

# Chapter 2



## Performance Measures: Part I

# Time Measurement and Operation Counts



## The Single Processor Case

### Definition

In general we call the time elapsed between issuing a command and receiving its results the **runtime**, or **execution time** of the corresponding process. Some authors also call it **elapsed time**, or **wall clock time**.

# Time Measurement and Operation Counts



## The Single Processor Case

### Definition

In general we call the time elapsed between issuing a command and receiving its results the **runtime**, or **execution time** of the corresponding process. Some authors also call it **elapsed time**, or **wall clock time**.

In the purely sequential case it is closely related to the so called **CPU time** of the process.

# Time Measurement and Operation Counts



## The Single Processor Case

### Definition

In general we call the time elapsed between issuing a command and receiving its results the **runtime**, or **execution time** of the corresponding process. Some authors also call it **elapsed time**, or **wall clock time**.

In the purely sequential case it is closely related to the so called **CPU time** of the process. There the main contributions are:

- **user CPU time:** Time spent in execution of instructions of the process.

# Time Measurement and Operation Counts



## The Single Processor Case

### Definition

In general we call the time elapsed between issuing a command and receiving its results the **runtime**, or **execution time** of the corresponding process. Some authors also call it **elapsed time**, or **wall clock time**.

In the purely sequential case it is closely related to the so called **CPU time** of the process. There the main contributions are:

- **user CPU time:** Time spent in execution of instructions of the process.
- **system CPU time:** Time spent in execution of operating system routines called by the process.

# Time Measurement and Operation Counts



## The Single Processor Case

### Definition

In general we call the time elapsed between issuing a command and receiving its results the **runtime**, or **execution time** of the corresponding process. Some authors also call it **elapsed time**, or **wall clock time**.

In the purely sequential case it is closely related to the so called **CPU time** of the process. There the main contributions are:

- **user CPU time:** Time spent in execution of instructions of the process.
- **system CPU time:** Time spent in execution of operating system routines called by the process.
- **waiting time:** Time spent waiting for time slices, completion of I/O, memory fetches. . .

# Time Measurement and Operation Counts



## The Single Processor Case

### Definition

In general we call the time elapsed between issuing a command and receiving its results the **runtime**, or **execution time** of the corresponding process. Some authors also call it **elapsed time**, or **wall clock time**.

In the purely sequential case it is closely related to the so called **CPU time** of the process. There the main contributions are:

- **user CPU time:** Time spent in execution of instructions of the process.
- **system CPU time:** Time spent in execution of operating system routines called by the process.
- **waiting time:** Time spent waiting for time slices, completion of I/O, memory fetches. . .

That means the time we have to wait for a response of the program includes the waiting times besides the CPU time.

# Time Measurement and Operation Counts



## Instructions: Timings and Counts

### clock rate and cycle time

The **clock rate** of a processor tells us how often it can switch instructions per second. Closely related is the **(clock) cycle time**, i.e., the time elapsed between two subsequent clock ticks.



# Time Measurement and Operation Counts



## Instructions: Timings and Counts

### clock rate and cycle time

The **clock rate** of a processor tells us how often it can switch instructions per second. Closely related is the **(clock) cycle time**, i.e., the time elapsed between two subsequent clock ticks.

### Example

A CPU with a clock rate of  $3.5 \text{ GHz} = 3.5 \cdot 10^9 \text{ 1/s}$  executes  $3.5 \cdot 10^9$  clock ticks per second. The length of a clock cycle thus is

$$1/(3.5 \cdot 10^9) \text{ s} = 1/3.5 \cdot 10^{-9} \text{ s} \approx 0.29 \text{ ns}$$

# Time Measurement and Operation Counts



## Instructions: Timings and Counts

Different instructions require different times to get executed. This is represented by the so called **cycles per instruction** (CPI) of the corresponding instruction. An average CPI is connected to a process A via  $CPI(A)$ .

# Time Measurement and Operation Counts



## Instructions: Timings and Counts

Different instructions require different times to get executed. This is represented by the so called **cycles per instruction** (CPI) of the corresponding instruction. An average CPI is connected to a process  $A$  via  $CPI(A)$ .

This number determines the total user CPU time together with the number of instructions and cycle time via

$$T_{U\_CPU}(A) = n_{instr}(A) \cdot CPI(A) \cdot t_{cycle}$$

# Time Measurement and Operation Counts



## Instructions: Timings and Counts

Different instructions require different times to get executed. This is represented by the so called **cycles per instruction** (CPI) of the corresponding instruction. An average CPI is connected to a process  $A$  via  $CPI(A)$ .

This number determines the total user CPU time together with the number of instructions and cycle time via

$$T_{U\_CPU}(A) = n_{instr}(A) \cdot CPI(A) \cdot t_{cycle}$$

Clever choices of the instructions can influence the values of  $n_{instr}(A)$  and  $CPI(A)$ .  $\rightsquigarrow$  compiler optimization.

# Time Measurement and Operation Counts



## MIPS versus FLOPS

A common performance measure of CPU manufacturers is the **Million instructions per second (MIPS)** rate.

It can be expressed as

$$MIPS(A) = \frac{n_{instr}(A)}{T_{U\_CPU}(A) \cdot 10^6} = \frac{r_{cycle}}{CPI(A) \cdot 10^6},$$

where  $r_{cycle}$  is the cycle rate of the CPU.

# Time Measurement and Operation Counts



## MIPS versus FLOPS

A common performance measure of CPU manufacturers is the **Million instructions per second (MIPS)** rate.

It can be expressed as

$$MIPS(A) = \frac{n_{instr}(A)}{T_{U\_CPU}(A) \cdot 10^6} = \frac{r_{cycle}}{CPI(A) \cdot 10^6},$$

where  $r_{cycle}$  is the cycle rate of the CPU.

This measure can be misleading in high performance computing, since higher instruction throughput does not necessarily mean shorter execution time.

# Time Measurement and Operation Counts



## MIPS versus FLOPS

More common for the comparison in Scientific computing is the rate of floating point operations (FLOPS) executed. The MFLOPS rate of a program  $A$  can be expressed as

$$MFLOPS(A) = \frac{n_{FLOPS}(A)}{T_{U\_CPU}(A) \cdot 10^6} [1/s],$$

with  $n_{FLOPS}(A)$  the total number of FLOPS issued by the program  $A$ .

# Time Measurement and Operation Counts



## MIPS versus FLOPS

More common for the comparison in Scientific computing is the rate of floating point operations (FLOPS) executed. The MFLOPS rate of a program  $A$  can be expressed as

$$MFLOPS(A) = \frac{n_{FLOPS}(A)}{T_{U\_CPU}(A) \cdot 10^6} [1/s],$$

with  $n_{FLOPS}(A)$  the total number of FLOPS issued by the program  $A$ .

Note that not all FLOPS (see also Chapter 3 summer term) take the same time to execute. Usually divisions and square roots are much slower. The MFLOPS rate, however, does not take this into account.



# Time Measurement and Operation Counts



## CPU-Time versus Execution Time

### Example

Input:

```
ct0=0;  
A=randn(1500);
```

```
tic  
ct0=cputime;  
pause(2)  
toc  
cputime-ct0
```

```
tic  
ct0=cputime;  
[Q,R]=qr(A);  
toc  
cputime-ct0
```

# Time Measurement and Operation Counts



## CPU\_Time versus Execution Time

### Example

Input:

```
ct0=0;  
A=randn(1500);
```

```
tic  
ct0=cputime;  
pause(2)  
toc  
cputime-ct0
```

```
tic  
ct0=cputime;  
[Q,R]=qr(A);  
toc  
cputime-ct0
```

Output:

```
Elapsed time is 2.000208 seconds.
```

```
ans =  
  
    0.0300
```

```
Elapsed time is 0.733860 seconds.
```

```
ans =  
  
    21.6800
```

Executed on a 4x8core Xeon<sup>®</sup> system.

# Time Measurement and Operation Counts



## CPU\_Time versus Execution Time

Obviously in a parallel environment the CPU time can be much higher than the actual execution time elapsed between start and end of the process.

In any case it can be much smaller, as well.

# Time Measurement and Operation Counts



## CPU\_Time versus Execution Time

Obviously in a parallel environment the CPU time can be much higher than the actual execution time elapsed between start and end of the process.

In any case it can be much smaller, as well.

The first result is easily explained by the splitting of the execution time into user/system CPU time and waiting time. The process is mainly waiting for the `sleep` system call to return whilst basically accumulating no active CPU time.

# Time Measurement and Operation Counts



## CPU\_Time versus Execution Time

Obviously in a parallel environment the CPU time can be much higher than the actual execution time elapsed between start and end of the process.

In any case it can be much smaller, as well.

The first result is easily explained by the splitting of the execution time into user/system CPU time and waiting time. The process is mainly waiting for the `sleep` system call to return whilst basically accumulating no active CPU time.

The second result is due to the fact that the activity is distributed to several cores. Each activity accumulates its own CPU time and these are summed up to the total CPU time of the process.

# Parallel Cost and Optimality



## Definition (Parallel cost and cost-optimality)

The cost of a parallel program with data size  $n$  is defined as

$$C_p(n) = p * T_p(n).$$

Here  $T_p(n)$  is the **parallel runtime** of the process its execution time on  $p$  processors.

The parallel program is called **cost-optimal** if

$$C_p = T^*(n).$$

$T^*(n)$  represents the execution time of the fastest sequential program solving the same problem.

In practice  $T^*(n)$  is often approximated by  $T_1(n)$ .

# Speedup



The **speedup** of parallel program

$$S_p(n) = \frac{T^*(n)}{T_p(n)},$$

is a measure for the acceleration in terms of execution time we can expect from a parallel program.

The speedup is strictly limited from above by  $p$  Since otherwise the parallel program would motivate a faster sequential algorithm. See [RAUBER/RÜNGER '10] for details.

In practice often the speedup is computed with respect to the sequential version of the code, i.e.,

$$S_p(n) \approx \frac{T_1(n)}{T_p(n)}.$$

# Parallel Efficiency



Usually the parallel execution of the work a program has to perform comes at the cost of certain management of subtasks. Their distribution, organization and interdependence leads to a fraction of the total execution, that has to be done extra.

## Definition

The fraction of work that has to be performed by a sequential algorithm as well is described by the **parallel efficiency** of a program. It is described by

$$E_p(n) = \frac{T^*(n)}{C_p(n)} = \frac{S_p(n)}{p} = \frac{T^*}{p \cdot T_p(n)}$$

The parallel efficiency obviously is limited from above by  $E_p(n) = 1$  representing the perfect speedup of  $p$ .



# Amdahl's Law



In many situations it is impossible to parallelize the entire program. Certain fractions remain that need to be performed sequentially. When a (constant) fraction  $f$  of the program needs to be executed sequentially Amdahl's law describes the attainable speedup.

# Amdahl's Law



In many situations it is impossible to parallelize the entire program. Certain fractions remain that need to be performed sequentially. When a (constant) fraction  $f$  of the program needs to be executed sequentially Amdahl's law describes the attainable speedup.

The total parallel runtime  $T_p(n)$  then consists of

- $f \cdot T^*(n)$  the time for the sequential fraction and
- $(1 - f)/p \cdot T^*(n)$  the time for the fully parallel part.

# Amdahl's Law



In many situations it is impossible to parallelize the entire program. Certain fractions remain that need to be performed sequentially. When a (constant) fraction  $f$  of the program needs to be executed sequentially Amdahl's law describes the attainable speedup.

The total parallel runtime  $T_p(n)$  then consists of

- $f \cdot T^*(n)$  the time for the sequential fraction and
- $(1 - f)/p \cdot T^*(n)$  the time for the fully parallel part.

The best attainable speedup can thus be expressed as

$$S_p(n) = \frac{T^*(n)}{f \cdot T^*(n) + \frac{1-f}{p} T^*(n)} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}.$$

# Scalability of Parallel Programs



## Question

Is the parallel efficiency of a parallel program independent of the number of processors  $p$  used?

The question is answered by the concept of [parallel scalability](#). Scientific computing and HPC distinguish two forms of scalability:

# Scalability of Parallel Programs



## Question

Is the parallel efficiency of a parallel program independent of the number of processors  $p$  used?

The question is answered by the concept of [parallel scalability](#). Scientific computing and HPC distinguish two forms of scalability:

- **strong scalability**  
captures the dependence of the parallel runtime on the number of processors for a fixed total problem size.

# Scalability of Parallel Programs



## Question

Is the parallel efficiency of a parallel program independent of the number of processors  $p$  used?

The question is answered by the concept of **parallel scalability**. Scientific computing and HPC distinguish two forms of scalability:

- **strong scalability**  
captures the dependence of the parallel runtime on the number of processors for a fixed total problem size.
- **weak scalability**  
captures the dependence of the parallel runtime on the number of processors for a fixed problem size per processor.