# IF3230
# Sistem Paralel dan Terdistribusi

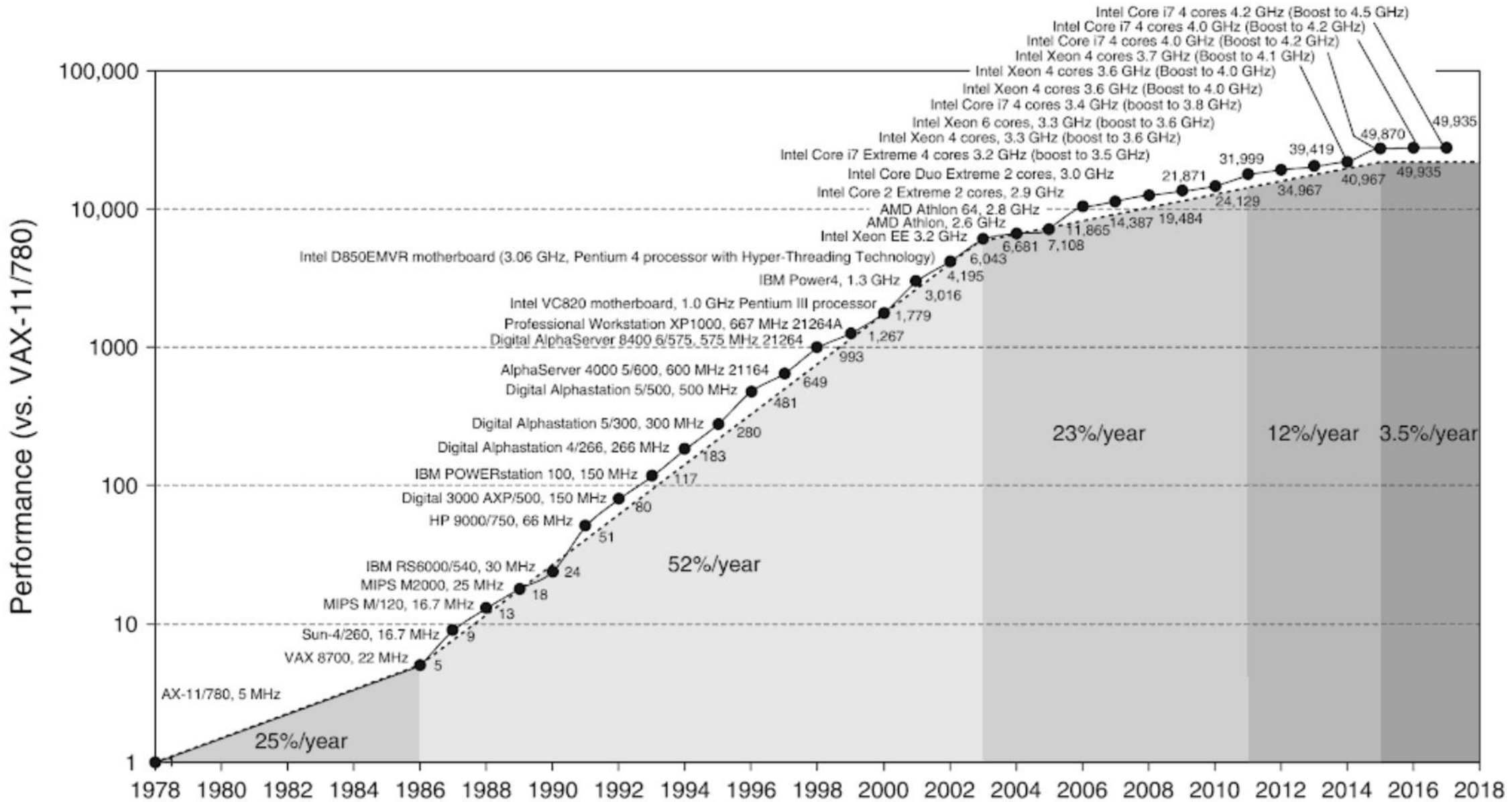
## konsep dasar

Achmad Imam Kistijantoro (imam@staff.stei.itb.ac.id)
Robithoh Annur (robithoh@stafff.stei.itb.ac.id)
Andreas Bara Timur (bara@staff.stei.itb.ac.id)

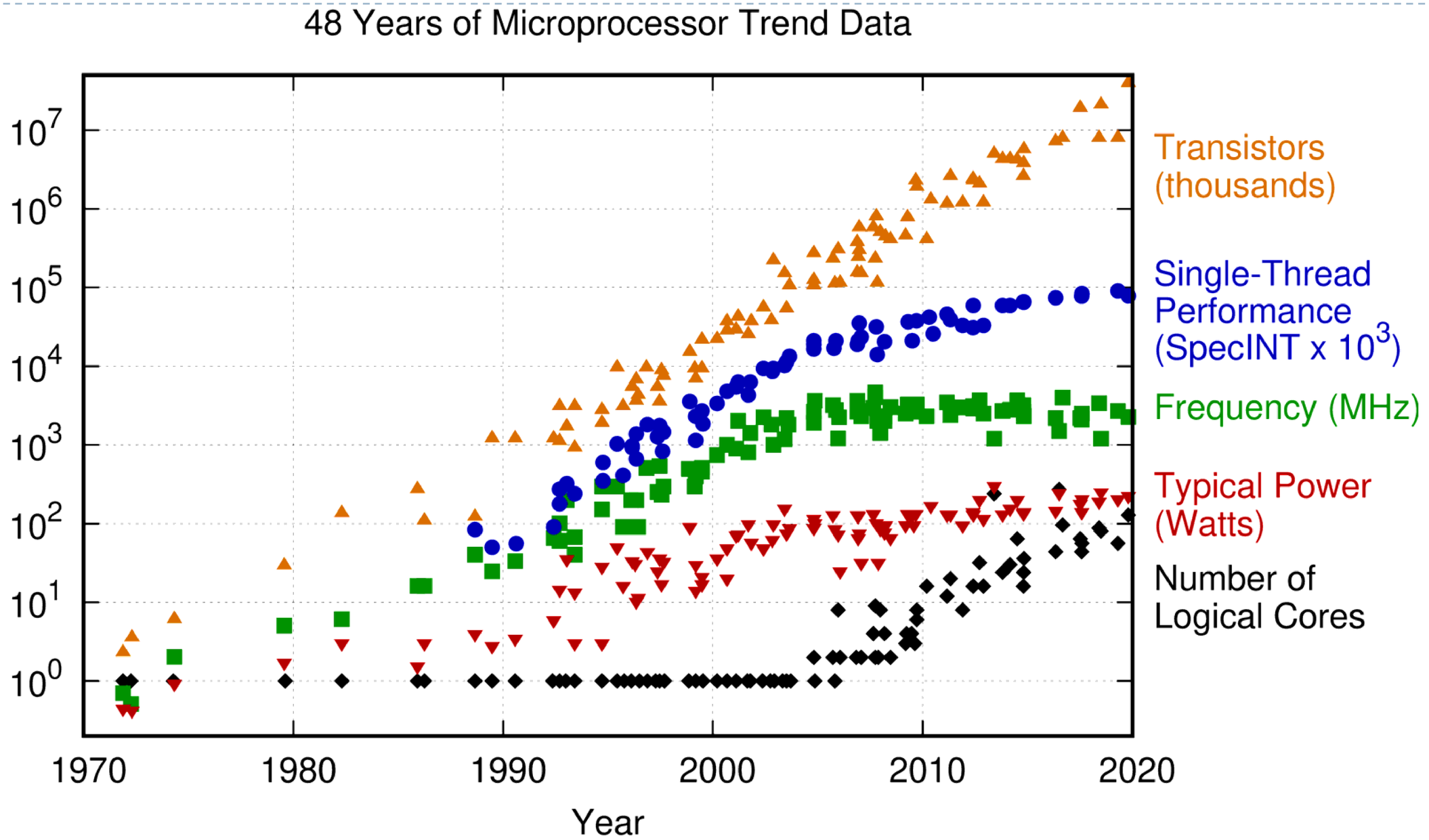
# Motivasi

- Mengapa paralel?
  - Perkembangan hardware
  - Perkembangan kebutuhan kinerja
- Mengapa terdistribusi?

# Tren perkembangan prosesor




Hennessy, J. L. & Patterson, D. A. Computer Architecture, A Quantitative Approach, 6th Edition *Morgan Kaufmann,* **2017**

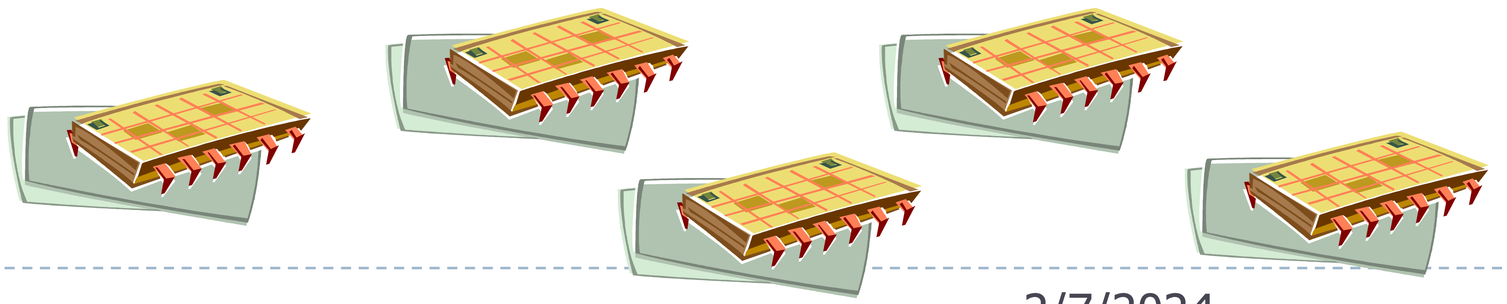





https://raw.githubusercontent.com/karlrupp/microprocessor-trend-data/master/48yrs/48-years-processor-trend.png

# Tren perkembangan prosesor

- Dari 1986 – 2002: peningkatan kinerja prosesor 50% per tahun
- Setelah 2002: turun menjadi 20% per tahun
- Mengapa?
  - Clock speed sudah jenuh (3 GHz)
  - Perkembangan teknologi mempercepat eksekusi instruksi sudah jenuh =>ILP wall teknik²nya udh mentok
- Tren
  - Multicore (Intel, 2005), mengemas multiple prosesor dalam satu chip



# Timeline Advanced Computing/HPC

komputer biasa (HPC → x86)

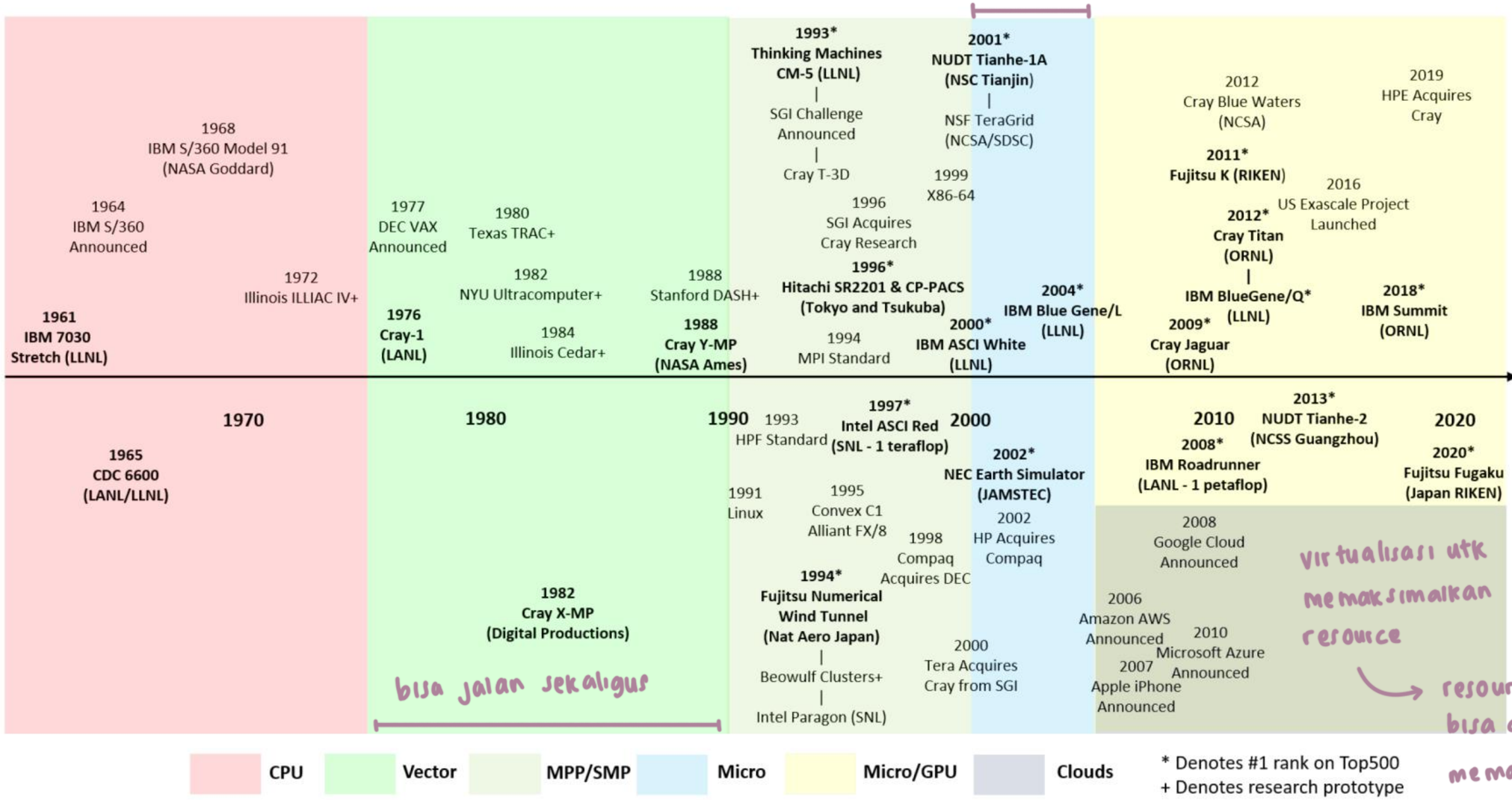



Figure 3: Timeline of Advanced Computing

trik agar software berjalan lbh cepat
↓
membuat proses I/O & komputasi

berjalan bersama

# Timeline Advanced Computing/HPC

- Mainframe computers (1965-1977)
- Vector computers (1977 – 1990)
- Massively Parallel computers, Shared Memory Multiprocessor (1990-2000)
- Microprocessor based (2000-2005)
- Microprocessor + Accelerator (2005-now)
- Cloud based (2005-now)

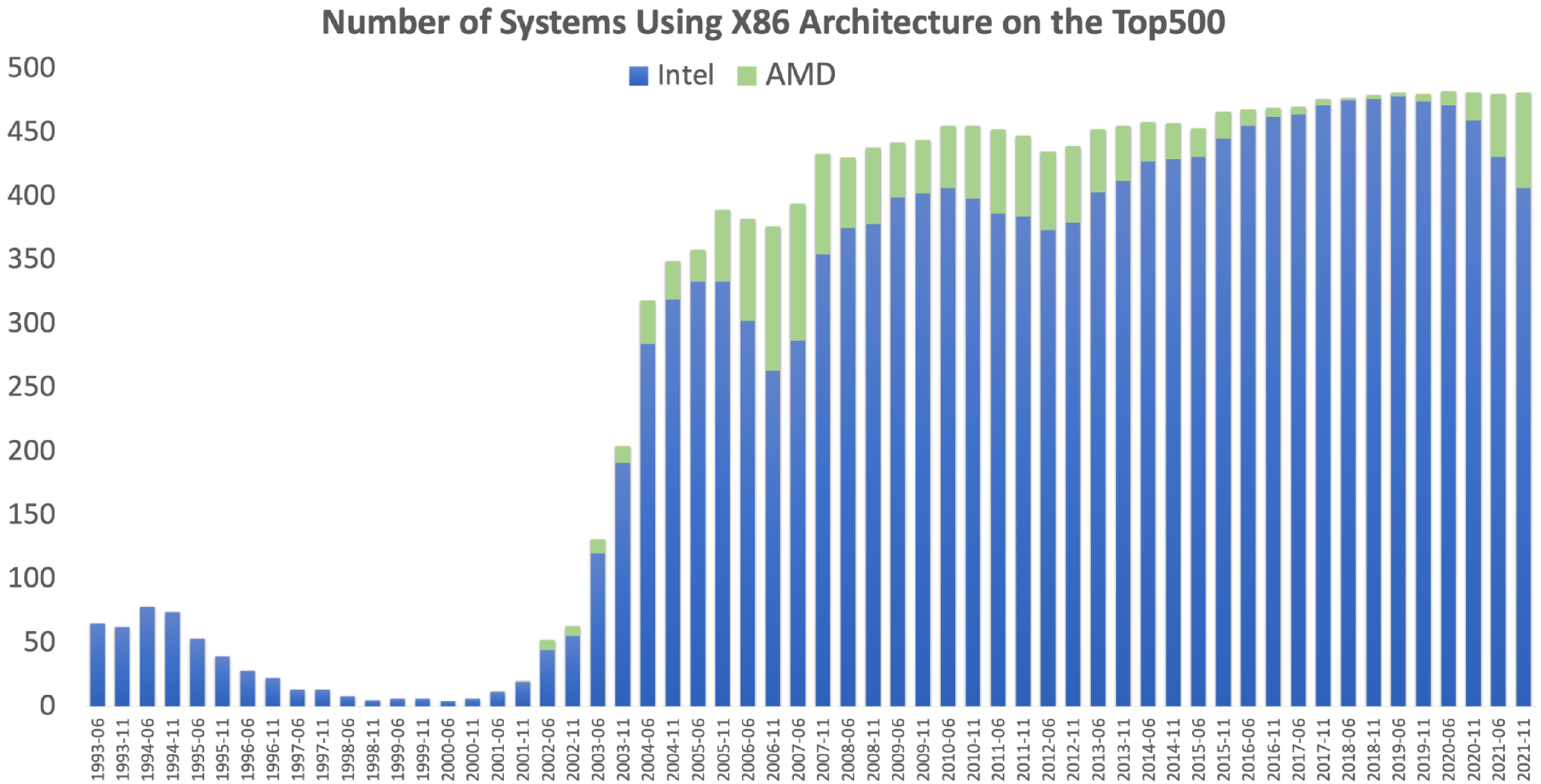



Figure 4: Systems Using the x86-64 Architecture on the TOP500

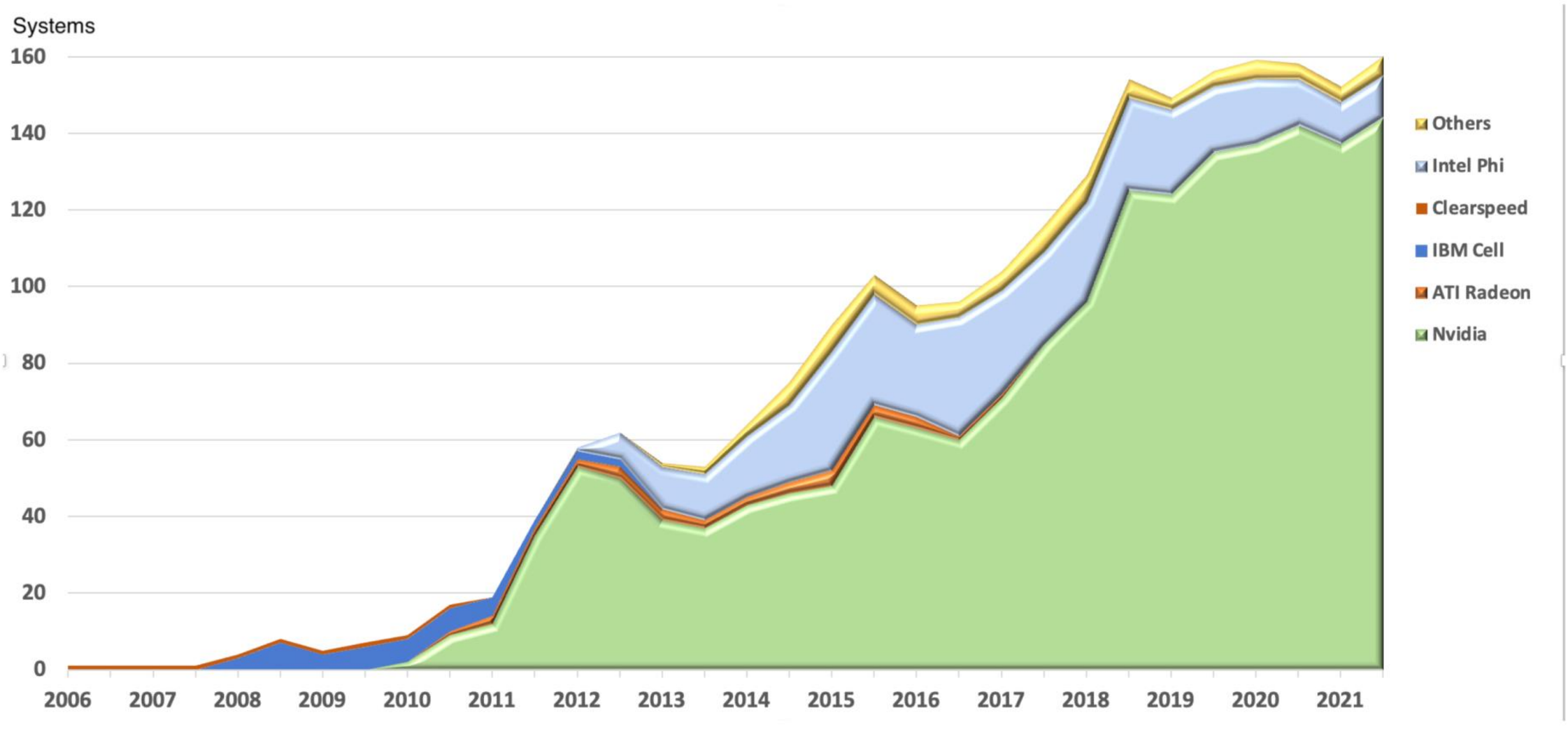
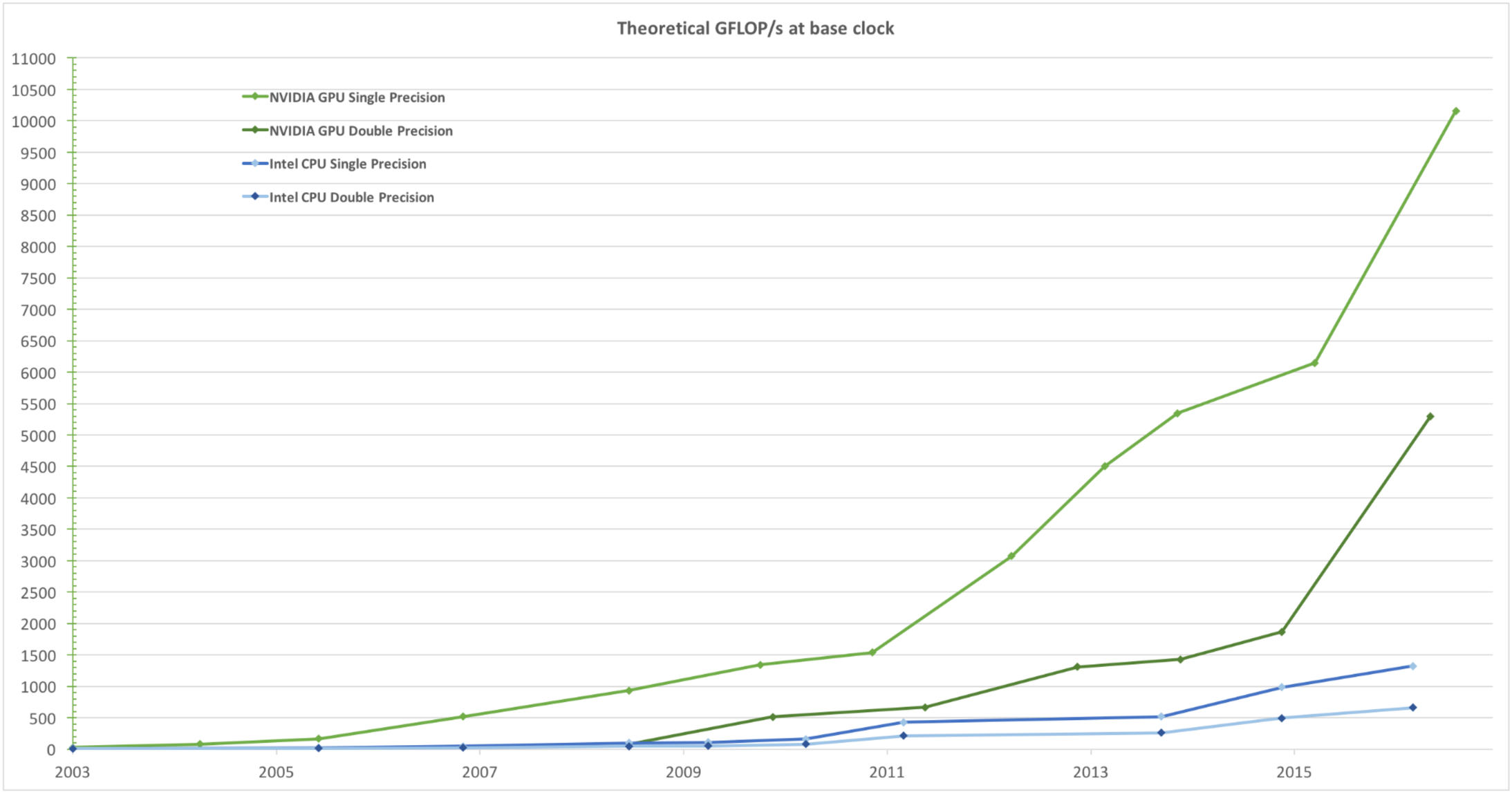


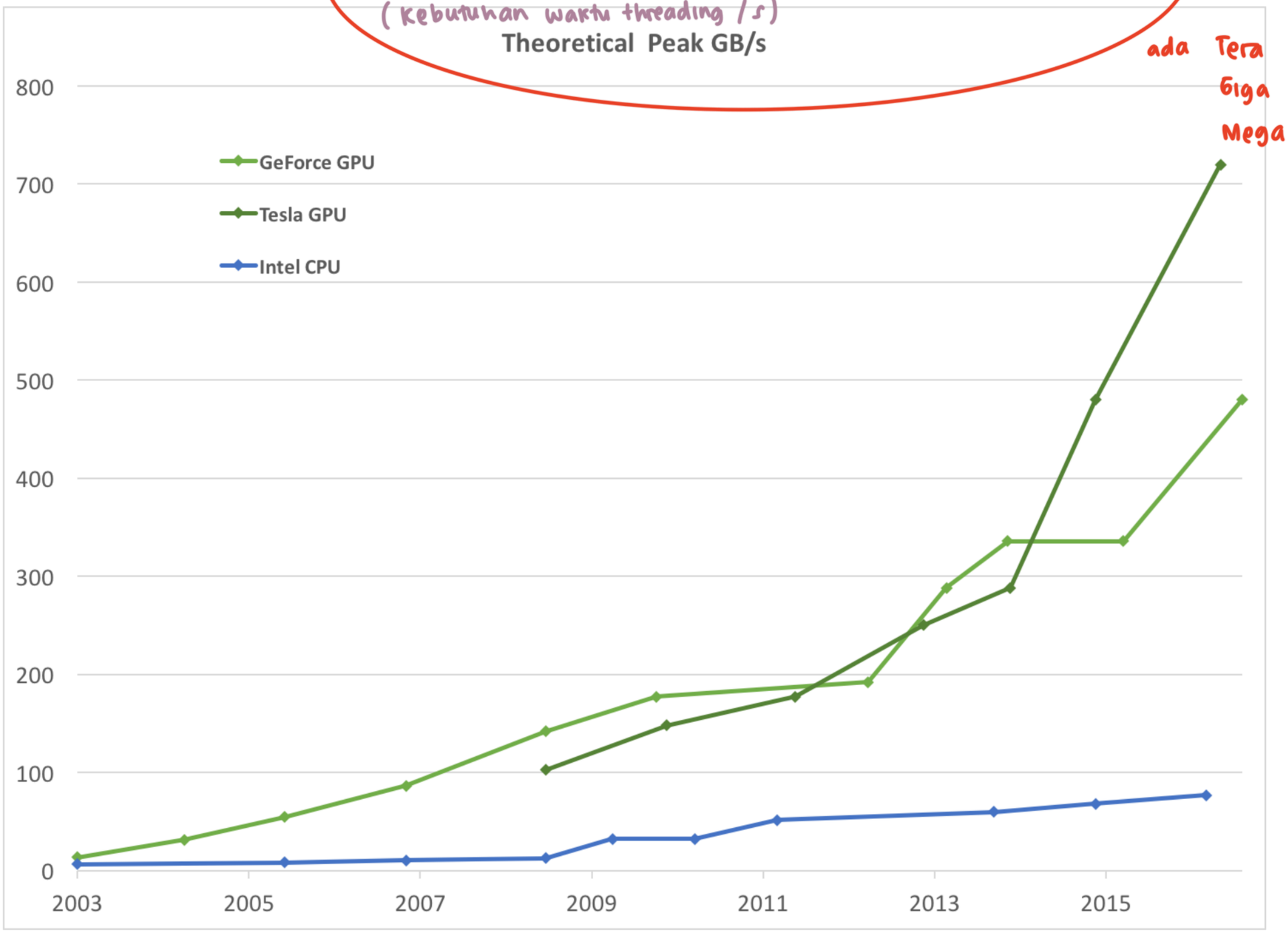

Figure 5: Systems Using GPU Accelerators on the TOP500


Theoretical GFLOP/s at base clock
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Single Precision
Intel CPU Double Precision
11000
10500
10000
9500
9000
8500
8000
7500
7000
6500
6000
5500
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
2003
2005
2007
2009
2011
2013
2015

FLOP → Floating point

FLOP /s
↓
Floating point per sec
(kebutuhan waktu threading /s)

FLOPs
↓
Floating point operations




ada Tera
Giga
Mega

# Teknologi Prosesor

- Berkembang pesat:
  - Handphone dengan harga 8 juta saat ini memiliki kinerja setara dengan Komputer tercepat yang dibeli pada tahun 1993 80 milyar.
- Peningkatan kinerja 1978 – 2018: 50.000x, memungkinkan developer melakukan tradeoff antara kinerja dan produktivitas
  - Runtime interpreter: Java, Scala
  - Scripting language: python, JavaScript
  - Software deployment: Software as a Service
- Data Center as Computer (Warehouse Scale Computer)
- GPU-based computing

# Tren Paralelisasi

Arsitektur Komputer up to 2004: peningkatan kinerja melalui Instruction Level Parallelism (ILP)

Model paralel:

- Data Level Parallelism: banyak data dapat diproses bersamaan
- Task Level Parallelism: banyak task yang dapat dikerjakan bersamaan secara independen

# Arsitektur Komputer Paralel

- **Instruction Level Parallelism**: data-level parallelism dengan kompilator, memanfaatkan pipelining dan speculative execution
- **Vector Architecture, GPU dan multimedia instruction sets**: data-level parallelism dengan menggunakan 1 instruksi yang beroperasi pada banyak data
- **Thread-level parallelism**: data-level parallelism dan task-level parallelism pada tightly-coupled hardware model sebagai parallel thread
- **Request-level parallelism**: parallelism dengan pemisahan task yang dispesifikasikan oleh programmer atau OS

# Komputer Parallel

eax → 32 bit-wide
floating point → 256 bit-wide
double → 128 bit-wide

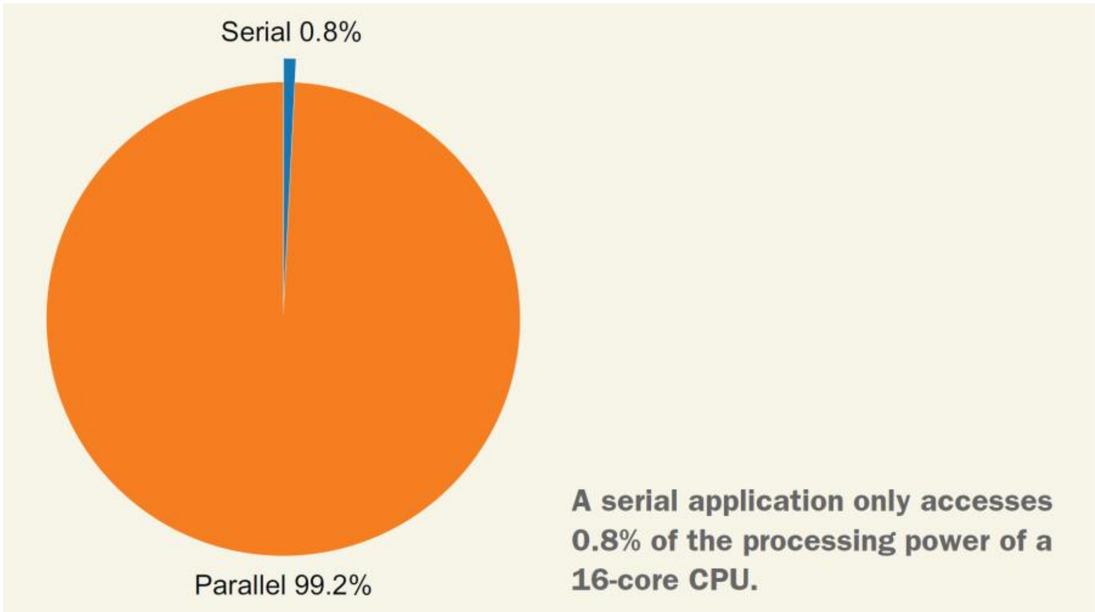
- Komputer parallel tersedia pada mesin yang kita gunakan sehari-hari, e.g. laptop, desktop, server dan bahkan mobile device/handphone
- Misal: Sebuah mesin memiliki 16 core (e.g. Core i9, Xeon E5/Silver), Hyperthreading, 256 bit-wide vector unit (AVX2), maka bisa eksekusi parallel sebesar 16 core x 2 HT x 256 (bit-wide vector unit)/(64 bit double) = 128-way parallelism.
  - 1 serial path hanya menggunakan 1/128 = 0.008 capacity

register khusus

ukuran vektornya 256

tdk semua program bs diparalelkan, ada yg hrs serial

e.g.

# Peran developer

- Multicore chip tidak akan menambah kinerja aplikasi, jika developer tidak memanfaatkannya
  - Serial program hanya berjalan pada salah satu core saja

- **Parallel computing**: the practice of identifying and exposing parallelism in algorithms, expressing this in our software, and understanding the costs, benefits, and limitations of the chosen implementation

- Fokus ke performance: speed, size of the problem & energy efficiency

# Perkembangan kebutuhan kinerja

- Kemampuan hardware meningkat pesat
- Namun kebutuhan komputasi juga meningkat pesat
- Banyak problem yang sebelumnya tidak terbayangkan bisa dilakukan, menjadi mungkin dan menimbulkan kebutuhan baru akan komputasi
  - Decoding human genome
  - Eksperimentasi (kimia, mekanik, molekul, cuaca) berbasis komputasi
  - Pengolahan data masif

# Contoh

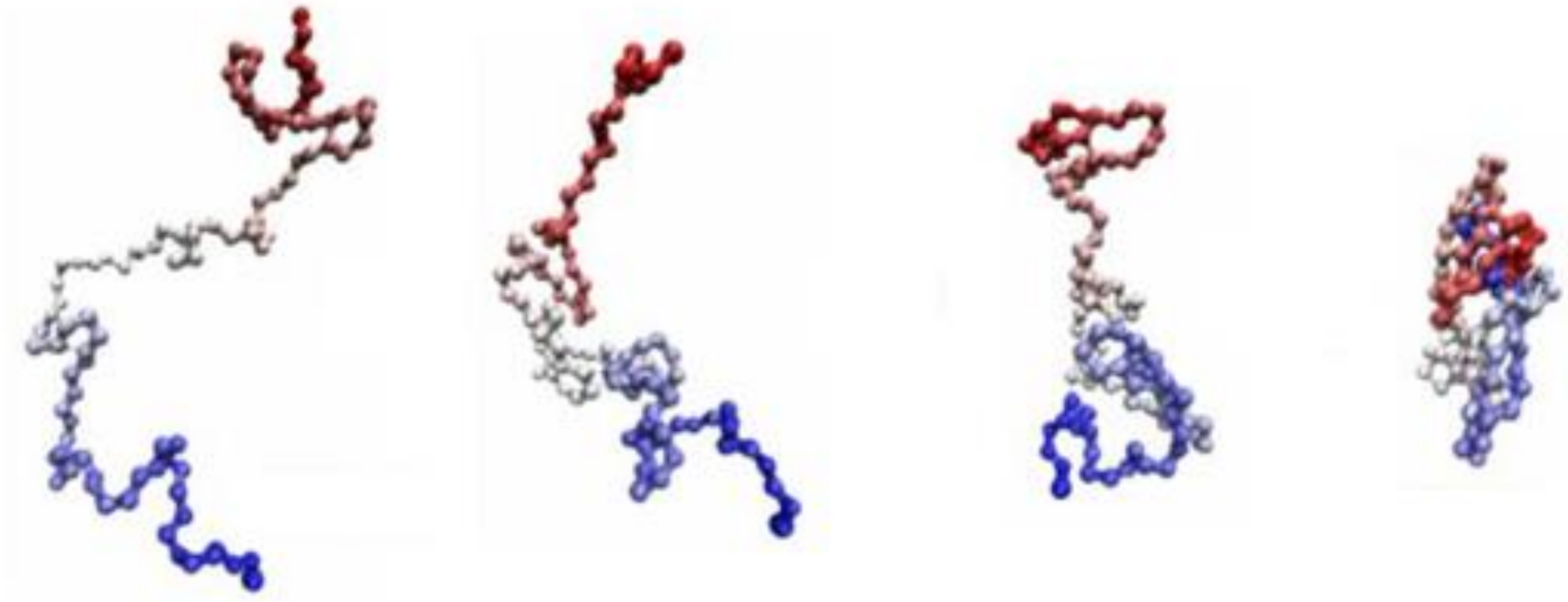
- Pemodelan cuaca





Pacheco, P. “An Introduction to Parallel Programming”. Morgan Kaufmann, 2011

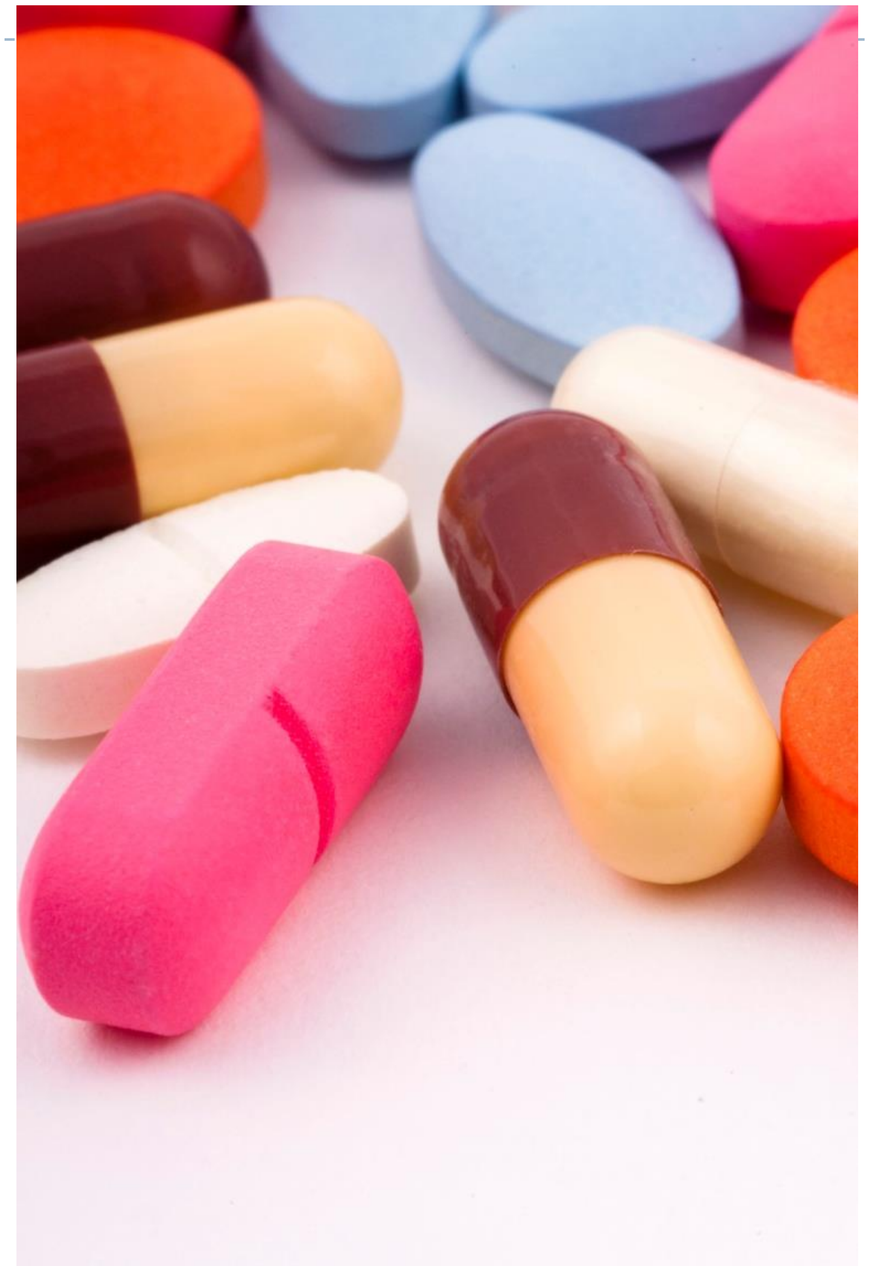
# Protein folding



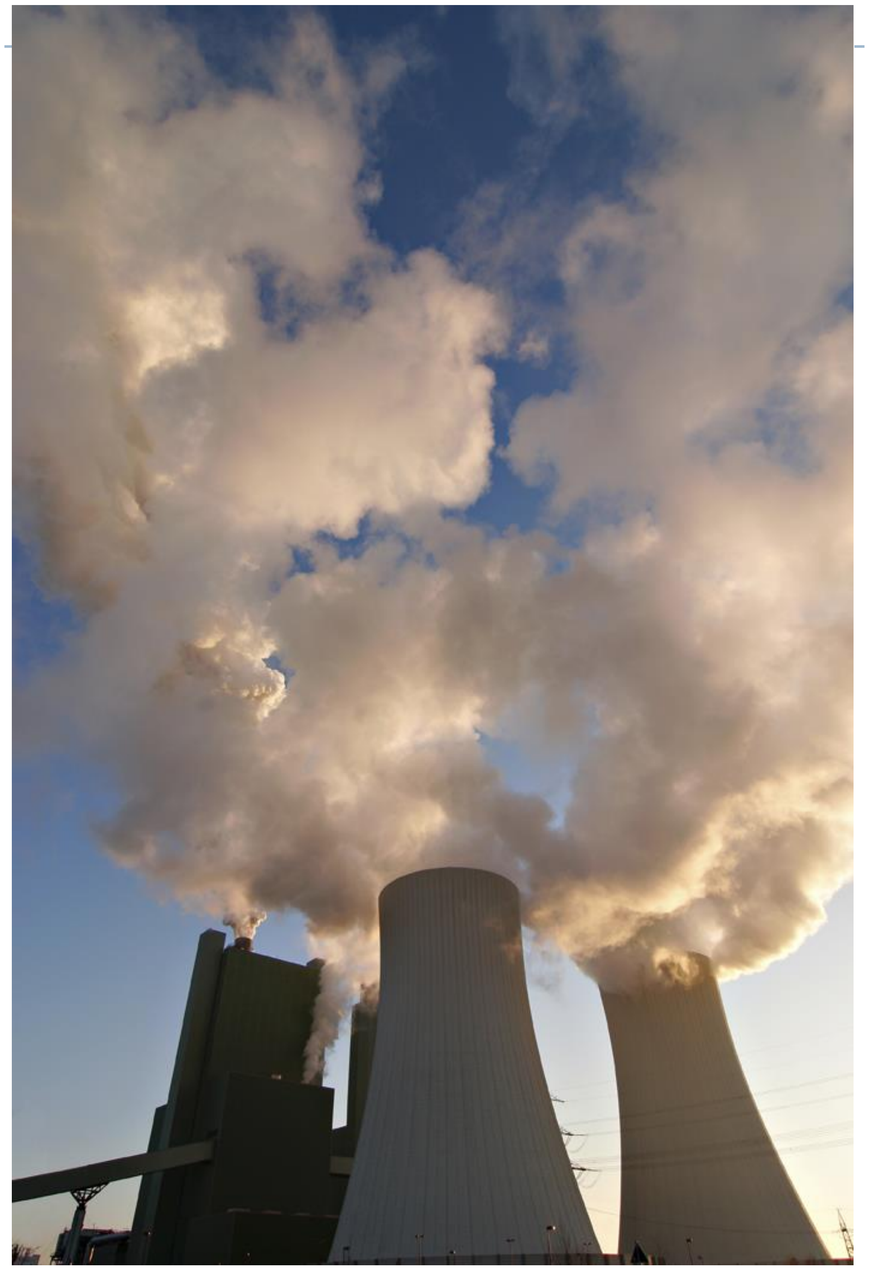


Pacheco, P. "An Introduction to Parallel Programming". Morgan Kaufmann, 2011

# Drug discovery



# Energy research



Pacheco, P. “An Introduction to Parallel Programming”. Morgan Kaufmann, 2011

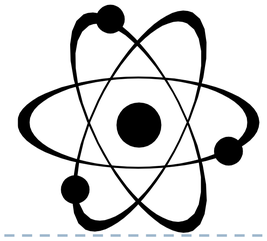
# Data analysis

# Mengapa membangun sistem paralel

- Hingga kini, peningkatan kinerja dilakukan dengan meningkatkan densitas transistor

- Namun ada masalah

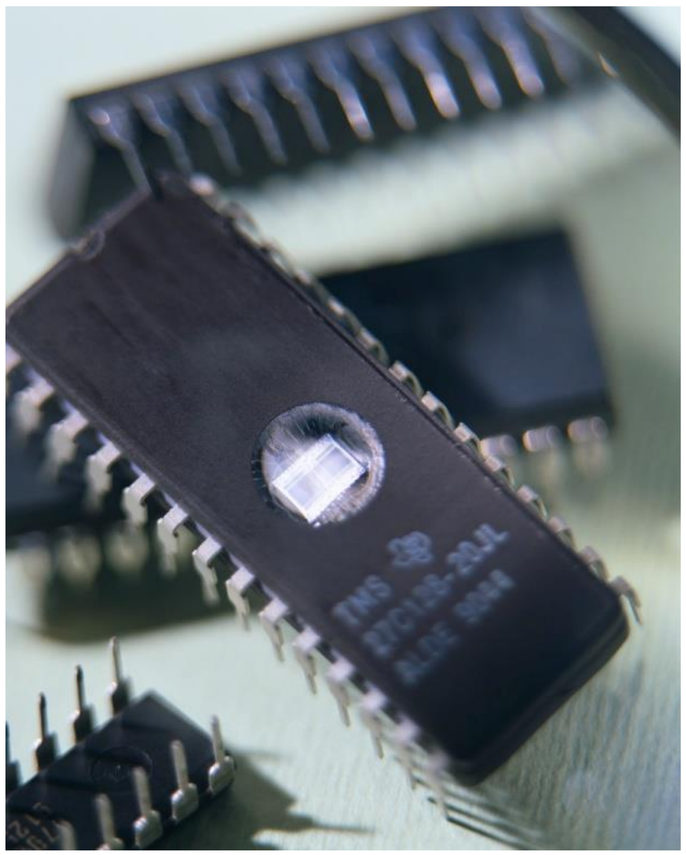
# A little physics lesson



- Smaller 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

- 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.

Kalo jumlah thread > jumlah core
bakal overhead di context switching

```
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 (cont.)

- After each core completes execution of the code, is 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;
}
```

message passing

asumsi tdk ada local variable

multithreading

tiap thread diassign masing2 local var

bikin array isinya kumpulan thread

misal : arr [0] → thread 0

arr [1] → thread 1

bisa ada shared var tp kl konflik jd hrs pasang blocking

# 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 |

But wait!

There's a much better way to compute the global sum.

# Better parallel algorithm

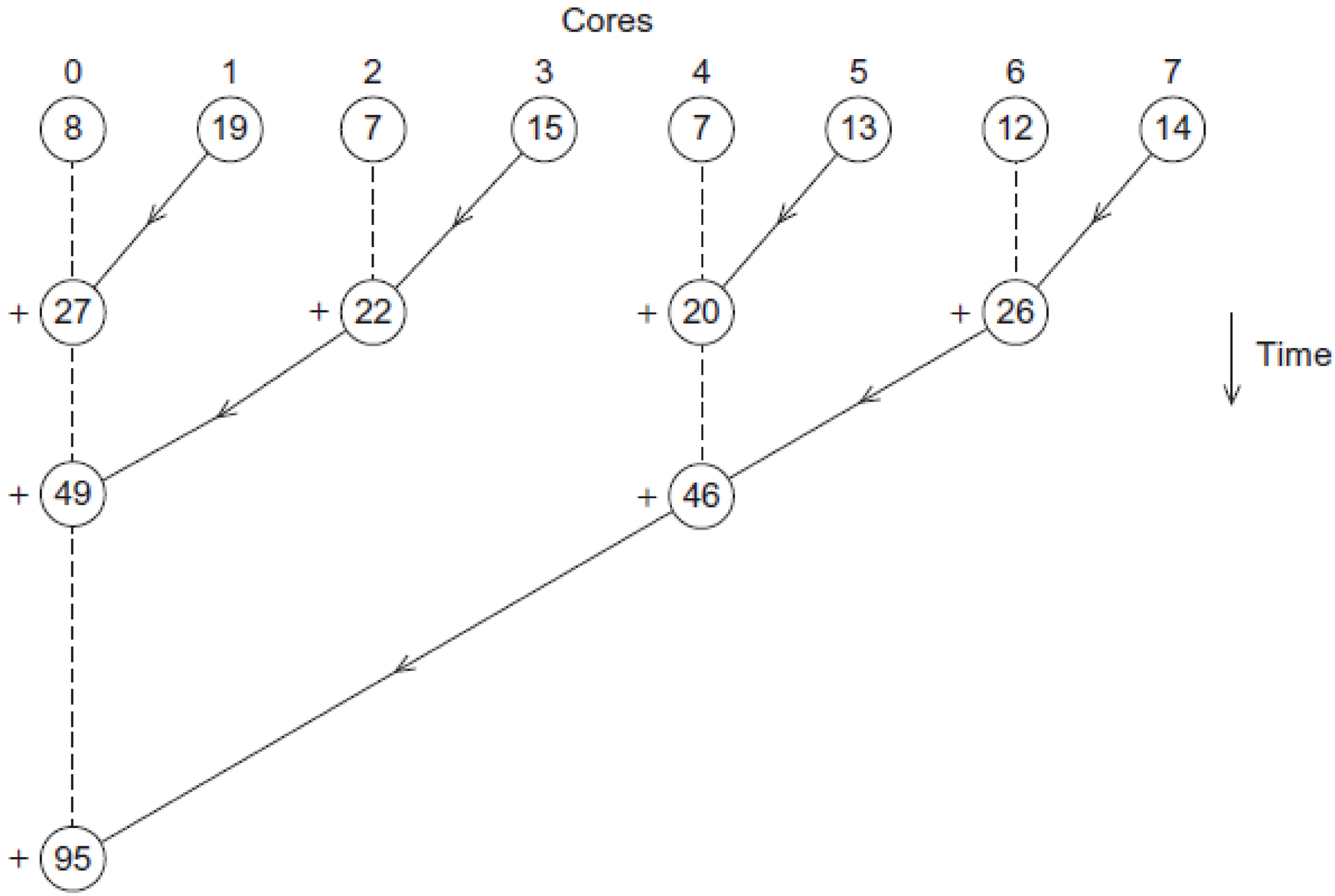
- Don't make the master core do all the work.
- Share it among the other cores.
- Pair the cores so that core 0 adds its result with core 1's result.
- Core 2 adds its result with core 3's result, etc.
- Work with odd and even numbered pairs of cores.

# 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!

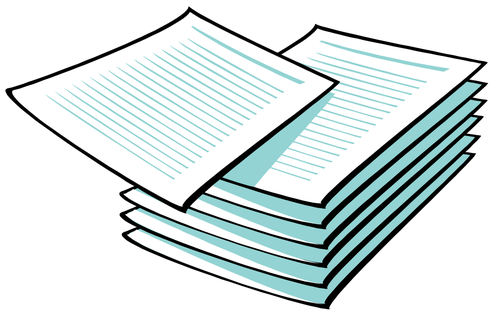
# How do we write parallel programs?

- Task parallelism
  - Partition various tasks carried out 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 it's part of the data.

# Professor P

15 questions

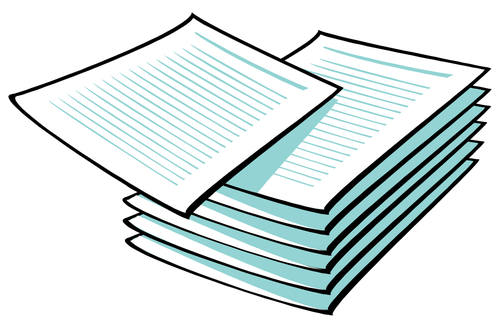
300 exams



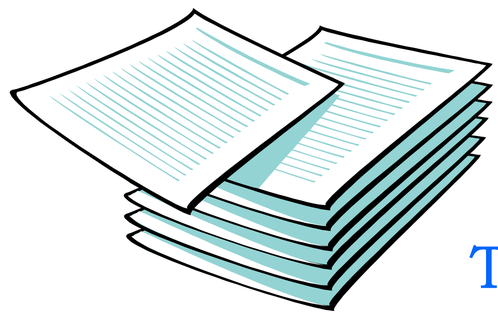
# Professor P's grading assistants

# Division of work – data parallelism

TA#1

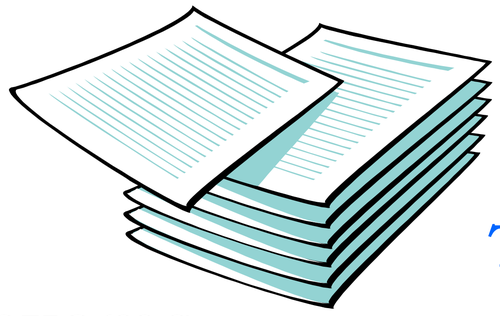


100 exams



TA#3

100 exams



TA#2

100 exams

# Division of work – task parallelism

TA#1

Questions 1 - 5

TA#3

Questions 11 - 15

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.

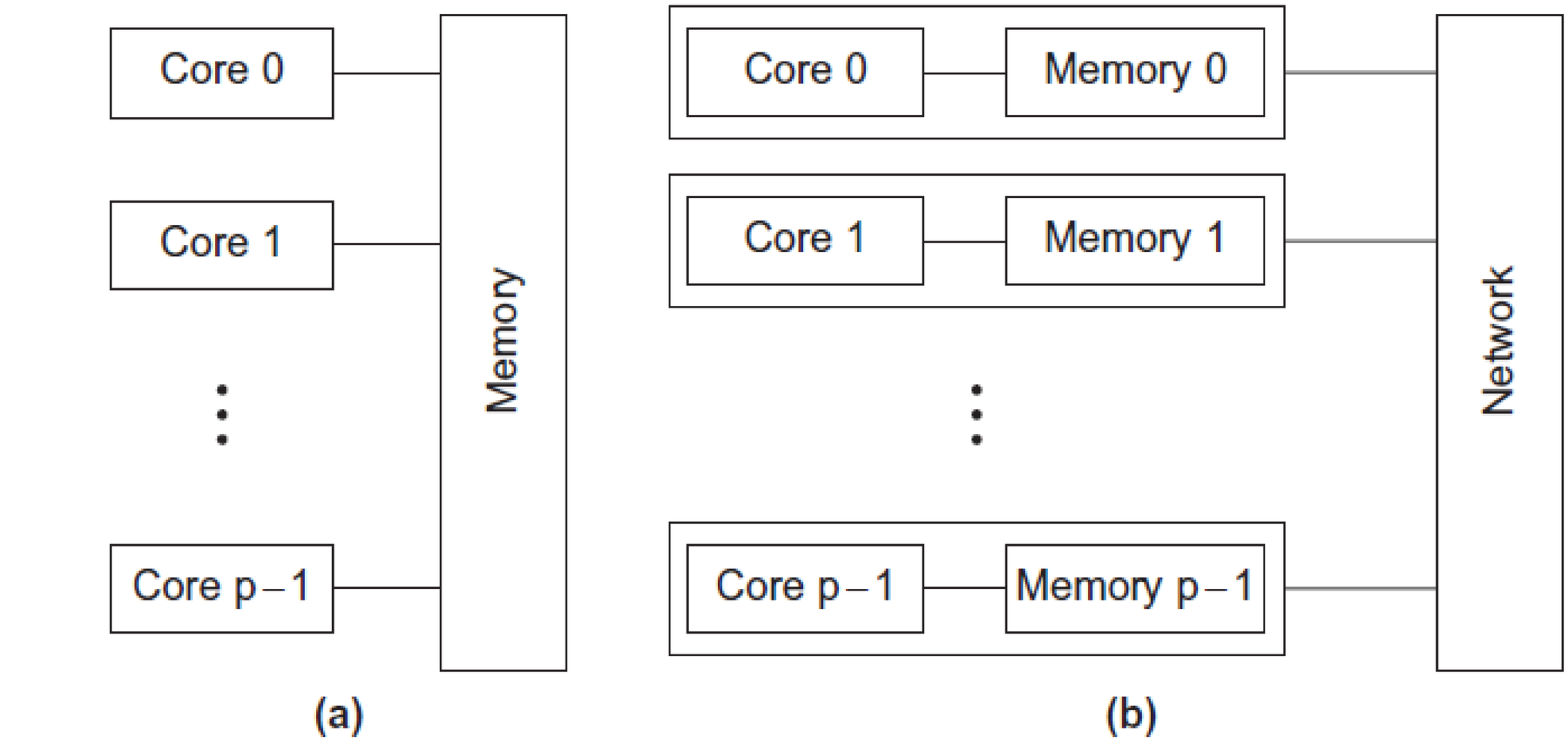
# What we'll be doing

- Learning to write programs that are explicitly parallel.
- Using the C language.
- Using four different extensions to C.
  - Message-Passing Interface (MPI)
  - Posix Threads (Pthreads)
  - OpenMP – ~~TBB - Cilk~~
  - CUDA

# Type of parallel systems

- Shared-memory → biasanya komputer pake ini
  - 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

1 komputer

Distributed-memory

# Terminology

- 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.

# 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.