Parallel Computing

Academic year – 2020/21, spring semester

Computer science

Lecture 2

Lecturer, instructor:

Balakshin Pavel Valerievich

(pvbalakshin@itmo.ru; pvbalakshin@hdu.edu.cn)

Assistant:

TBD

(TBD@hdu.edu.cn)

A typical scenario of a parallel programmer's work

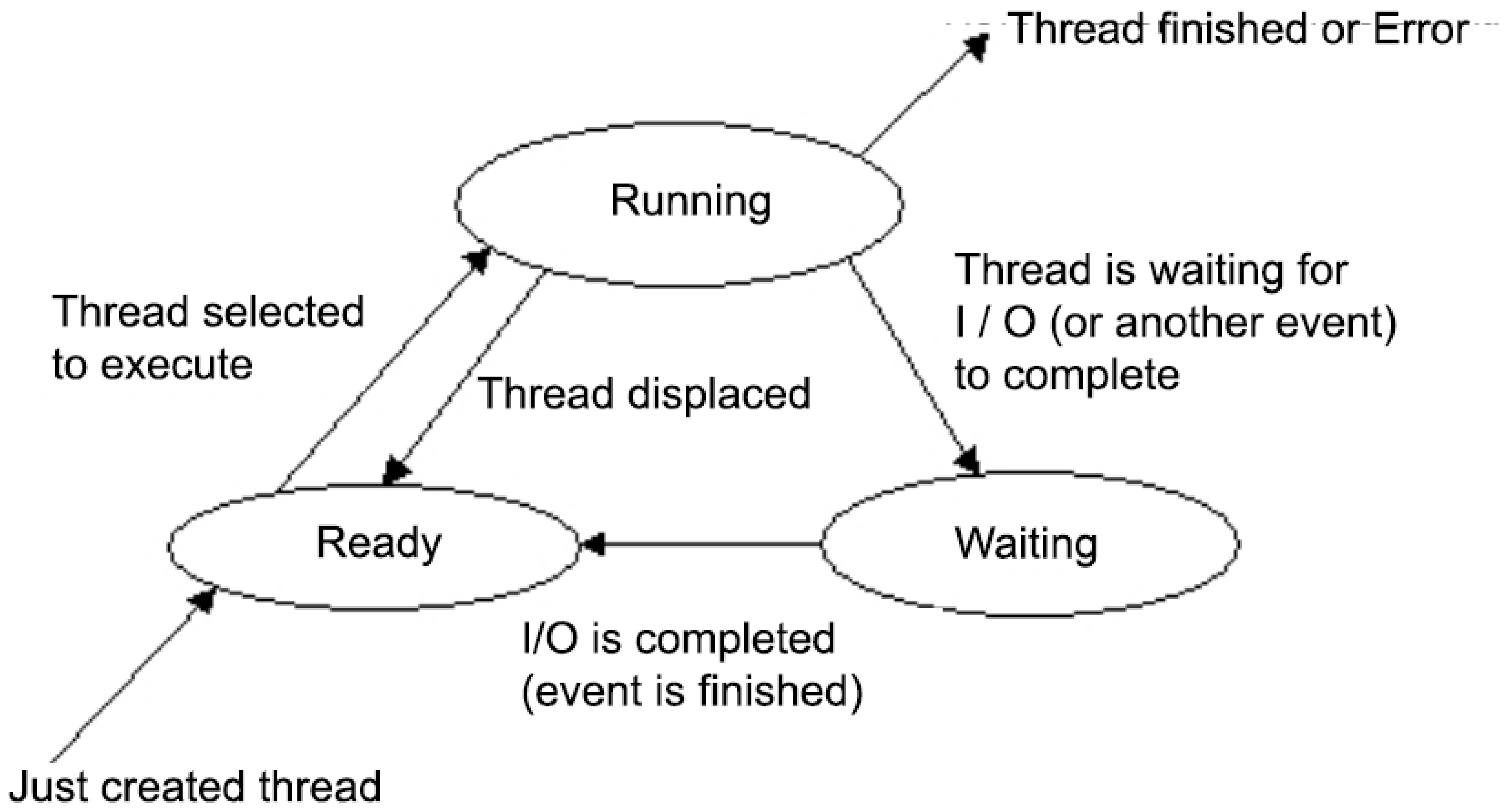
1. Write a single threaded program.
2. Stop.
3. Try automatic paralleling.
4. Stop.
5. Try parallel libraries and data structures.
6. Stop.
7. Profile the application, estimate the maximum achievable paralleling effect.
8. Stop.
9. Parallelize bottlenecks and frequently called functions.

The goal of parallel programming is to avoid parallel programming!

Parallel programming terminology

- Process – the most heavyweight mechanism used for parallelization. Each process has its own independent address spaces, so data synchronization between processes is slow and complicated. May include several threads of execution.
- Thread – runs independently of other threads, but has a common address space with other threads into the same process. At this level data synchronization mechanisms are used.
- Fiber – lightweight thread of execution. Like threads, fiber has a common address space, however, it uses joint multitasking instead of preemptive one. All fibers work on one core, unlike threads, which can work on different kernels.

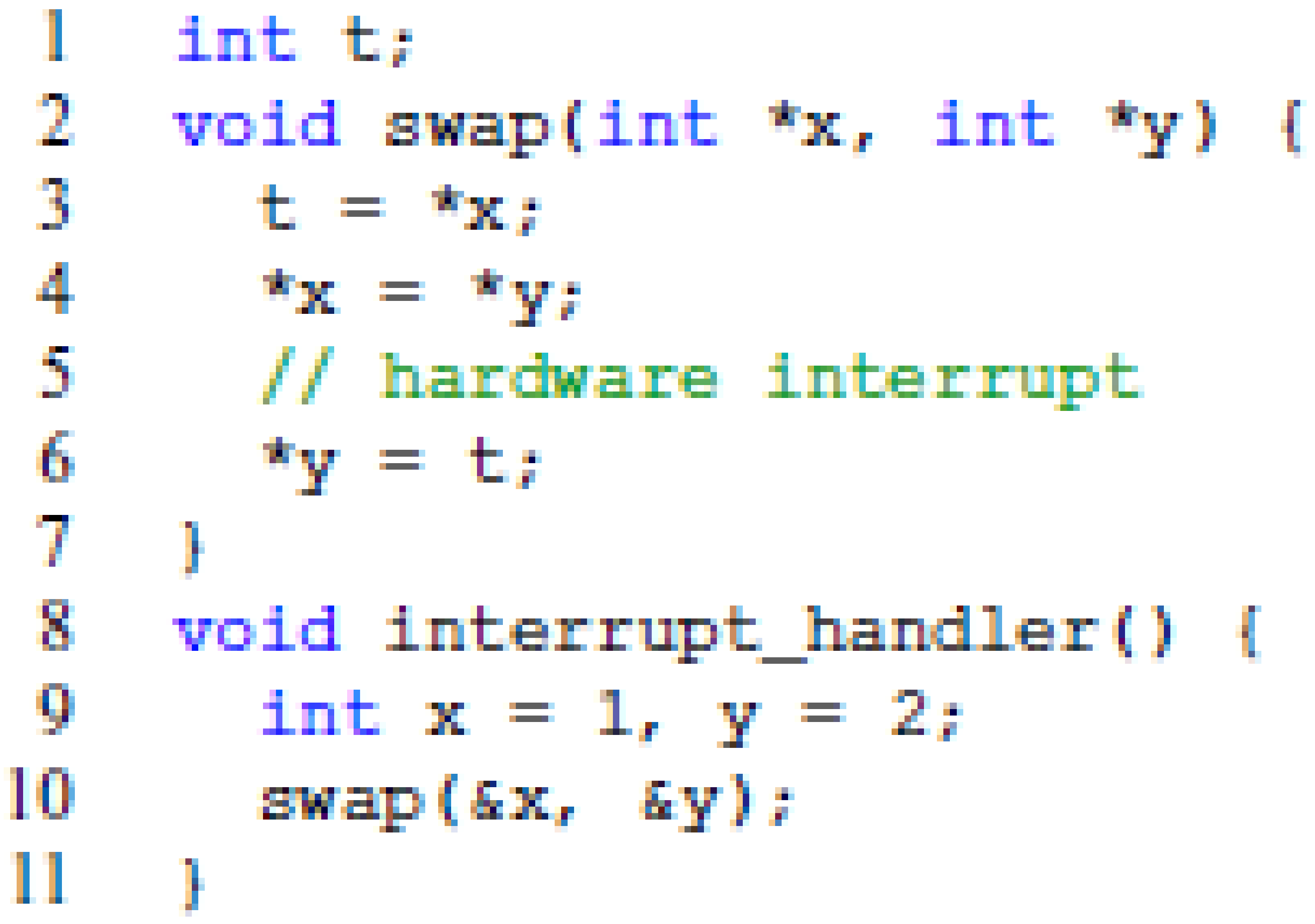
Thread lifecycle


Thread finished or Error
Running
Thread selected to execute
Thread displaced
Thread is waiting for I / O (or another event) to complete
Ready
Waiting
I/O is completed (event is finished)
Just created thread

Parallel programming terminology (2)

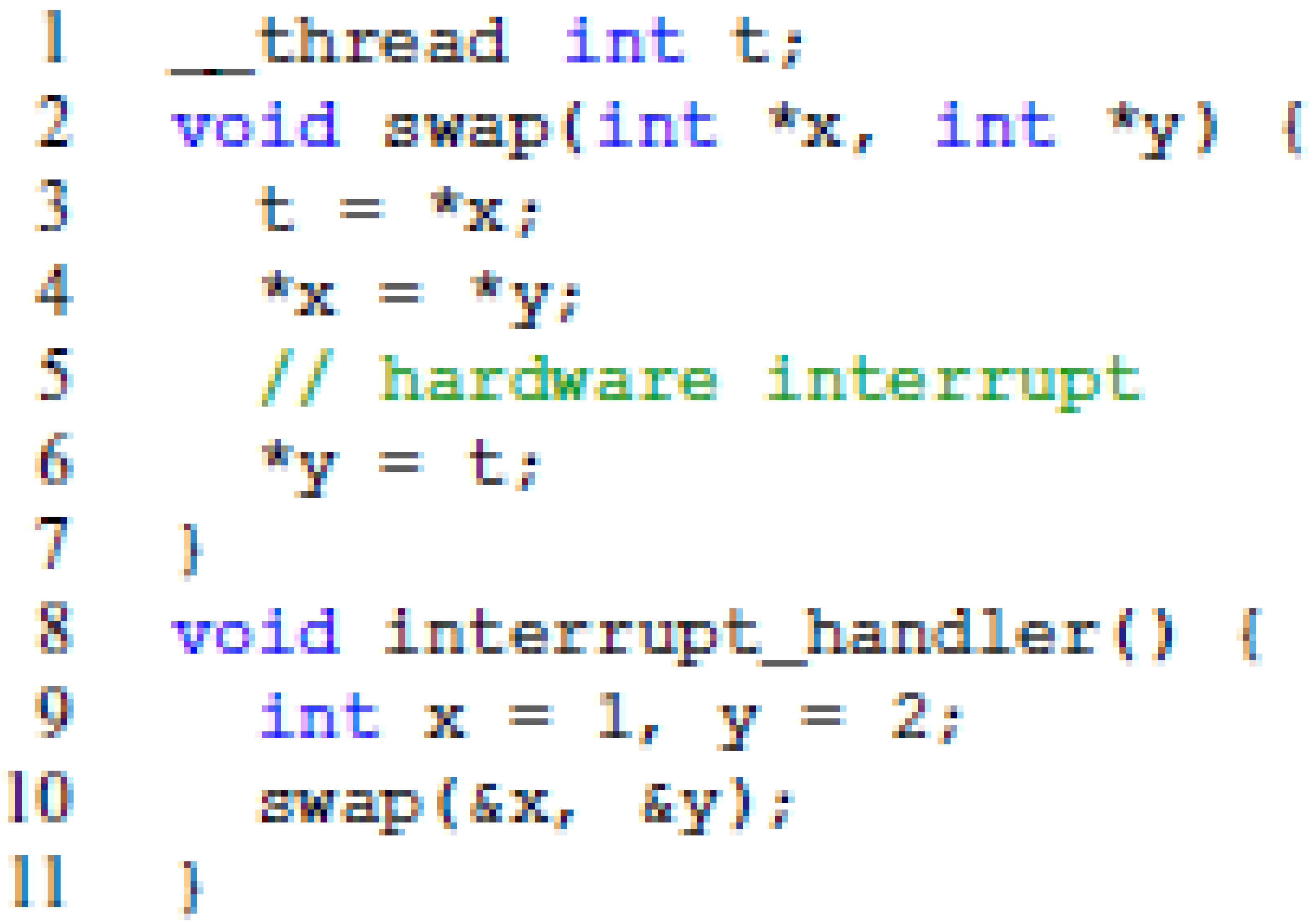
Source of definition	Thread-safe	Reentrant / reentrable
Qt	Inside the function all common variables are addressed strictly sequentially, but not in parallel (thread-safe is reentrant but not vice versa).	Function is called by several threads at the same time, correct operation is gua-ranteed only if the threads do not use shared data.
Linux	The function shows the correct results, even if called by several threads at the same time.	The function shows the correct results, even if It recalled internally.
POSIX	?	The function shows correct results even if it is called by several threads at the same time.

Not thread-safe, not reentrant



```c
int t;
void swap(int *x, int *y) {
  t = *x;
  *x = *y;
  // hardware interrupt
  *y = t;
}
void interrupt_handler() {
  int x = 1, y = 2;
  swap(&x, &y);
}
```

Thread-safe, not reentrant

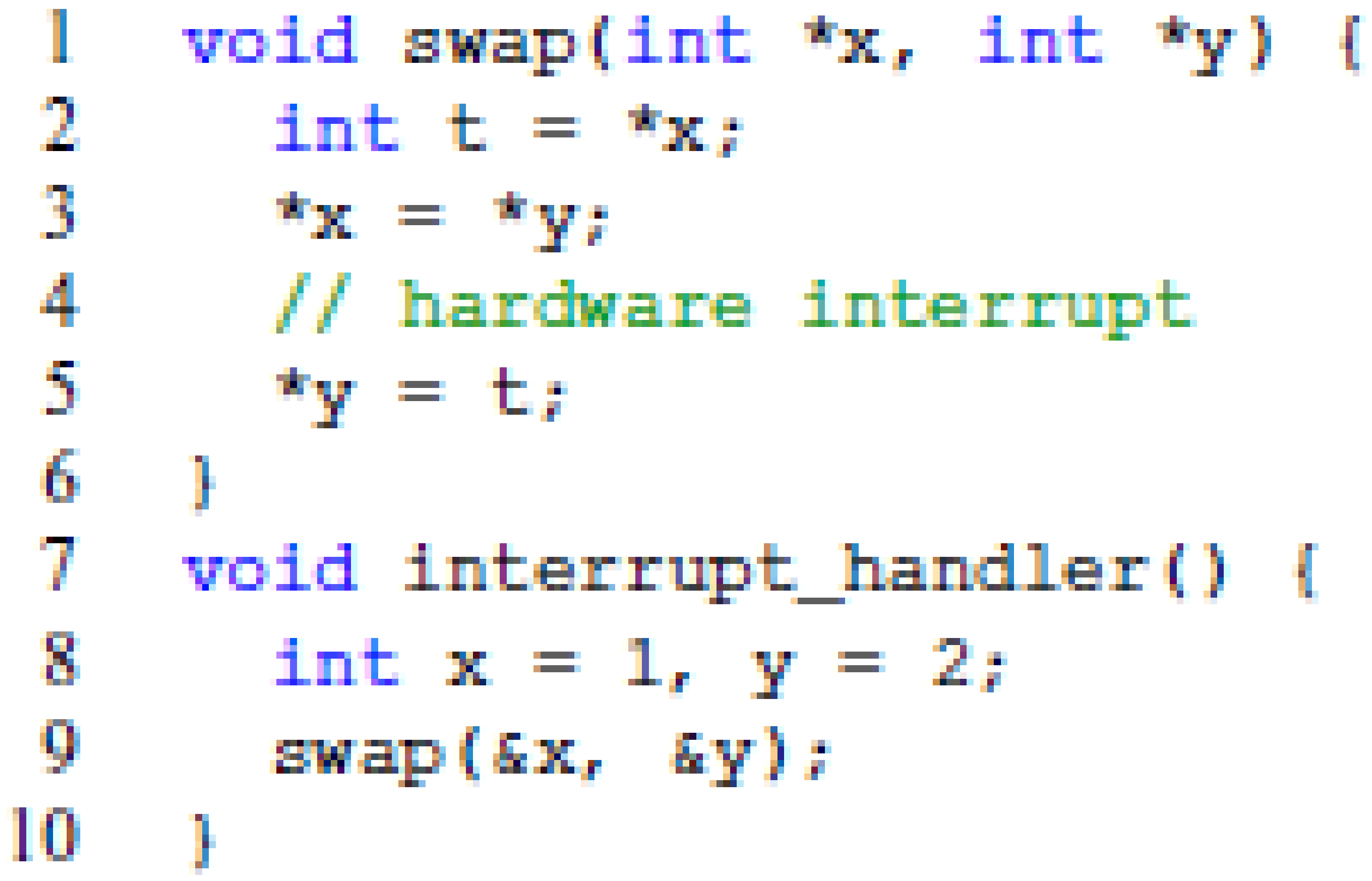


```
__thread int t;
void swap(int *x, int *y) {
  t = *x;
  *x = *y;
  // hardware interrupt
  *y = t;
}
void interrupt_handler() {
  int x = 1, y = 2;
  swap(&x, &y);
}
```

Not thread-safe, reentrant

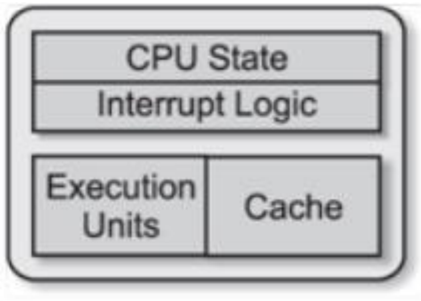
```c
int t;
void swap(int *x, int *y) {
  int s;
  s = t; // save global variable
  t = *x;
  *x = *y;
  // hardware interrupt
  *y = t;
  t = s; // restore global variable
}
void interrupt_handler() {
  int x = 1, y = 2;
  swap(&x, &y);
}
```

Thread-safe, reentrant

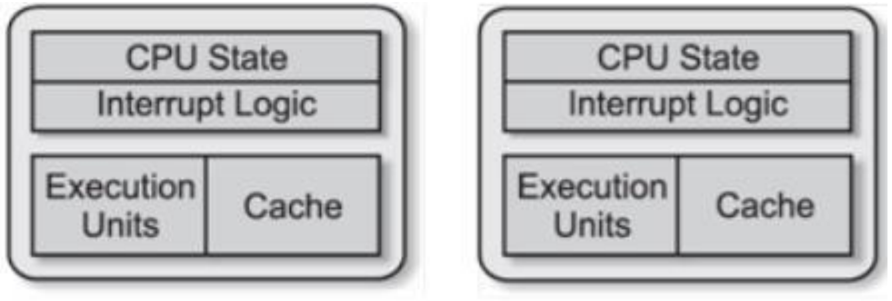


```c
void swap(int *x, int *y) {
  int t = *x;
  *x = *y;
  // hardware interrupt
  *y = t;
}
void interrupt_handler() {
  int x = 1, y = 2;
  swap(&x, &y);
}
```

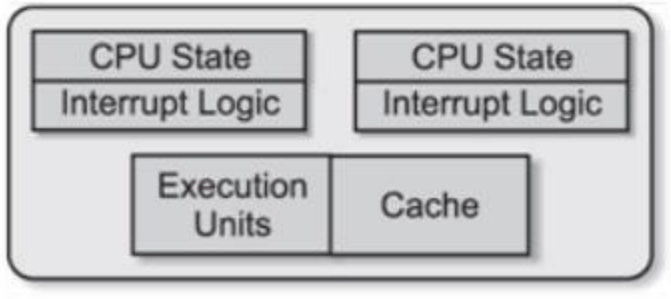
Hardware parallelism types



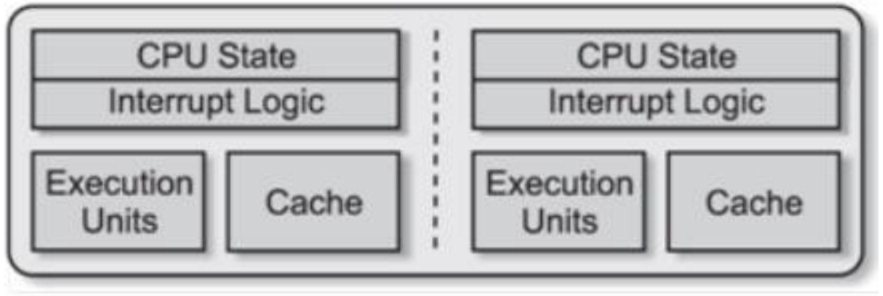
A) Single Core



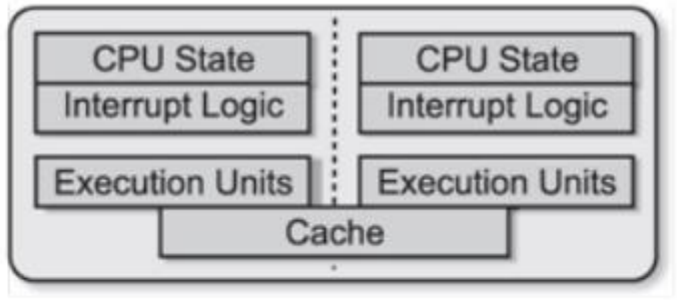
B) Multiprocessor



C) Hyper-Threading Technology



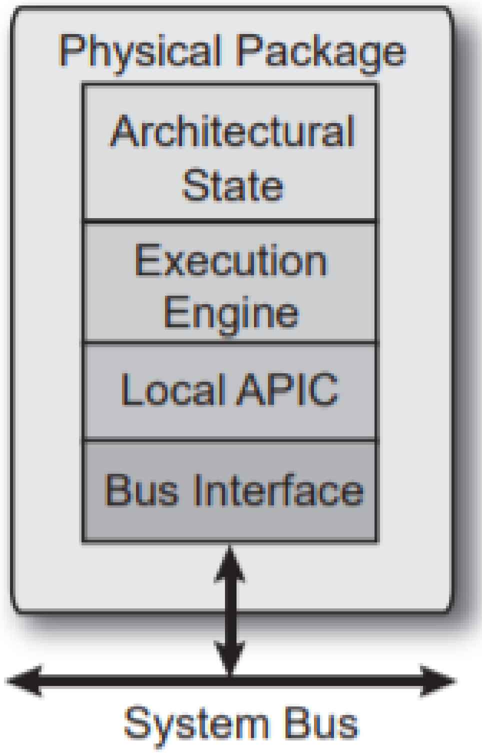
D) Multi-core



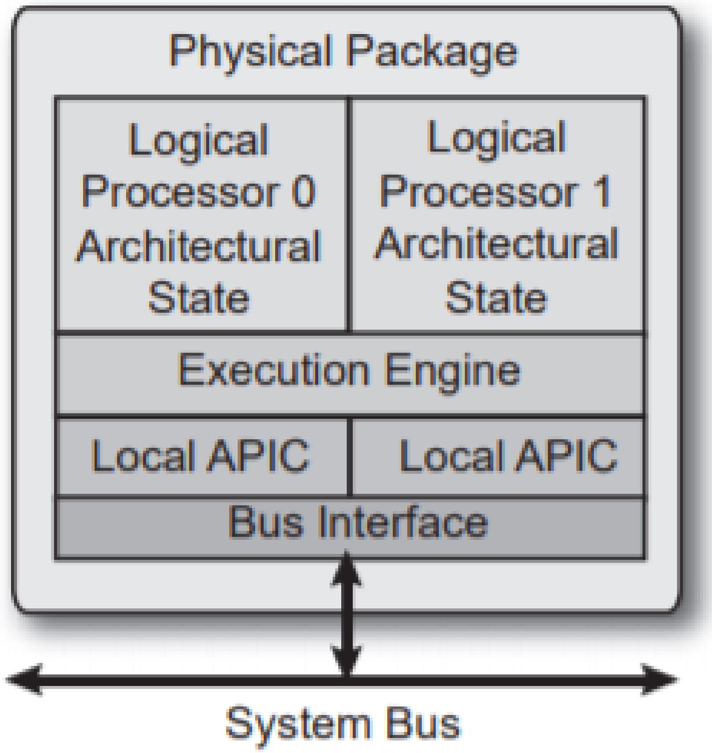
E) Multi-core with Shared Cache

According to S. Akhter “Multi-Core Programming”

Hyperthreading



Single Processor


Physical Package
Logical Processor 0 Architectural State
Logical Processor 1 Architectural State
Execution Engine
Local APIC
Local APIC
Bus Interface
System Bus

Single Processor with Hyper-Threading Technology

According to S. Akhter "Multi-Core Programming"

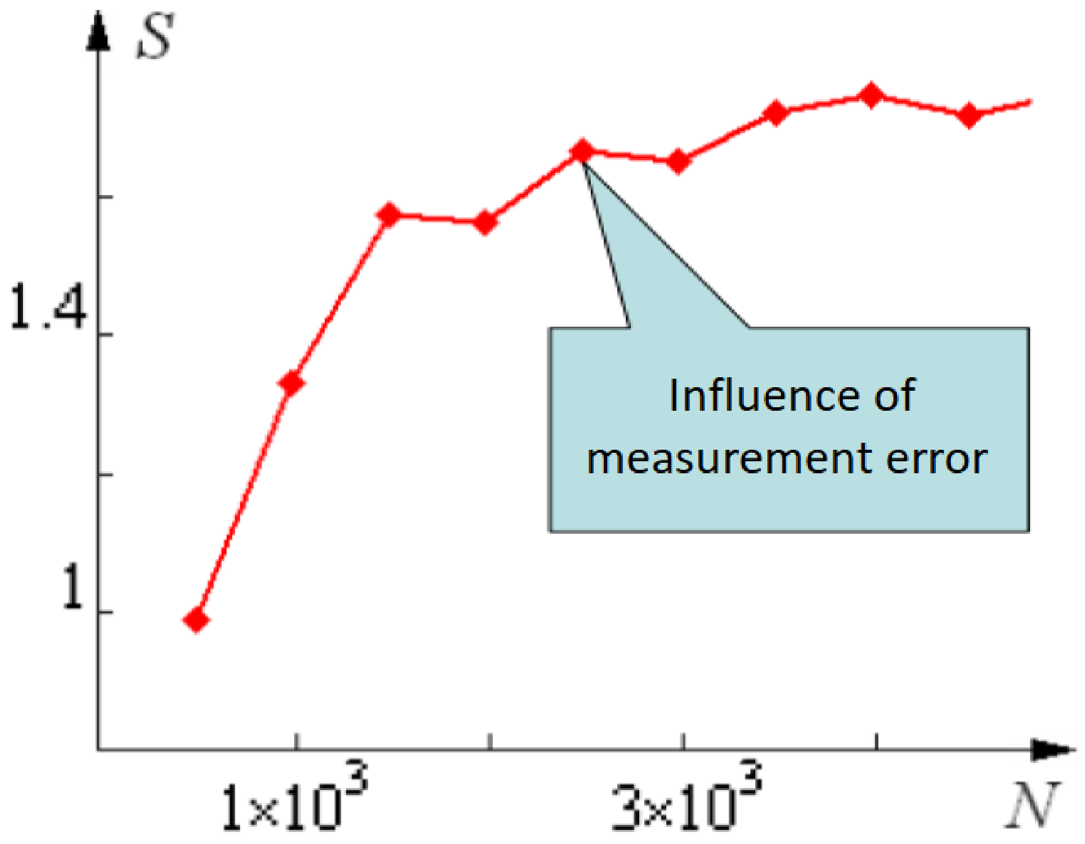
Multithreading on single-processor systems vs. multiprocessor systems

- On a single-processor (single-core) system you can use thread priorities to arrange their access to the processor (on a multi-processor system you cannot do it).
- On a single-processor system you need to pack structures more tightly to optimize the memory cache, and on a multi-core system you need to cut them by the size of the cache string to prevent False Sharing.

Parallel programming problems

- Measuring parallel efficiency.
- Instability of floating point calculation results.
- Race conditions.
- Mutual interlocks.
- The ABA problem.
- Inversion of priorities.
- Load Balancing (LJF).
- Scalability.
- False sharing (+ hardware optimization).

Problems measuring parallel speedup



Reasons:

1. Background load presence.
2. Random nature of the algorithm.

According to materials of Prof. A.V. Boukhanovsky

Measurement of parallel program execution time

Universal tools:

- **Unix utility time** (measures the total execution of a program without possibility of any detail).
- **C-function clock** (does not take into account idle time).
- **C-function ctime** (not reenterable, see not cross-platform alternative to ctime_r).
- **C-function gettimeofday**, **clock_gettime**.

Specialized tools:

- **Library function omp_get_wtime** (maybe the function omp_get_wtick will need to be used).

Measurement of parallel program execution time (2)

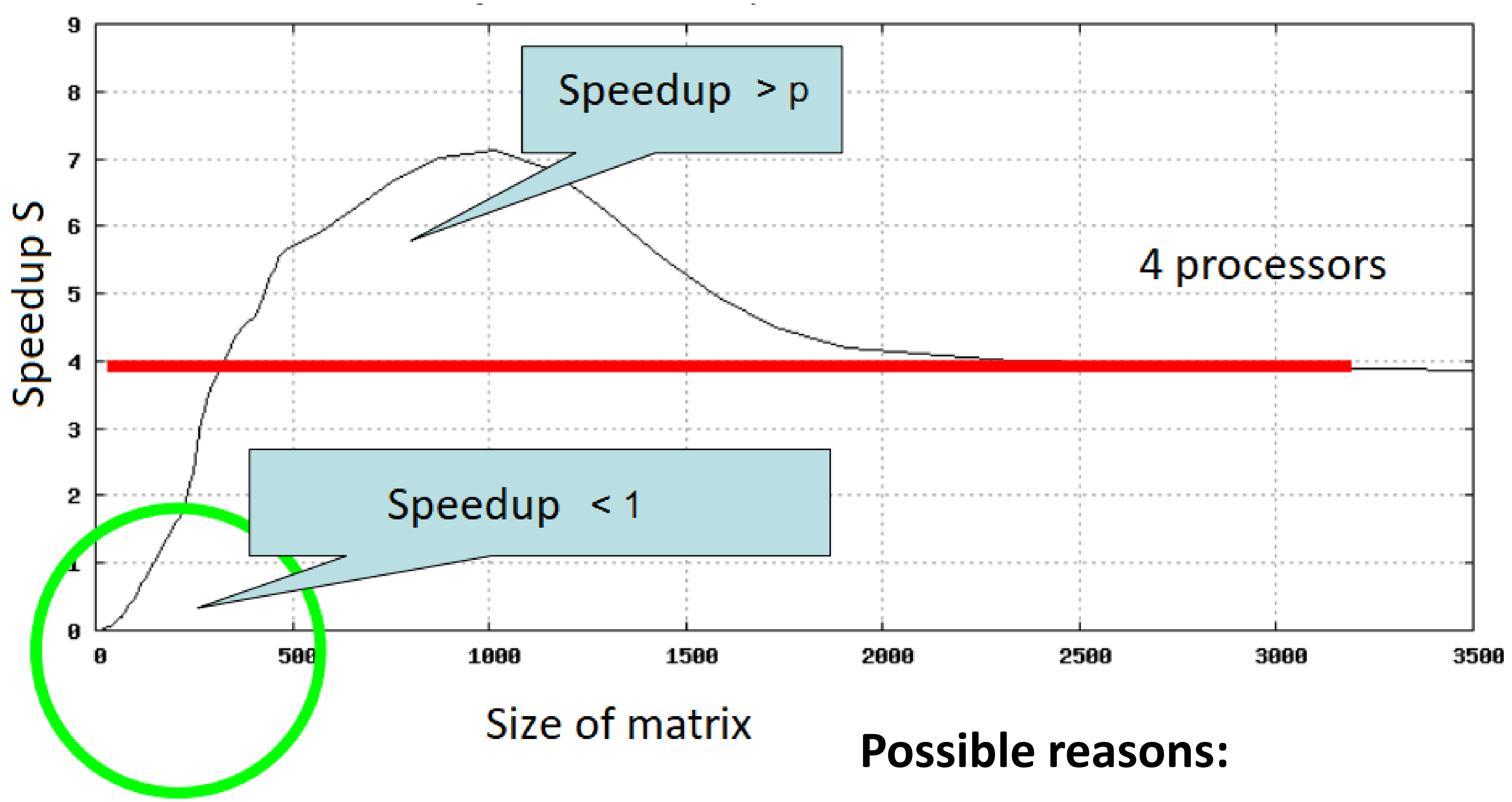
Solutions:

- Use thread-safe reentrant functions (for example **rand_r** instead of **rand**).
- Measure the time several times and then:

  1) take minimum sampling/measurement OR

  2) calculate the confidence interval.

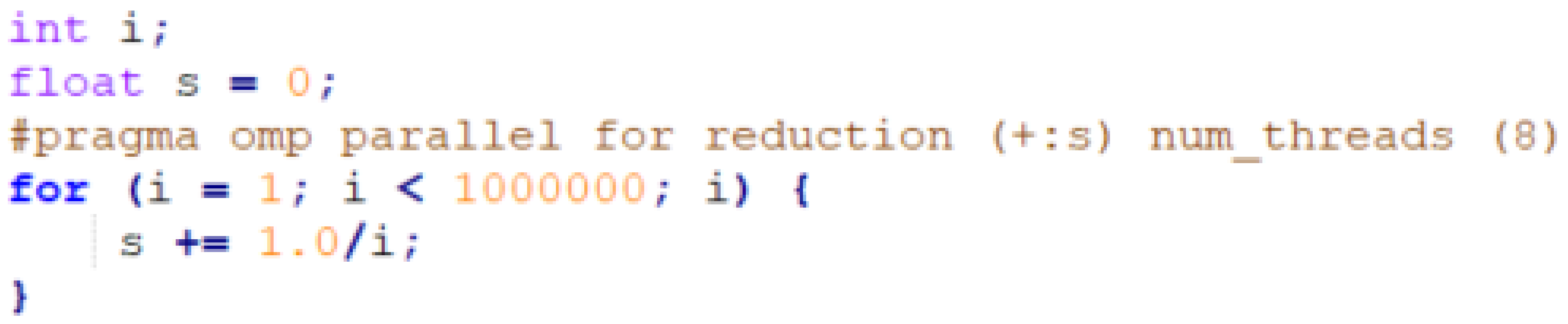
Subunit and super linear speedups



Possible reasons:

1. Overhead costs (i.e. 1000 division instructions).
2. Caching effects.
3. Paralleling algorithm error.

Changing the results of floating point calculations



```
int i;
float s = 0;
#pragma omp parallel for reduction (+:s) num_threads (8)
for (i = 1; i < 1000000; i) {
    s += 1.0/i;
}
```

1 thread:	s = 14.357357
8 threads:	s = 14.393189
Diff:	**0.25%**