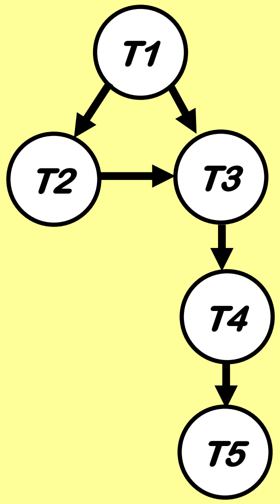
Dependencies: an execution order between two tasks Ta and Tb.
Ta must complete before Tb can execute. Notation: Ta → Tb.
Dependencies limit the amount of parallelism in an application.

Example task dependency graphs:

no dependencies,
maximum parallelism:



dependencies, no
parallelism possible:

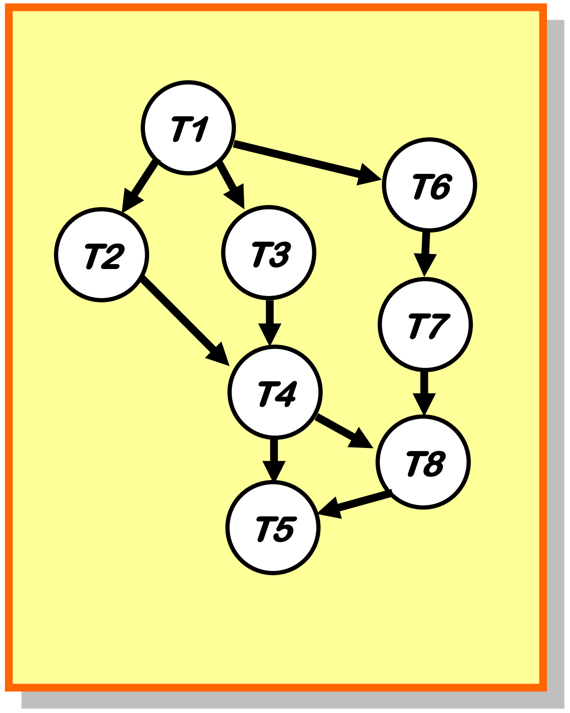


Definitions (cont. cont.)

Dependencies impose a **partial ordering** on the tasks:

Two tasks Ta and Tb can execute in parallel iff

1) there is no path in the dependence graph from Ta to Tb
2) there is no path in the dependence graph from Tb to Ta

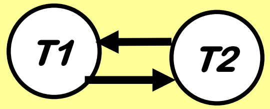


Dependency is transitive:

Ta→Tb and Tb→Tc implies Ta→Tc

Say: Ta has to complete befor Tb, and Tb has to complete before Tc, therefore Ta has to complete before Tc.

What about this?



Who will do the actual parallelization ?

- The compiler?
  - Would be nice. Programmers could continue writing high-level language programs
  - The compiler would find a task-decomposition for a given multicore processor.
  - Unfortunately this approach does not work (yet).
    - Esp. heterogeneous multiprocessors are difficult to program
    - The speed-up gained from automatic parallelization is limited.
  - Parallelism from automatic parallelization is called **implicit parallelism**.
- The programmer?
  - Yes! (contents of this course)
    - Needs to understand the program to find a task-decomposition.
    - Needs to understand the hardware to achieve a task-decomposition that fits the underlying hardware.
    - Needs to take care of communication & coordination among tasks.
  - Parallelism done by the programmer (her/him)self is called **explicit parallelism**.
  - The research community is working on programming languages and tools that ease this task.

Outline

- Introduction to parallelism ✓
  - Software development ✓
  - Hardware development ✓
- Forms of parallelism ✓
  - Task-parallelism ✓
  - Data-parallelism ✓
- Examples

Task Parallelism Example

Example: Assume we have a large data-set in an array and the task is to compute the minimum, average and maximum value. This task can be decomposed into 3 tasks: computing minimum (T1), average (T2), and maximum value (T3).

```
#define maxN 1000000000

int m[maxN];
int i;
int min = m[0];
int max = m[0];
double avrg = m[0];

for(i=1; i < maxN; i++) {
   if(m[i] < min)
     min = m[i];
   avrg = avrg + m[i];
   if(m[i] > max)
     max = m[i];
}
avrg = avrg / maxN;
```

```
#define maxN 1000000000
int m[maxN];
```

```
int i; int min = m[0];
for(i=1; i < maxN; i++) {
   if(m[i] < min)
     min = m[i];
}
```

T1

```
int j;
double avrg = m[0];
for(j=1; j < maxN; j++) {
   avrg = avrg + m[j];
}
avrg = avrg / maxN;
```

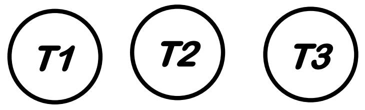
T2

```
int k; int max = m[0];
for(k=1; k < maxN; k++) {
   if(m[k] > max)
     max = m[k];
}
```

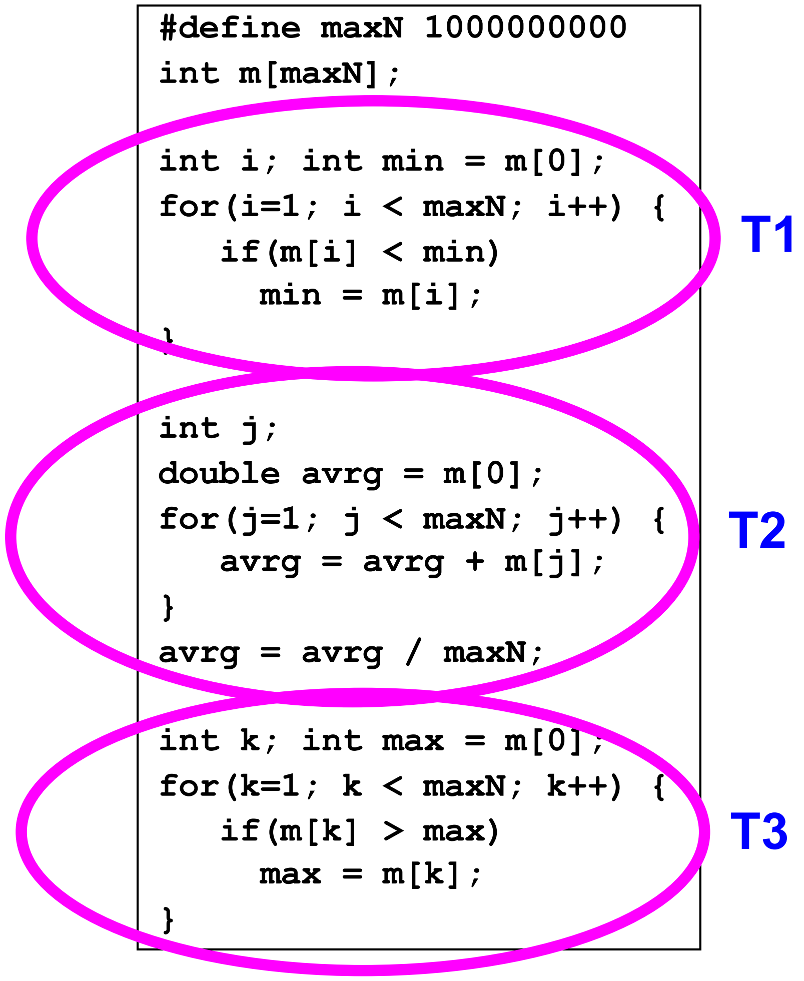
T3

Note: if T1 – T3 should execute in parallel, then each task needs its own loop index variable (i, j, k)!

Dependence graph:



Task Parallelism Example (cont.)



```
#define maxN 1000000000
int m[maxN];

int i; int min = m[0];
for(i=1; i < maxN; i++) {
   if(m[i] < min)
     min = m[i];
}

int j;
double avrg = m[0];
for(j=1; j < maxN; j++) {
   avrg = avrg + m[j];
}
avrg = avrg / maxN;

int k; int max = m[0];
for(k=1; k < maxN; k++) {
   if(m[k] > max)
     max = m[k];
}
```

- The problem is now decomposed into three tasks T1, T2, T3.
- However: still sequential (T1, then T2 then T3).
- Need a way to tell the compiler that tasks T1, T2 and T3 shall be executed in parallel.
- We will use POSIX threads to do that.
- We will discuss other ways to express task parallelism in latter parts of the lecture.

Data Parallelism Example

Example 1: parallel sum computation on array.

Example 2: vector operations:

```
float a[4] = {1,2,3,4};
float b[4] = {1,2,3,4};
float c[4];
int i;

for(i=0; i < 4; i++) {
   c[i] = a[i] + b[i];
}
```

- Assume two arrays of integers, compute the pair-wise sum.
- A sequential version uses a for-loop to compute the sum.
  - Requires one loop iteration per array index.
- A Streaming SIMD Extension (SSE) extension of the Intel processor can do this sum operation in one step.
  - See example on next slides.

SIMD Vectorization

- To sum the values of 2 arrays, a conventional CPU needs one add operation ("+") per array index:

```
float a[4] = {1,2,3,4};
float b[4] = {1,2,3,4};
float c[4];

c[0] = a[0] + b[0];
c[1] = a[1] + b[1];
c[2] = a[2] + b[2];
c[3] = a[3] + b[3];
```

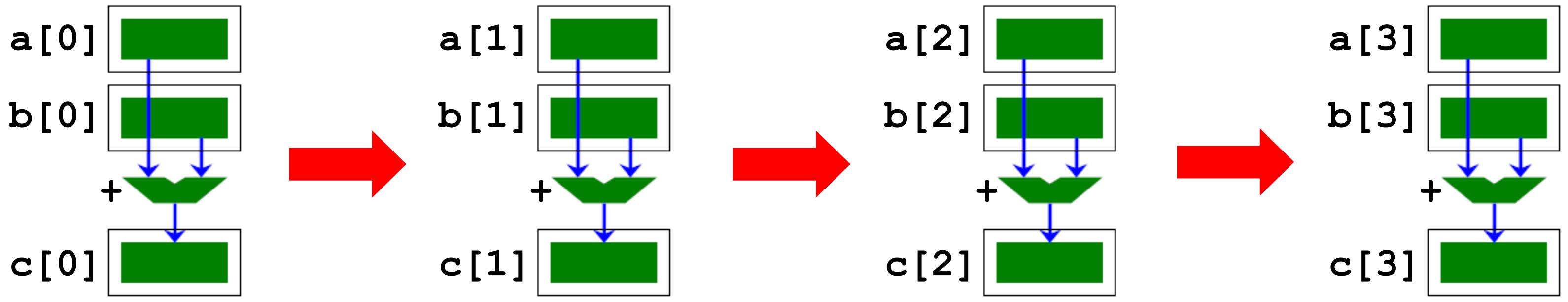
sequential array sum

```
float a[4] = {1,2,3,4};
float b[4] = {1,2,3,4};
float c[4];
int i;

for(i=0; i < 4; i++) {
   c[i] = a[i] + b[i];
}
```

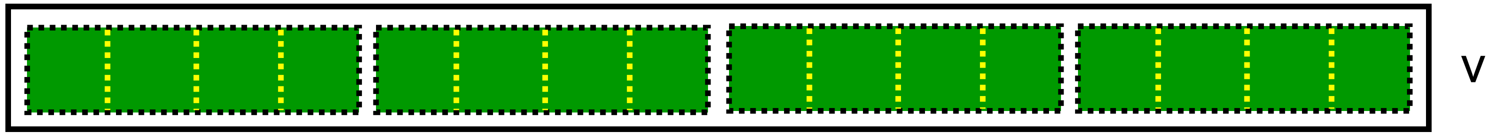
array sum using loop

- Reason: a register of a conventional CPU can only hold only 1 data item at a time (such a register is called a scalar register):



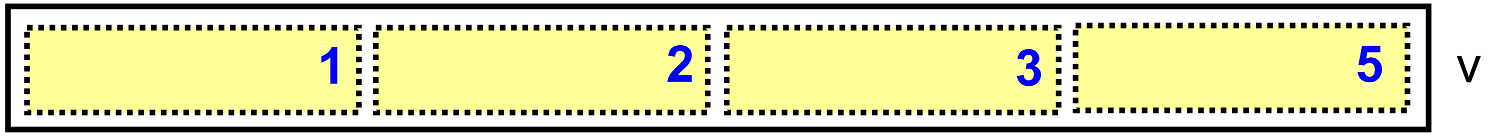
Data Parallelism Example (cont.)

- Vector processors have large registers that can hold multiple values of the *same* data type.
- The registers of the SSE extension of Intel's CPUs are 128bit wide.
  - `v4sf v;` declares vector v, which consists of four fp numbers:



  - The v4sf keyword is an extension to gcc. It takes a primitive data type (char, short, int, ...) and uses it across the whole SSE register.

  - `v = (v4sf) {1.0, 2.0, 3.0, 5.0};` **assigns** values to the **elements** of v.



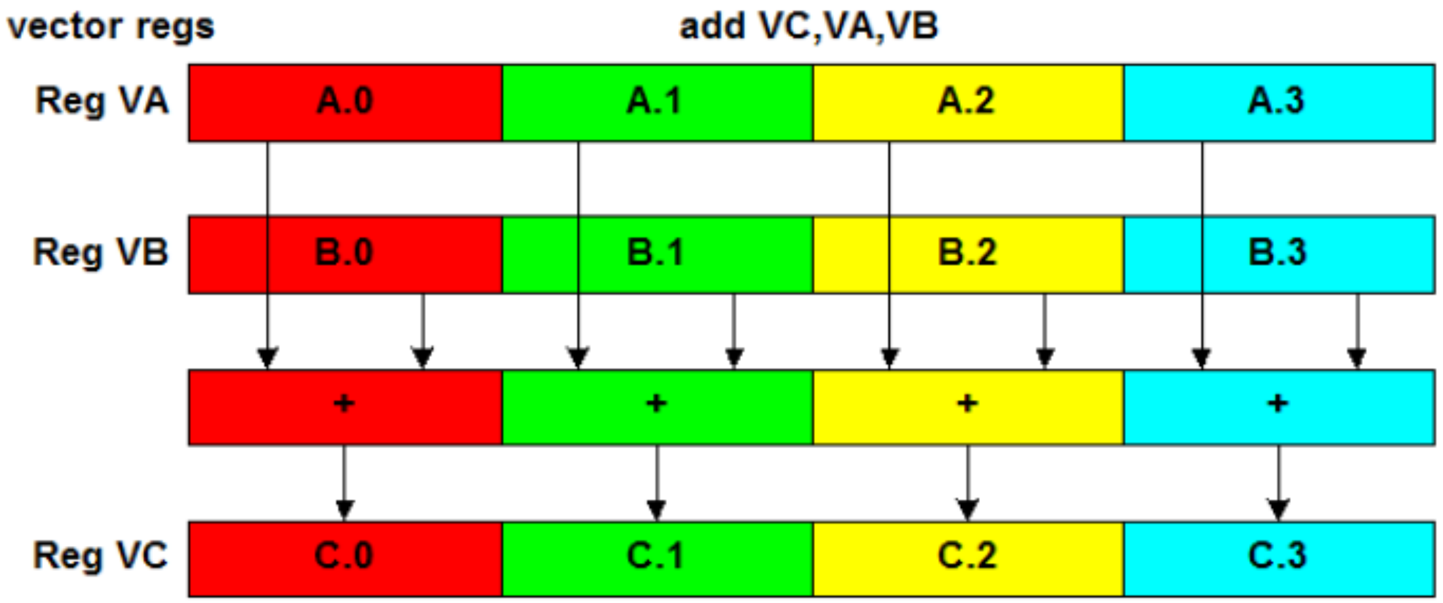
- The SSE can apply an operation to all elements of a vector **at once**.
  - See example on next slide.

Data Parallelism Example (cont.)

- Example: adding two vectors of floats in one step.

```
v4sf VA, VB, VC;

VC =  __builtin_ia32_addps(VA, VB);
```



Data Parallelism Example (cont.)

- Note: the vector instructions on the previous slides are specific to the SSE extensions of Intel processor.

- Many of today’s mainstream CPU architectures support vector instructions.
  - Cell Processor (Playstation 3)
  - Architecture-specific
  - Intel x86: MMX-extensions
  - AMD: 3DNow! for floating-point numbers → Intel SSE
  - PowerPC: AltiVec

Outline

- Introduction to parallelism ✓
  - Software development ✓
  - Hardware development ✓
- Forms of parallelism ✓
  - Task-parallelism ✓
  - Data-parallelism ✓
- Examples ✓