Pacific Northwest
NATIONAL LABORATORY
Proudly Operated by Battelle Since 1965

Parallel Programming

John Feo

Thursday agenda

Thursday, October 11

Welcome	9:00 – 9:15
Parallel Programming Essentials	9:15 – 10:30
Break	10:30 – 10:45
PPE Lab	10:45 – 12:00
Lunch	12:00 – 1:00
PyCUDA	1:00 – 2:30
Break	2:30 – 2:45
PyCUDA Lab	2:45 – 4:00

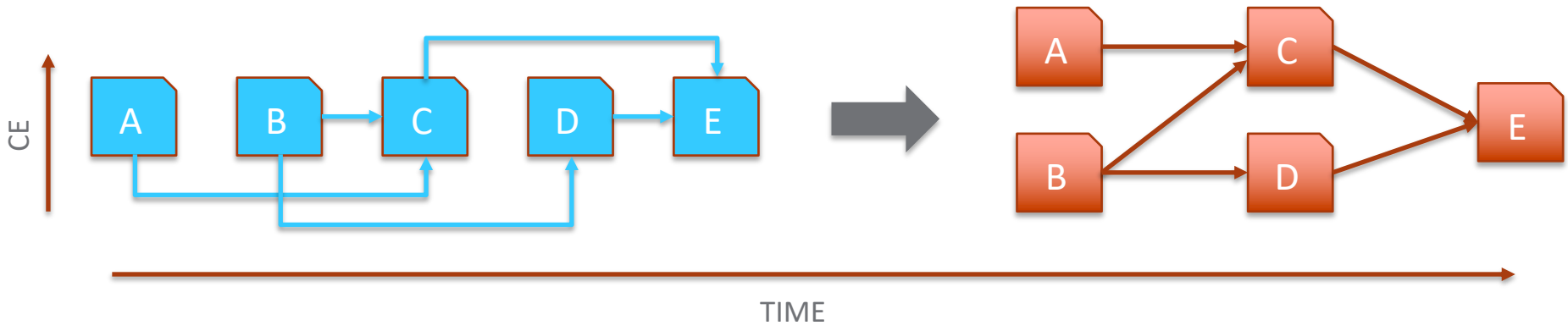
Friday agenda

Friday, October 12

Python Parallel Computing Packages	9:15 – 10:30
Break	10:30 – 10:45
PPCP Lab	10:45 – 12:00
Lunch	12:00 – 1:00
Python to Parallel C++	1:00 – 2:30
Break	2:30 – 2:45
Lab	2:45 – 3:45
Closing Remarks	3:45 – 4:00
Social Event	4:00 – 6:00

What is parallel computing?

- Using multiple computing elements to solve a problem faster
  - Multiple functional units
  - Multiple cores
  - Multiple sockets
  - Multiple nodes
  - Multiple systems
- Requires dividing the program into concurrent units of computation
- … and scheduling the units to computing elements, so as to preserve data and control dependencies



Why parallelism?

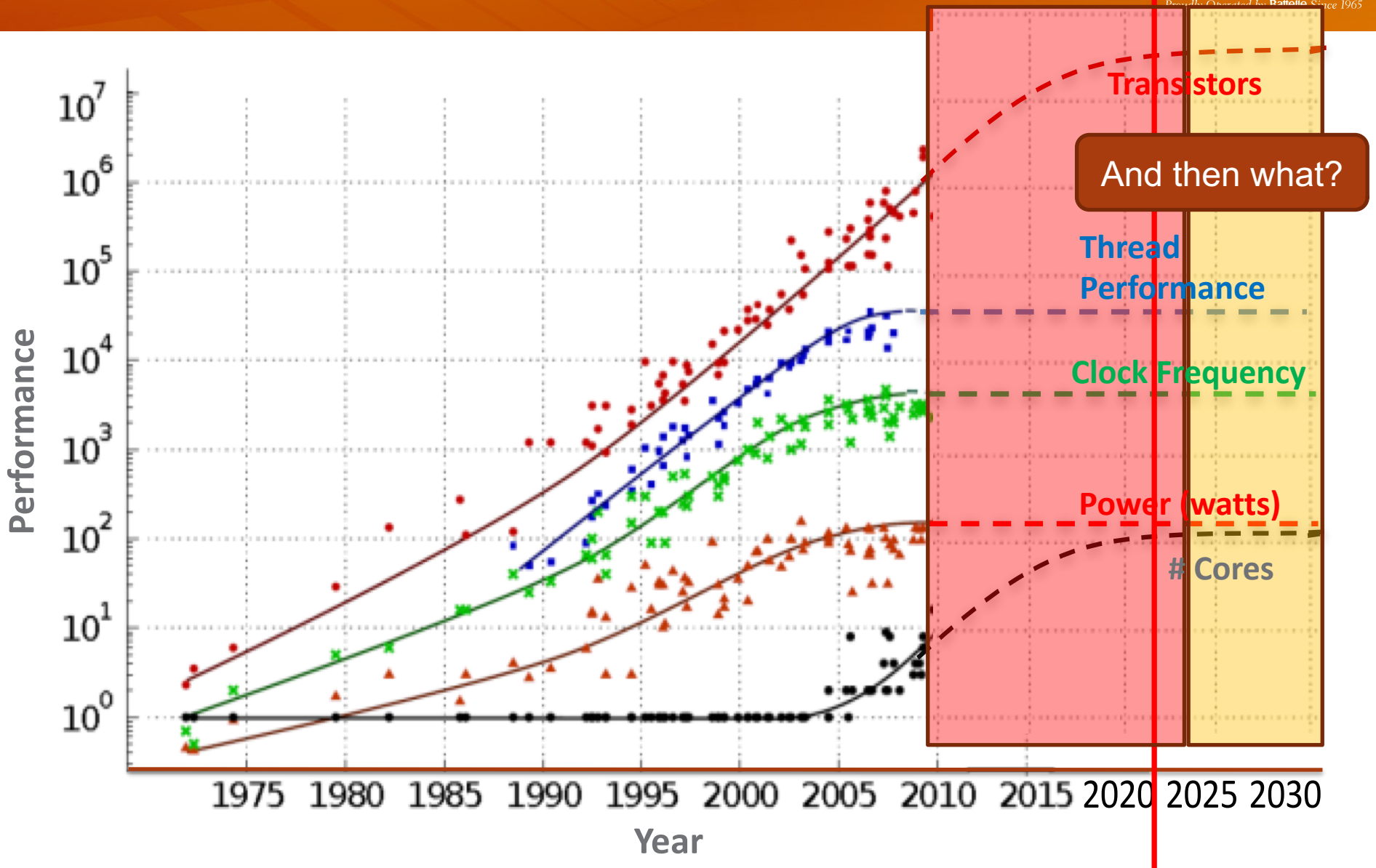
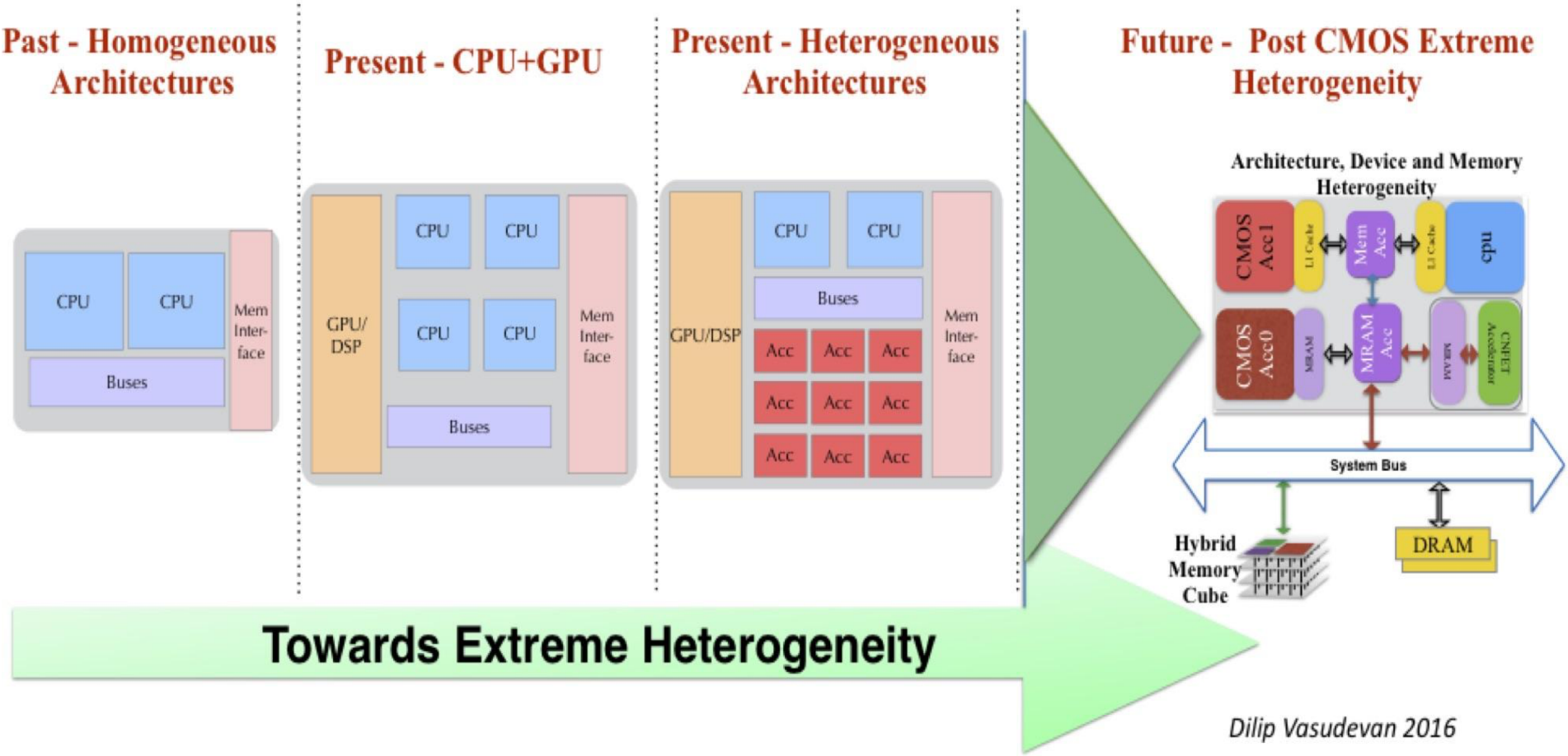


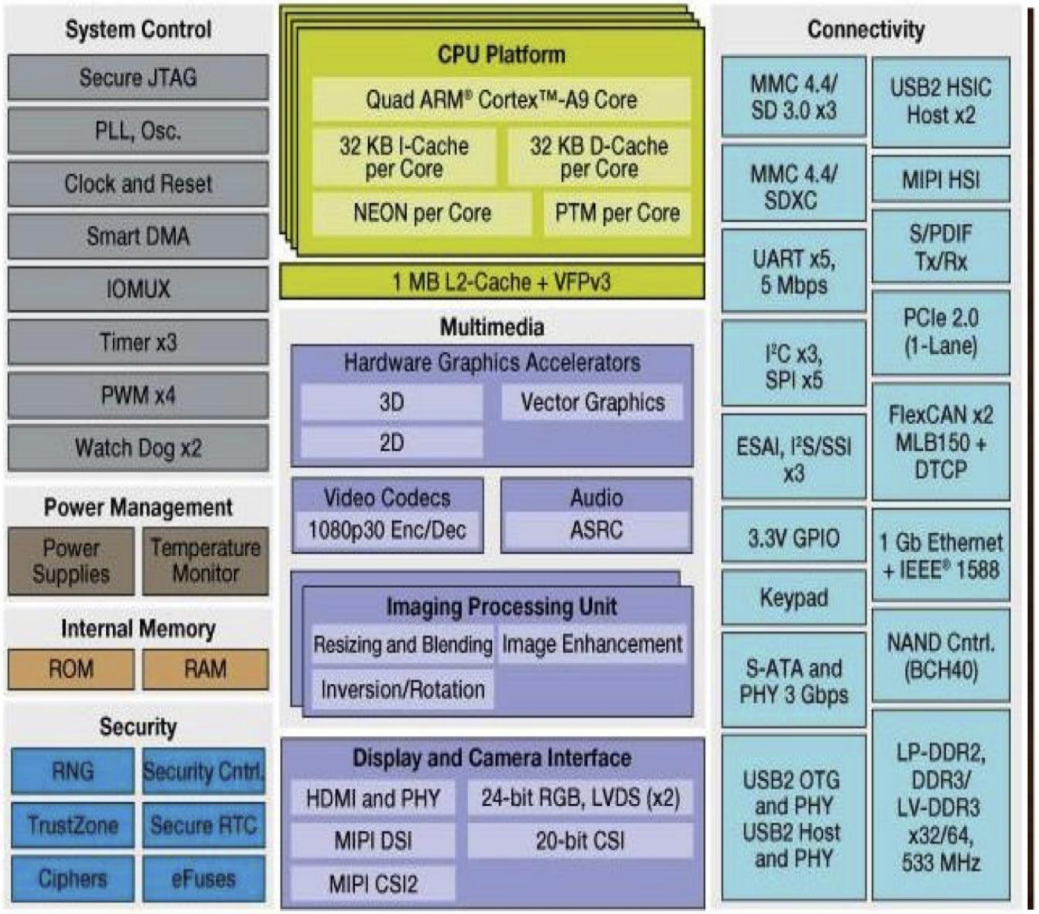
Figure courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter, and Burton Smith

A move to heterogeneity

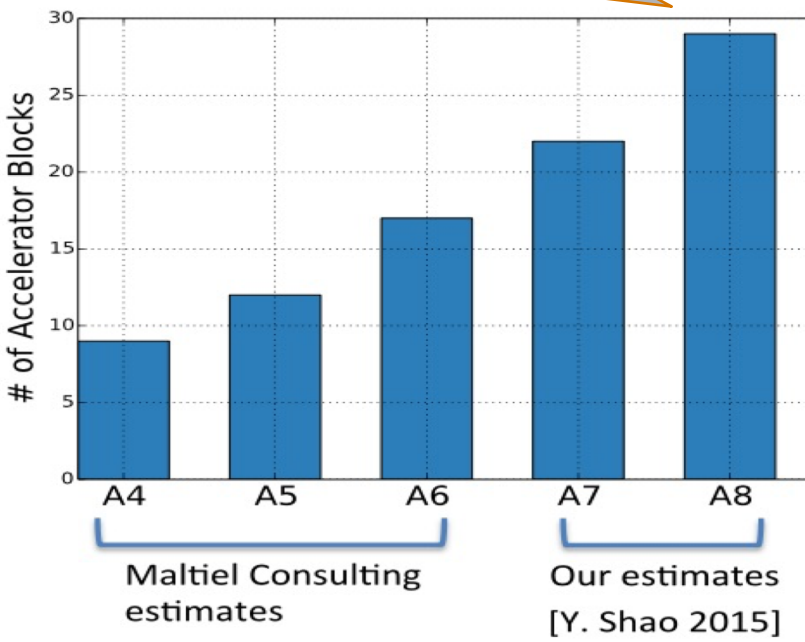


Hardware specialization is happening now

Cell phones and even mega-datacenters (Google TPU, Microsoft FPGAs…)

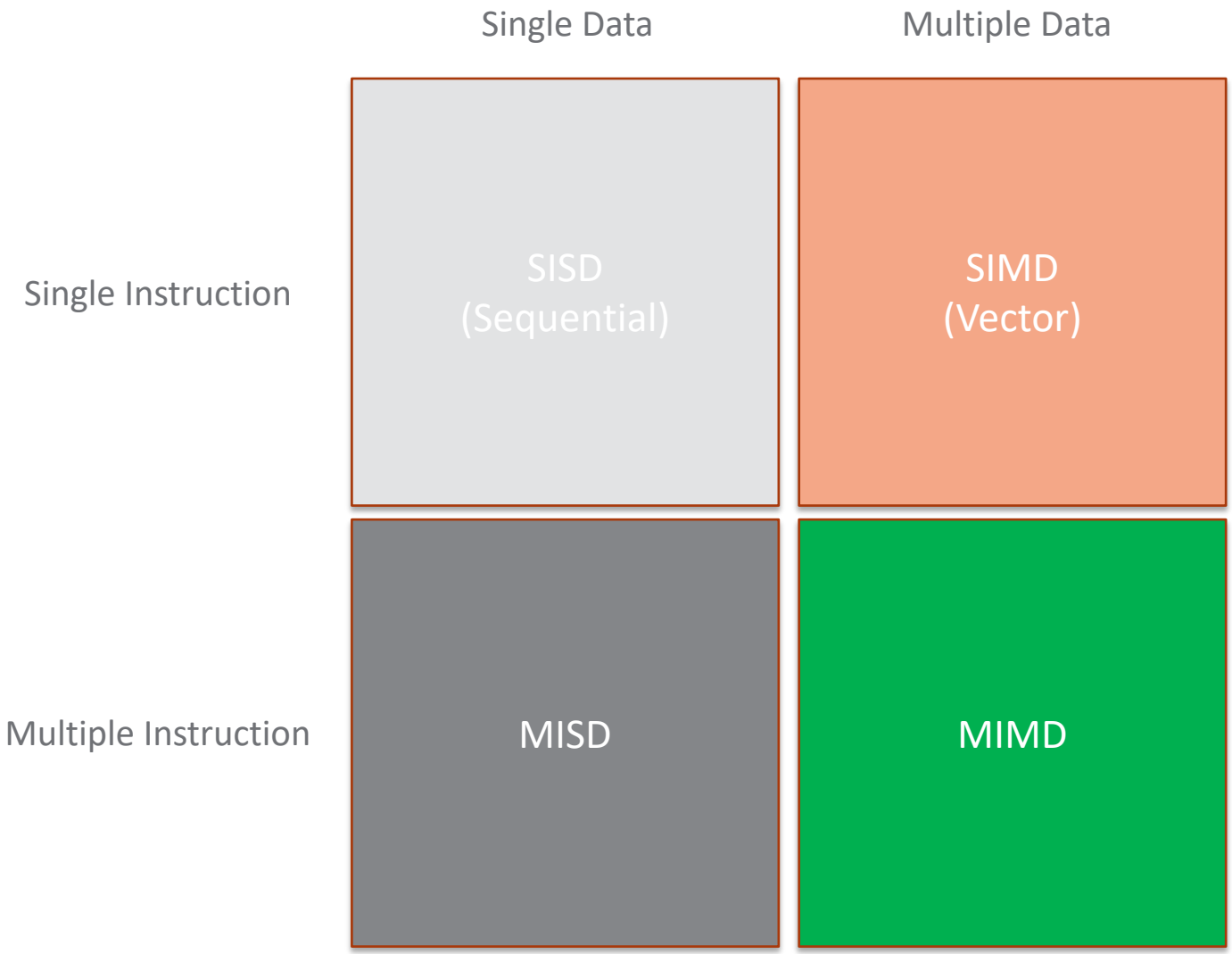


29 different heterogeneous accelerators in Apple A8 (2016)

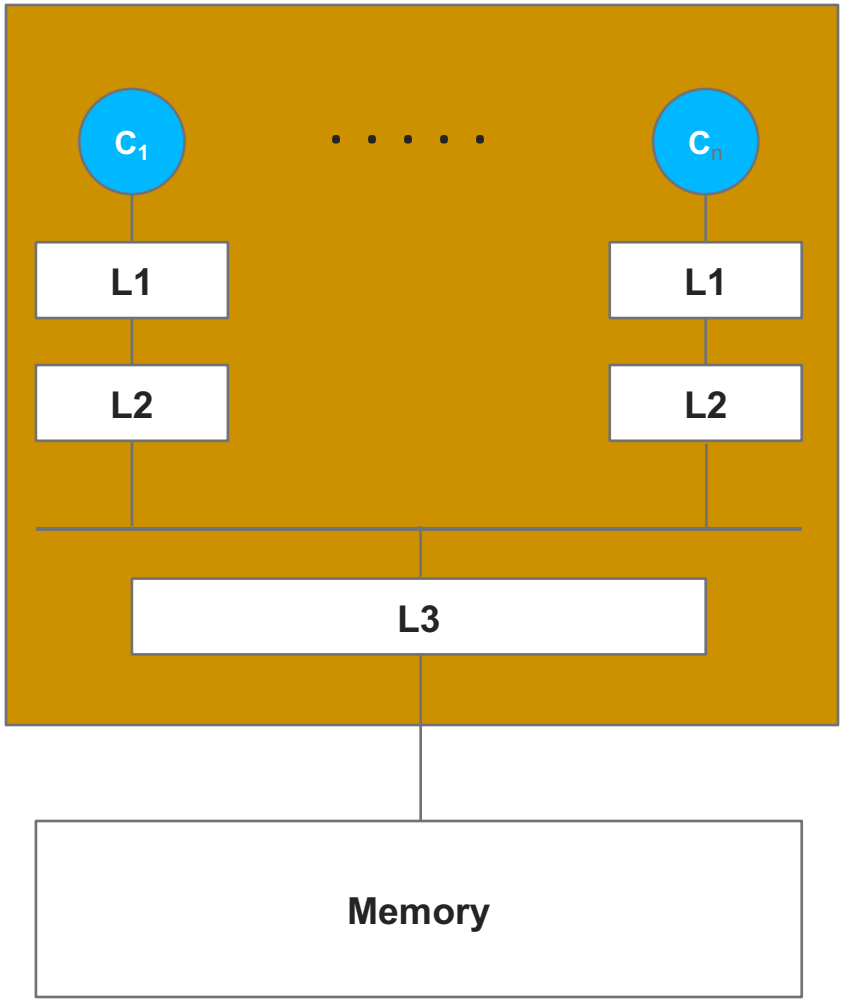


[www.anandtech.com/show/8562/chipworks-a8]

Flynn's taxonomy

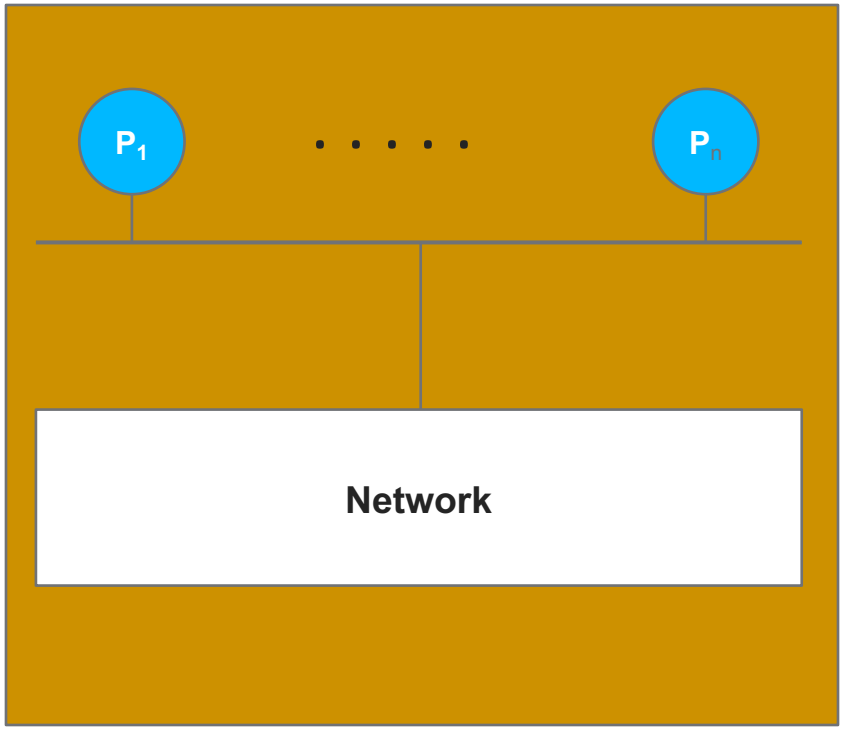


Shared memory system



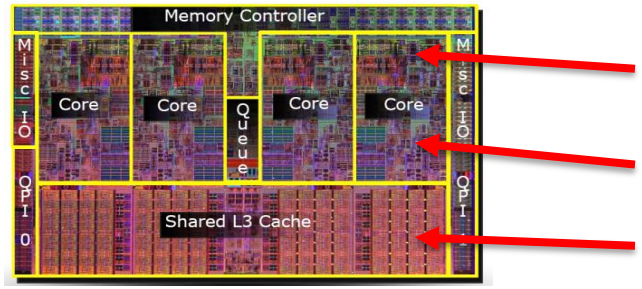
- **The Good**
  - Read and write any data item
    - No partitioning
    - No message passing
  - Reads and write performed by hardware
    - Little overhead → lower latency, higher bandwidth
- **The Bad**
  - Read and write any data item
    - *Race conditions*

Distributed memory system



- **The Good**
  - Huge systems
  - Processors ”own” their data
    - No race conditions
- **The Bad**
  - Message passing
    - High latency
    - Large data transfers
  - Global synchronization
  - Need to partition tasks and data
    - Static partitioning
    - Load imbalances

Memory latencies



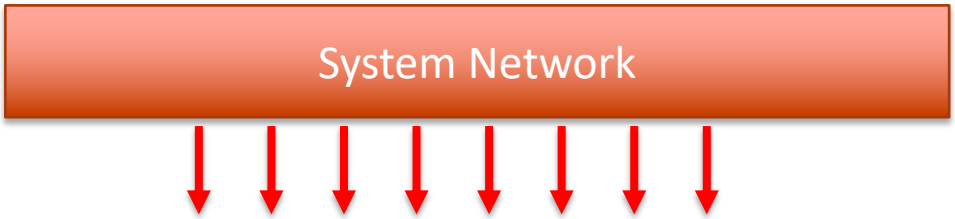
On-chip L1 cache — 1 cycle

On-chip L2 cache — 3 cycles

On-chip L3 cache — 10 cycles

Main memory — 60 cycles

200 cycles



10000 cycles

Task parallelism

COOK ITALIAN DINNER PROGRAM

CREATE GUEST LIST

CHOOSE MENU

BUY FOOD

PREPARE APPETIZER

PREPARE PASTA COURSE

PREPARE MAIN COURSE

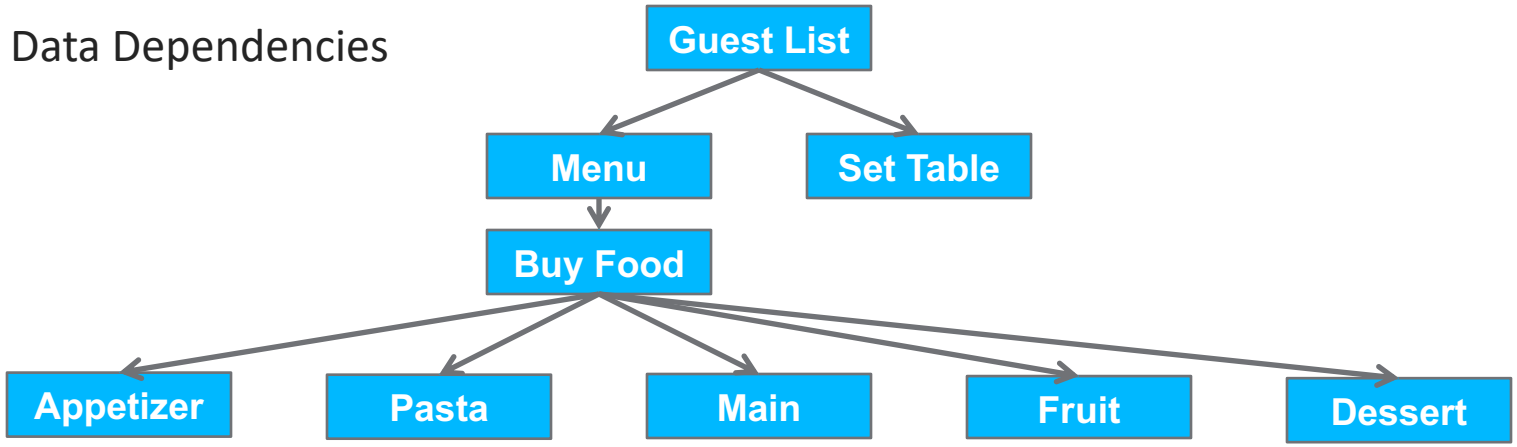
PREPARE FRUIT

PREPARE DESSERT

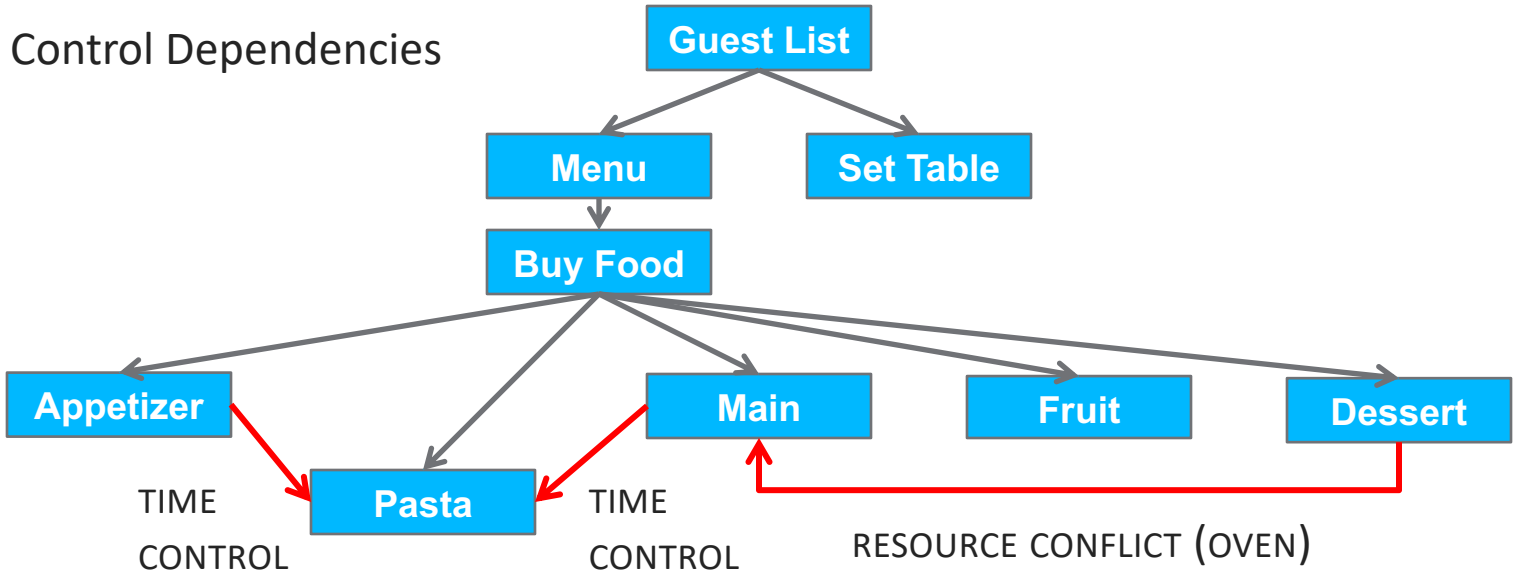
SET TABLE

Dependencies

Data Dependencies



Control Dependencies



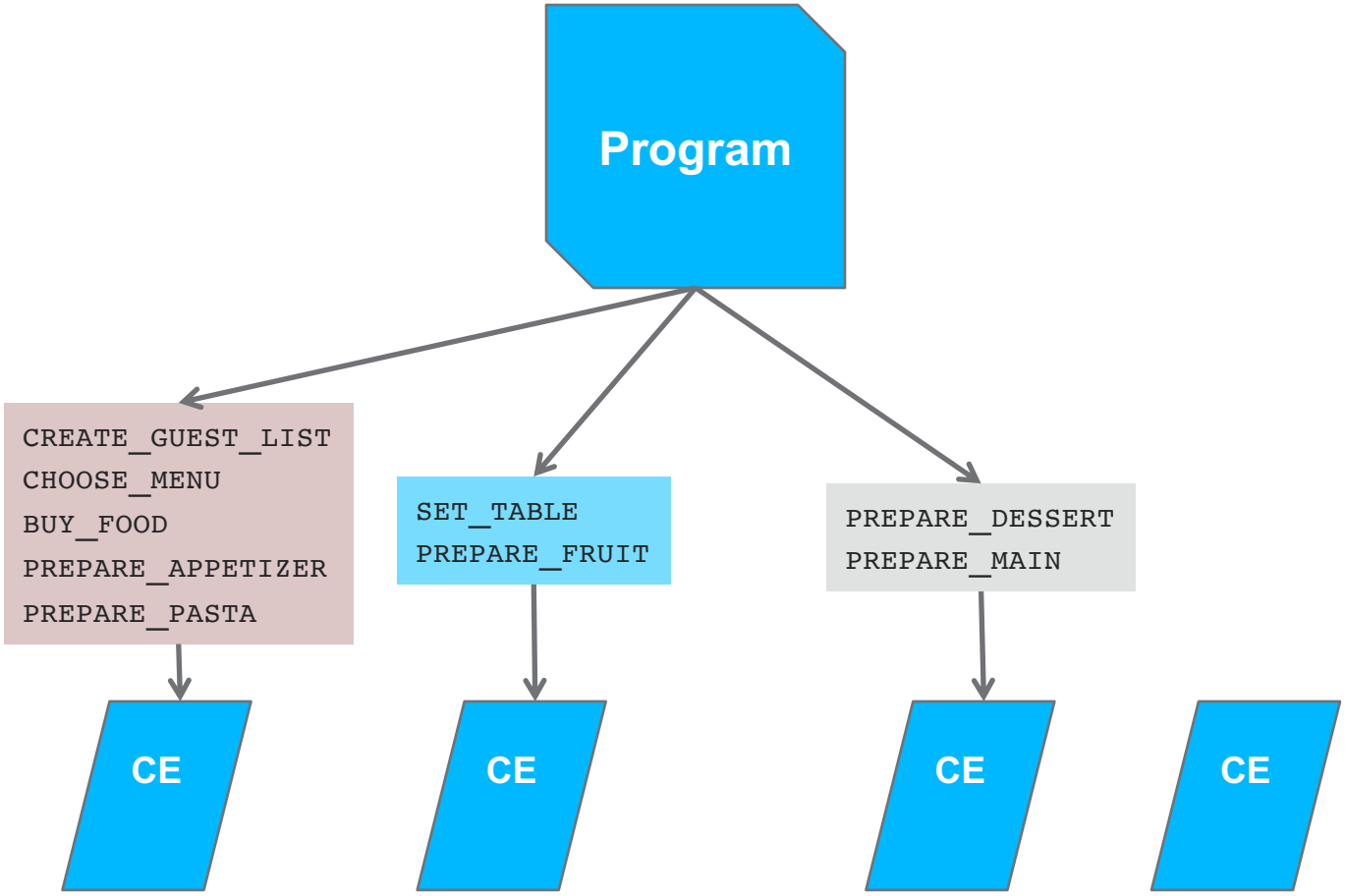
Co-begin


```
VOID COOK_ITALIAN DINNER () {

  GUESTS = CREATE_GUEST_LIST();                          T1
  MENU   = CHOOSE_MENU(GUESTS);                          T2

  CO_BEGIN {
    FOOD = BUY_FOOD(GUESTS, MENU);                       T3
    SET_TABLE(GUESTS);
  }

  CO_BEGIN {
    PREPARE_APPETIZER(GUESTS, MENU, FOOD);               T4
    PREPARE_FRUIT(GUESTS, MENU, FOOD);
    { PREPARE_DESSERT(GUESTS, MENU, FOOD);               T4a
      PREPARE_MAIN(GUESTS, MENU, FOOD);                  T4b
    }
  }

  PREPARE_PASTA(GUESTS, MENU, FOOD);                     T5
}
```

Threads

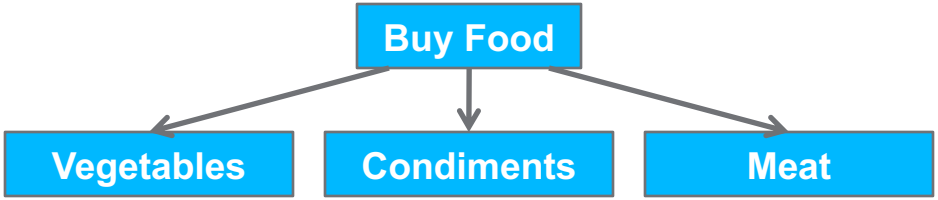
- A set of instructions assigned to a computing element for execution
- **THREADS ARE SOFTWARE**
- Threads are a schedulable units of computation
- Threads can be defined by
  - The user
  - The compiler
  - The runtime systems
- Threads can be created at compile time or runtime
- Threads are scheduled by the runtime system

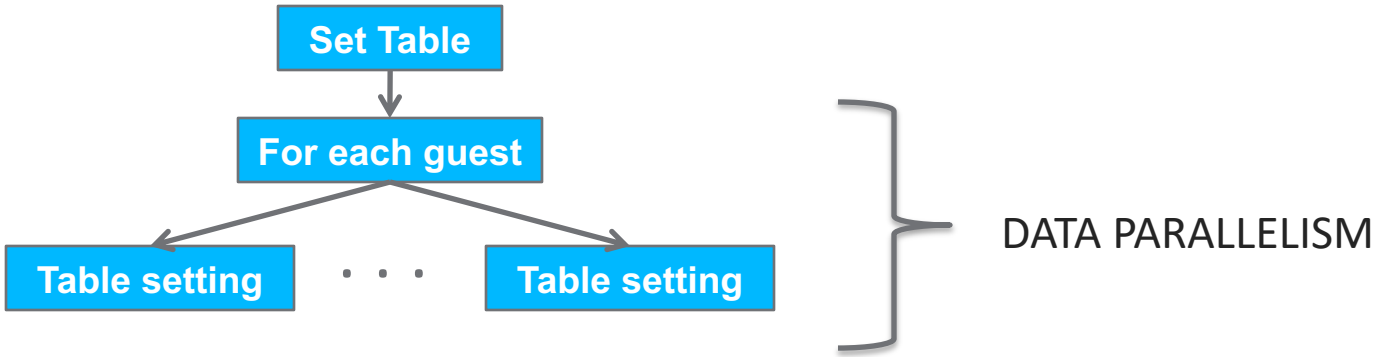
Scheduling threads



More parallelism

- You want to write your program, so that there is enough parallelism to keep the system busy





Data parallelism

- Loop over data elements
- Work on each element in parallel
- Best source of parallelism in almost all programs
- Always think data parallel first

```
FOR ALL GUESTS, GUESTS[i]
```

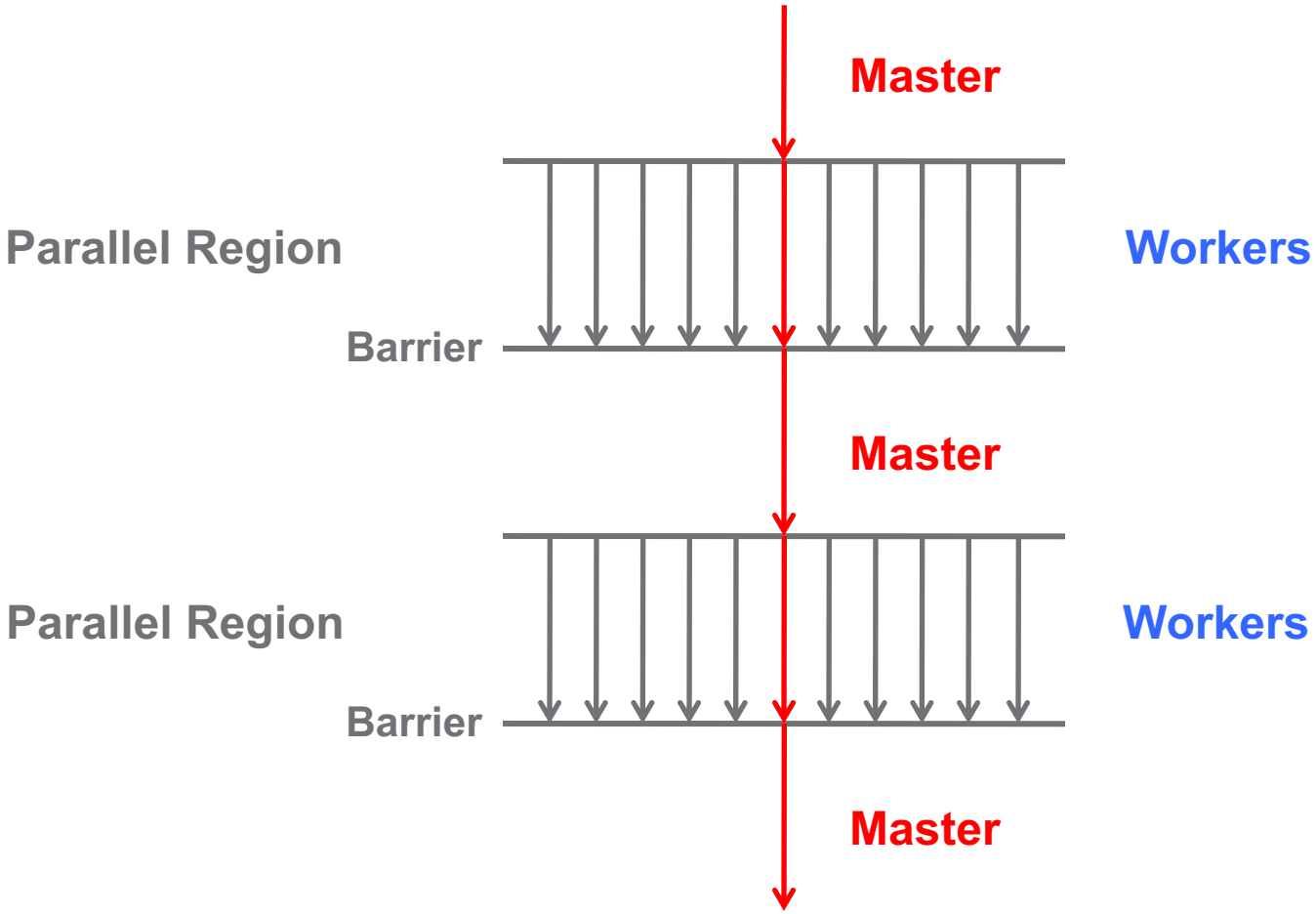
```
FOR ALL DOCUMENTS, LIBRARY[i]
```

```
FOR ALL ATOMS, ATOMS[i]
```

```
FOR ALL NODES, GRAPH[i]
```

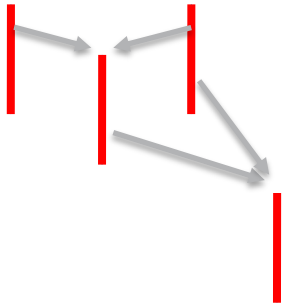
```
FOR ALL ELEMENTS, A[i, j]  // ORDER N^2 PARALLELISM
```

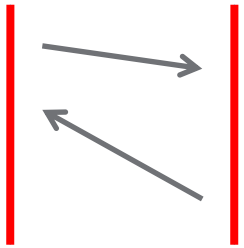
Fork-join model

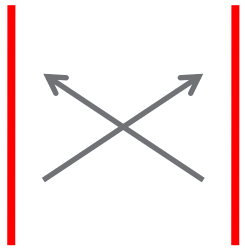


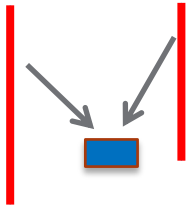
Strict, non-strict threads

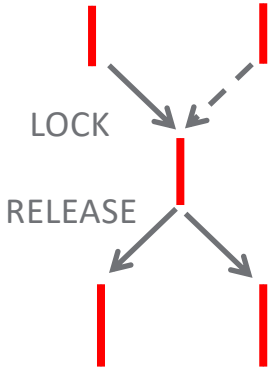
- Strict – all inputs available before thread is scheduled
  - No preemption
  - Synchronization only at thread boundaries
- Non-strict – all inputs may not be available
  - Threads may stall waiting for data
  - Deadlocks are possible
  - Critical sections
  - AMOs











What happens at the end of a parallel region?

- The thread that spawns the region's work (*the Master*) may
  - Continue to next instruction (*asynchronous*)

```
for num in range(10):
    t = multiprocessing.Process(target=f, args=(num))
    t.start()
```

  - Suspend until all spawned threads terminate (*synchronous*)

```
processes = [ ]
for num in range(10):
    t = multiprocessing.Process(target=f, args=(num))
    processes.append(t)
    t.start()

for one_process in processes:
   one_process.join()
```

Example of critical section


```
#pragma omp parallel for
for (i = 0; i < nBonds; i++) {
    int headNode = Bond[i][0];
    int tailNode = Bond[i][1];

    … calculate x, y, z force of bond i …

    #pragma omp critical
    {
     Force[headNode][0] += … x force …
     Force[headNode][1] += … y force …
     Force[headNode][2] += … z force …
    }
  ……
}  /* End of parallel for loop – Synchronous */
```

Example of atomic


```
#pragma omp parallel for
for (i = 0; i < nBonds; i++) {
  int headNode = Bond[i][0];
  int tailNode = Bond[i][1];

  … calculate x, y, z force of bond i …

  #pragma omp atomic
  Force[headNode][0] += … x force …

  #pragma omp atomic
  Force[headNode][1] += … y force …

  ……

} /* End of parallel for loop – Synchronous */
```

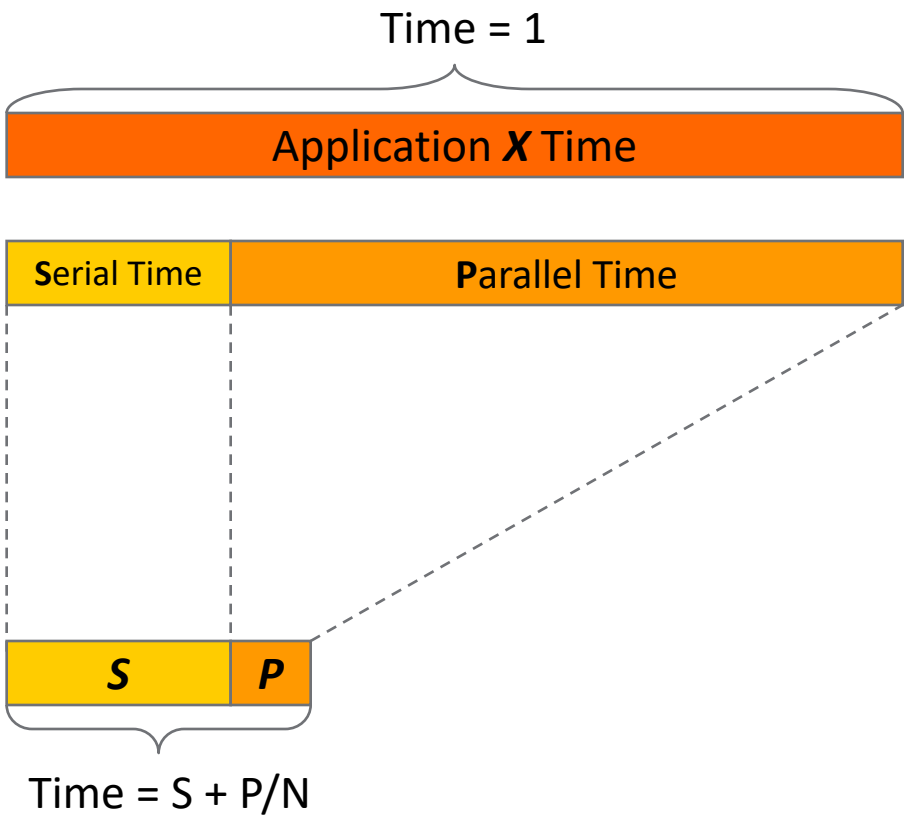
Reductions


```
int count3s(int length, int *A) {
  int i, count = 0;

  #pragma omp parallel
  {
    int count_p = 0;

    #pragma omp for
    for (i = 0; i < length; i++) {
       if (A[i] == 3) count_p ++;

    #pragma omp critical
    { count += count_p; }
  }

  return count;
}
```

Reductions


```
int count3s(int length, int *A) {
  int i;

  #pragma omp parallel for reduction(+: count)
    for (i = 0; i < length; i++) {
       if (A[i] == 3) count ++;
    }

  return count;
}
```

Speedup – Amdahl's Law



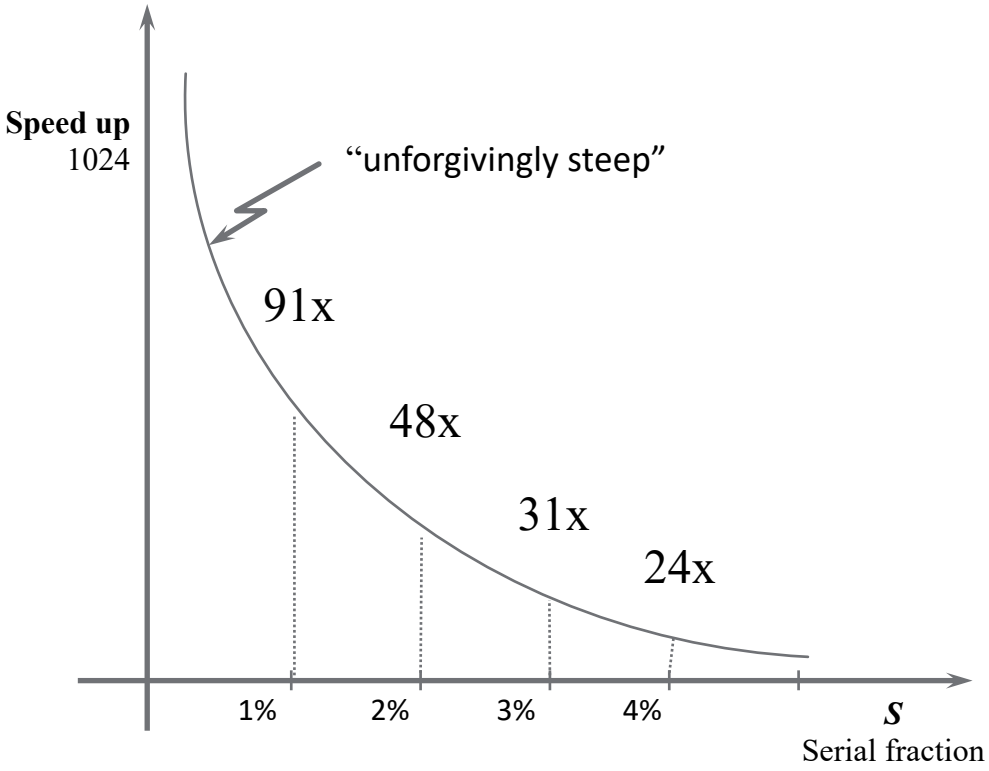
S + P = 1

$$\frac{T_1}{T_N} = \frac{S+P}{S+\frac{P}{N}} = \frac{1}{S+\frac{P}{N}}$$

$$\lim_{N\to\infty} \frac{1}{S+\frac{P}{N}} = \frac{1}{S}$$

Strong scaling

- Keep problem size fixed as you increase the number of processors



Weak scaling

- Increase problem size as you increase the number of processors
- Linear scaling ➔ execution time remains constant
- Makes sense because I want to run bigger problems on bigger systems; however, some problems cannot get bigger
  - Weather per square kilometer (…maybe)
  - Weather per square meter (… NO!!)
- Usually, as problem size grows, parallel part grows faster than sequential part, so weak scaling can circumvent Amdahl's Law

Problem Size	= P
# processors	= P
Serial work	= P
Parallel work	= P * P

$$\frac{T_1}{T_P} = \frac{P + P * P}{P + \frac{P * P}{P}} = \frac{P * (1 + P)}{2 * P} = O(P)$$