Algoritmi Paraleli si Distribuiti
an 3 C+TI
Parallel and Distributed Algorithms
3 C-engl

Conf.dr.ing. Ioana Şora

http://staff.cs.upt.ro/~ioana

ioana.sora@cs.upt.ro

1. General Overview and Scope of this course

2. Intro to Parallel Algorithms

General Overview and Scope of this course

Terminology:

Parallel, Distributed, Concurrent.

Shared-memory, Message-Passing.

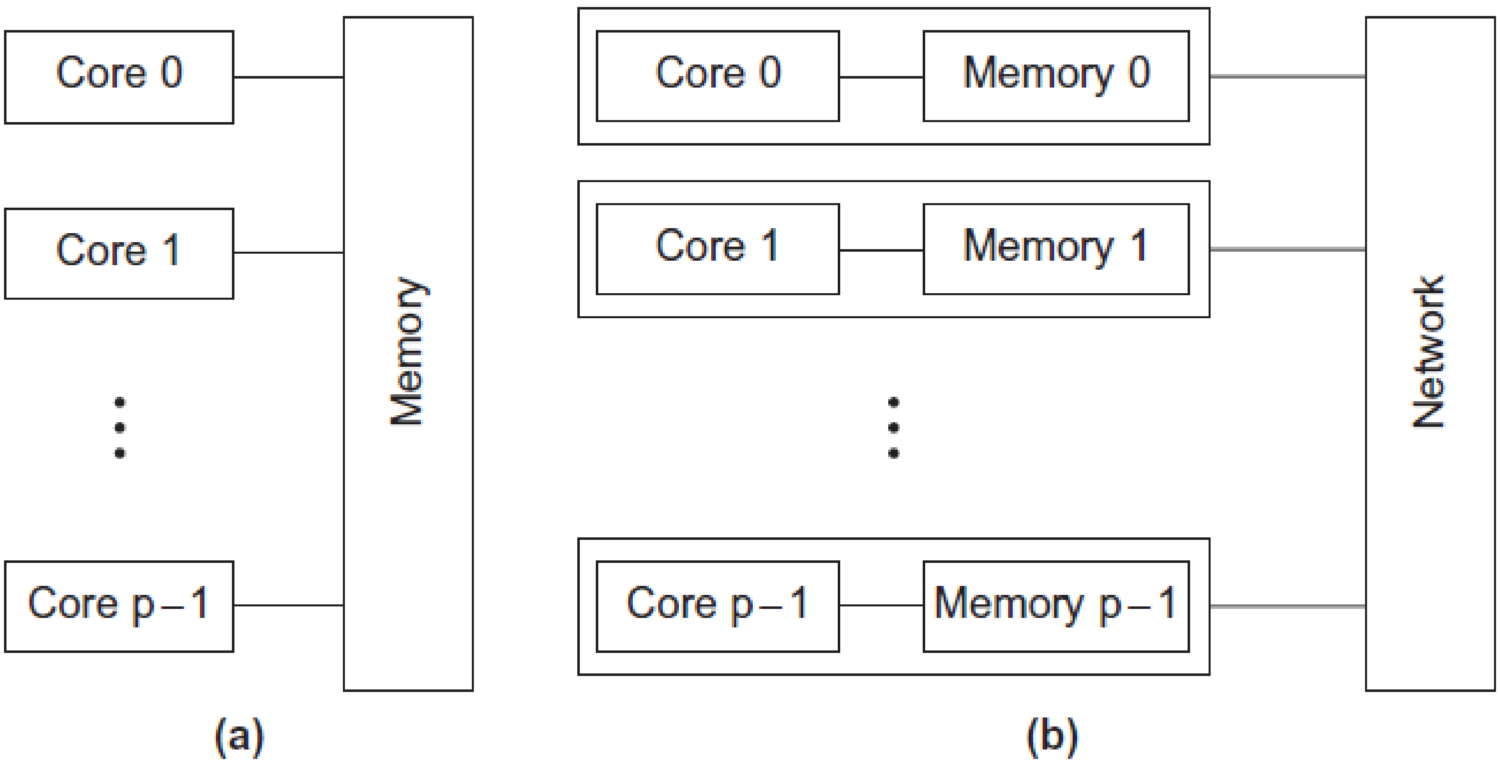
Terminology: Concurrent, Parallel, Distributed

- Concurrent computing: a system is said to be concurrent if it can support two or more actions (tasks) *in progress at the same time*.
  - Executing multiple tasks at the same time by:
    - **Timesharing**: the tasks actually share timeslices of the same processing element (a processor, single core). The execution "at the same time" is an illusion.
    - **True parallelism**: the tasks are executed on different processing elements (processors, cores). The execution of the different tasks really happens at the same time.
      - Needs a **parallel system:** a system with *multiple processing elements* (processors, cores)
      - Takes the forms of **Parallel computing** or **Distributed computing**

Type of parallel systems

- Shared-memory
  - The processing elements (processors, cores) can share access to the computer's memory
  - Coordinate the processing elements by having them examine and update shared memory locations
- Distributed-memory (Message Passing)
  - Each processing element has its own, private memory
  - The processing elements can communicate only by explicit **message passing** across a network

Type of parallel systems



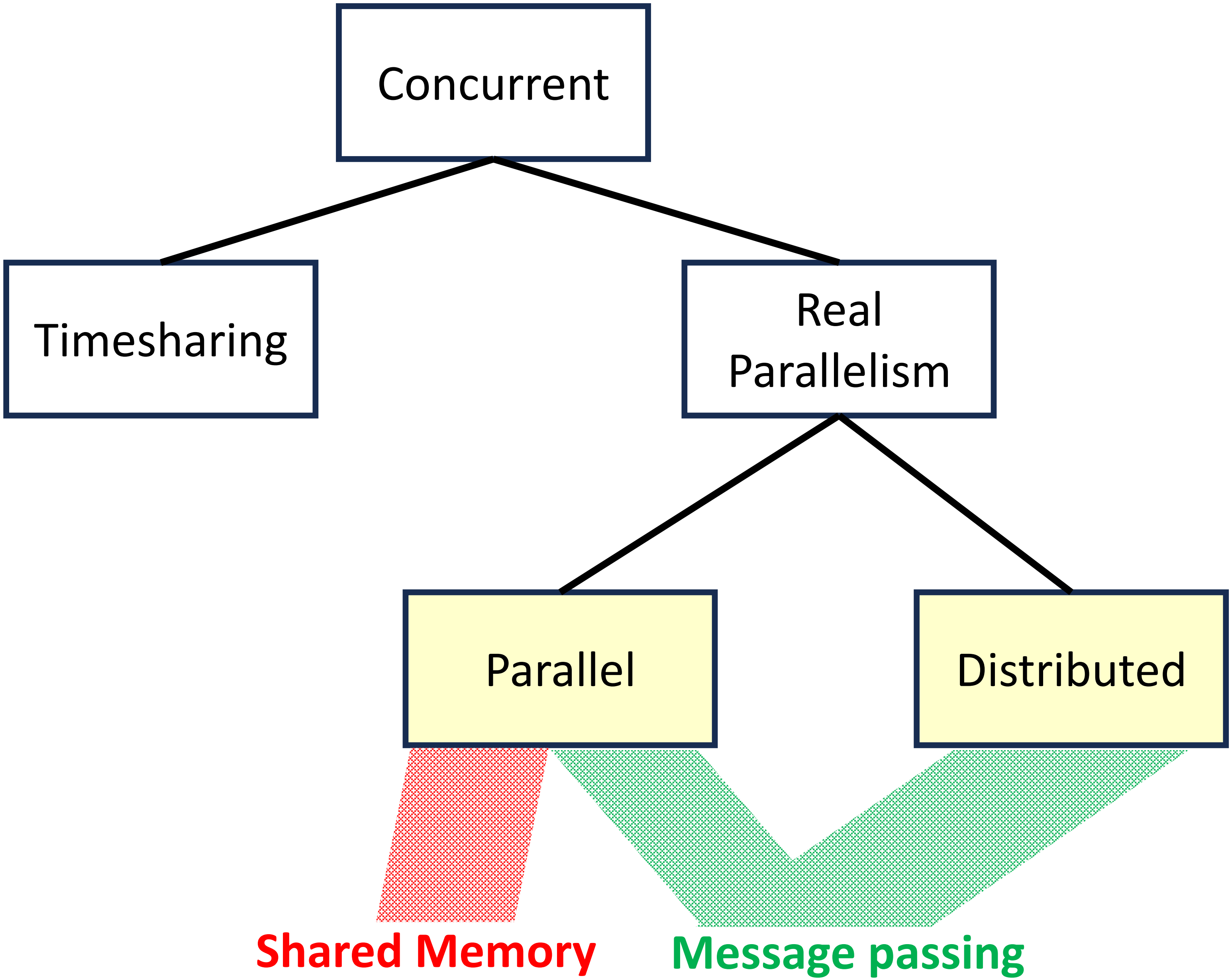
Shared-memory

Distributed-memory (Message passing)

Parallel vs Distributed

- **Parallel Systems:** can use a system with shared memory or a system based on message passing
- ***Parallel Algorithms*: have as goal to SPEED UP solving a problem**

- **Distributed Systems**: a set of independent processors connected by a network (message passing)
- ***Distributed Algorithms*: manage shared resources and coordinate participants**


Concurrent
Timesharing
Real Parallelism
Parallel
Distributed
Shared Memory
Message passing

Scope of this Class

Parallel Algorithms
Parallel Programming

Distributed Algorithms
Distributed Programming

Systems with
Shared Memory

POSIX threads. OpenMP. Java Threads.

Systems with
Distributed Memory
(with Message Passing)

MPI (Message Passing Interface). Sockets.

Example: Parallel with Shared Memory

- A very large array of numbers is in the memory of a multicore computer NrCores=4
- We want to speed up computing the sum of these numbers
- Each core runs a task that computes a partial sum for ¼ of the array, accessing directly the array
- The 4 partial sums are added at the end

Example: Parallel with Message Passing

- A huge array of numbers is in the memory of a computer that is connected in a network with N other workstations
- We want to compute the sum of the square roots of these numbers
- We split the array in N subarrays and send a subarray to each workstation of the network
- Each workstation computes the sum and sends its result back
- The partial sums are added at the end
- Sending data across the network is a big overhead. It depends on the complexity and amount of computation if it still pays off (we measure a speedup compared with the simple sequential solution)

Example: Distributed algorithms

- A number of nodes in a network can receive computing tasks. However, some nodes may suffer failures and become unavailable.
- The distributed system must detect which nodes are “alive” at a moment in order to decide task assignments
- Example of liveness/failure detection algorithm: Heartbeat based detection

Intro to Parallel Algorithms

Parallel Program Design

Parallel Performance Metrics

Bibliography

- [Pacheco]: Peter Pacheco, Matthew Malensek, Introduction to Parallel Programming, 2nd Edition, Morgan Kaufmann Publisher, March 2020, Chapter 1.4, 2.6, 2.7

The need for parallel programming

- Most existing programs have been written for conventional single-core systems
- Multicore systems are now everywhere => We need parallel versions of programs
- Approaches to parallelization:
  - Redesign and rewrite serial programs so that they are explicitly *parallel*
  - Use *automatic parallelization*: use translation programs that will convert serial programs into parallel programs
    - Parallelizing compilers, or compiler optimization for parallelism: they exist and work but have limited applicability and performances
    - In many cases the best parallelization may be obtained by designing an entirely new algorithm

Example: Automatic parallelization success and limitations

```
int a,b,c,x,y,z;
....
x=y; //Instruction1
a=b; //Instruction2
z=x; //Instruction3
c=a; //Instruction4
```

- Automatic parallelization (parallelizing compilers) must perform a ***Dataflow analysis*** first
- Data dependencies prevent that Instr1 and Instr3 are done in parallel.
- Also data dependencies prevent that Instr2 and Instr4 are done in parallel.
- Possible parallelization solution:
  - Thread1: Instr1; Instr3
  - Thread2: Instr2;Instr4
- Looking for ***simple statements*** that can be executed in parallel *is not really "worth the trouble"*
- *Automatic parallelization focuses mainly on finding **loops** that can be parallelized*

Example: Automatic parallelization success and limitations

Parallelizing a loop= executing its iterations in parallel

```
int a[N], b[N], c[N];
...
for (int i=0; i<N; i++) {
    c[i] = a[i] + b[i];
}
```

Can be easily parallelized because Iterations are independent (no iteration depends on previous iterations. Iterations can be executed in any order)

```
int a[N];
...
int sum = 0;
for (int i=0; i<N; i++) {
    sum = sum + a[i];
}
```

Dataflow analysis shows that every iteration depends on the previous one due to the use of variable sum

Steps in parallel program design

- In many cases the best parallelization may be obtained by “manual parallelization” = designing an entirely new algorithm
- Way to go:
  1. Divide (partition) the work into *tasks* that *could be* performed in the same time
  2. Coordinate the tasks so that the software correctly and efficiently is doing its requirements

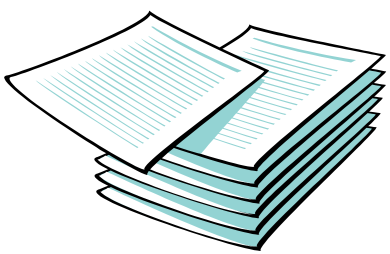
Approaches for work partitioning

- Task parallelism
  - Partition various tasks carried out solving the problem among the processing elements
- Data parallelism
  - Partition the data used in solving the problem among the processing elements
  - Each processing element carries out similar operations on it's part of the data

Intuitive Example: Data parallelism vs Task parallelism

300 exam papers

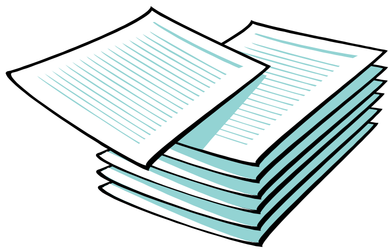
15 questions each



- Professor and teaching assistants have to grade a grand total of 300 exam papers. Each exam paper answers 15 questions/exercises.
- How can they parallelize the grading process?

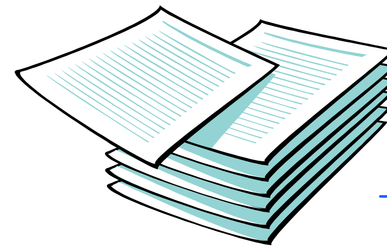
Division of work – data parallelism

TA#1



100 exams

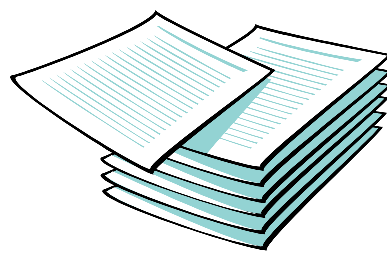
Question 1-15



TA#3

100 exams

Question 1-15



TA#2

100 exams

Question 1-15

Division of work – task parallelism

TA#1

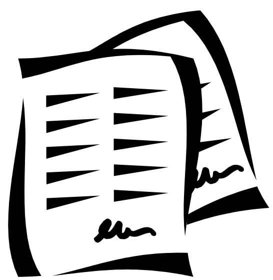
300 exams

Questions 1 - 5

TA#3

300 exams

Questions 11 - 15



TA#2

300 exams

Questions 6 - 10

Coordination

- Processing elements usually need to coordinate their work
- Communication – one or more processing element send their current partial sums to another processing element
- Load balancing – share the work evenly among the processing elements so that one is not heavily loaded
- Synchronization – because each processing element works at its own pace, sometimes need to make sure one does not get too far ahead of the rest

Example: Sum of n numbers

Decomposition (Division of work):

- Suppose we have *p* processing elements and $p \leq n$
- Each processing element will work on a subarray of *n/p* elements and compute the partial sum.
  - This part is example of **data parallelism**: all processing elements execute the same code (the same task) on different data
- When the processing elements are done computing their values of partial sums, they send their results to a designated "master" processing element, which can add their partial results.
  - This part can be considered an example of **task-parallelism**. There are *two* types of tasks: one executed by the master, the other by everybody else.

Example: Sum of n numbers

Coordination:

- **communication**: processing elements send their partial sums to another processing element.
- **load balancing**: we want the amount of time taken by each processing element to be roughly the same. If the processing elements are identical, we assign them the same number of elements

Parallel Performance Metrics

- Speedup
- Efficiency
- Amdahl’s law
- Scalability

Speedup of a parallel program

- Number of processing elements = p
- We assume that all processing elements are identical
- Serial run-time = T_{serial} is the time elapsed between the beginning and the end of its execution on a sequential computer
- Parallel run-time = $T_{parallel}$ is the time elapsed from the moment a parallel computation starts to the moment the last processing element finishes execution.
- T_{serial} $T_{parallel}$ are measured as ***wall-clock-times (NOT CPU-times)***

$$S = \frac{T_{serial}}{T_{parallel}} \leq p$$

Wall-clock time vs CPU time

- **Wall clock time**: the **total time elapsed** during the measurement.
  - the time you can measure with a stopwatch, assuming that you are able to start and stop it exactly at the execution points you want
- **CPU Time**: the time **the CPU was busy processing the program's instructions**.
  - The time spent waiting for other things to complete (like I/O operations) or while the process is idle (sleep) or waiting for resources to become available(mutex locks) is not included in the CPU time!

Wall-clock time vs CPU time

```
#include <time.h>
int clock_gettime(clockid_t clk_id, struct timespec *tp);
```

`clock_gettime()` retrieves the time of the specified clock `clk_id`.

The `tp` argument is a `timespec` struct:

```
struct timespec {
        time_t   tv_sec;        /* seconds */
        long     tv_nsec;       /* nanoseconds */
};
```

- The `clk_id` argument:
  - `CLOCK_MONOTONIC`: Clock that cannot be set and represents monotonic time since some unspecified starting point -> **Wall clock time**
  - `CLOCK_PROCESS_CPUTIME_ID`: Per-process timer from the CPU -> **CPU time per process**

Measuring wall clock time

```
struct timespec start, finish;
double elapsed;

printf("\nMeasuring Wall-clock time \n");
printf("Start ...\n");
clock_gettime(CLOCK_MONOTONIC, &start); //

//….code that is measured

clock_gettime(CLOCK_MONOTONIC, &finish);

elapsed = (finish.tv_sec - start.tv_sec);
elapsed += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;

printf("Wall-clock time =%lf \n", elapsed);
```

Thinking Question

- For the same sequence of code, we measure:
- **TCPU** (using clock_gettime() with **`CLOCK_PROCESS_CPUTIME_ID`**)
- **Twall** (using clock_gettime() with **`CLOCK_MONOTONIC`**)
- Which are the possible relationships between TCPU and Twall? When do they happen (give examples)?
  - TCPU=Twall ?
  - TCPU<Twall ?
  - TCPU>Twall ?

Other functions

- Function **`clock()`** – Do NOT use !
  - On **Unix** systems: **`clock()`** measures the CPU-time (see Linux man page)
  - On **Windows** systems: **`clock()`** measures the wall-clock time (see Microsoft online reference).
    - On Windows systems, clock() is measuring what we need, BUT it is a custom solution, not a portable solution

Parallel overhead

$$T_{parallel} = T_{serial} / p + T_{overhead}$$

- Sources of overhead:
  - Interaction and communication between processing elements
  - Idling when some processing elements can not work (waiting for synchronization with others, or there is not enough work due to intrinsic serial part of algorithm)
  - Excess computation when parallel algorithm is different than the serial algorithm

Serial fraction

- Suppose that the parallelization is "perfect," without involving any overhead. In this case, speedup S=p.
- In this case, if p $\rightarrow\infty$, will S$\rightarrow\infty$?
  - NO: Every algorithm has a serial fraction: a fraction *r* of any program is inherently sequential and cannot be parallelized

$$T_{parallel} = (1-r) \times T_{serial} / p + r \times T_{serial}$$

$$S = \frac{T_{serial}}{(1-r) \times T_{serial} / p + r \times T_{serial}}$$

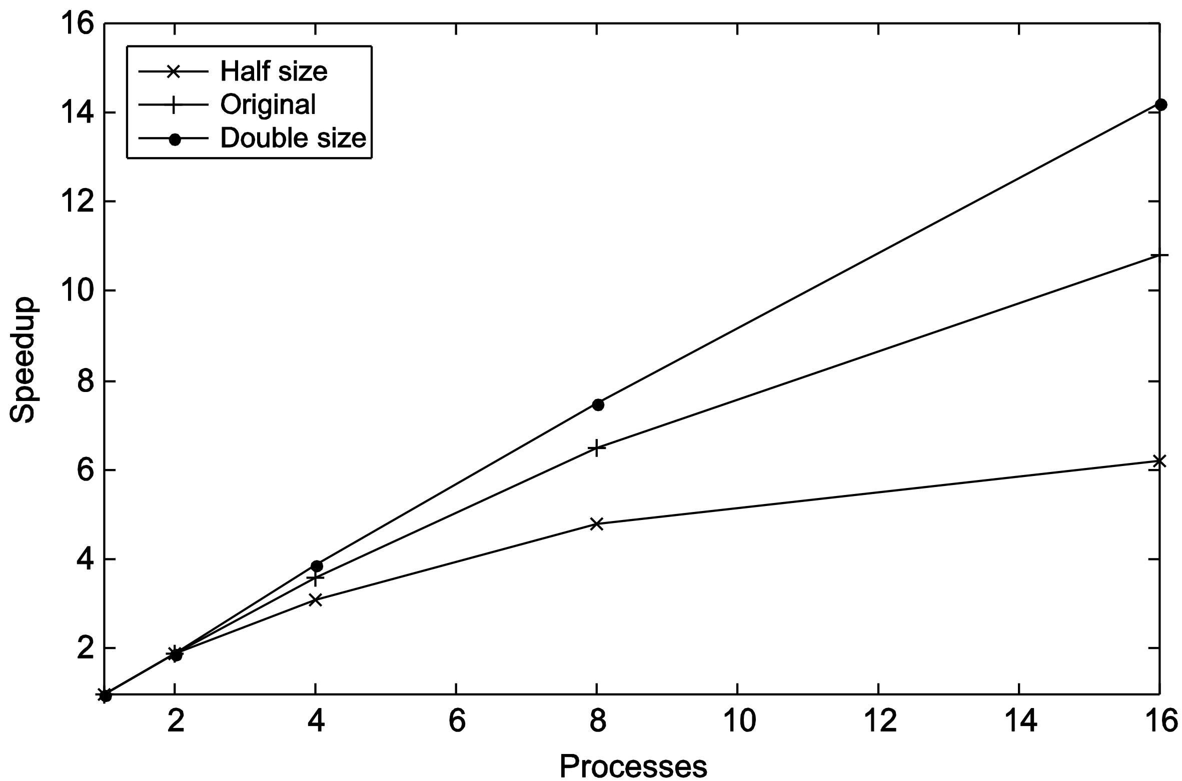
If p $\rightarrow\infty$, then S$\rightarrow$ 1/r

Amdahl’s Law

- Amdahl established (1967) how slower parts (not parallelizable parts) of an algorithm influence its overall performance:
- ***The fraction r of inherently sequential or unparallelizable computation limits the speed-up S that can be achieved with any number of p processors to the value S<1/r***
- Amdahl's law assumes that for any given input, the parallel and serial implementations perform exactly the same number of computational steps
- Example: If r=5%, then S<1/0.05=20 no matter how many processing elements are used (even with p=1000 cores and perfect parallelization without overhead, S=20)

Speedup as function of p, on different problem sizes

- In practice, in many cases the serial fraction r decreases as a function of problem size. Therefore, the upper bound on the speed-up S usually increases as a function of problem size.



Efficiency of a parallel program

$$E = \frac{S}{p} = \frac{\left(\dfrac{T_{serial}}{T_{parallel}}\right)}{p} = \frac{T_{serial}}{p \cdot T_{parallel}} \leq 1$$

Scalability

- **Informal definition**: a solution is *scalable* if it can obtain speedups when it is run on larger systems
- **Formal definition**: a solution is *scalable* if there is a rate at which the problem size can be increased so that as the number of processing elements is increased, the efficiency remains constant.
- If we increase the number of processing elements and keep the efficiency constant without increasing problem size, the problem is *strongly scalable*.
- If we keep the efficiency constant by increasing the problem size at the same rate as we increase the number of processing elements, the problem is *weakly scalable*.

Scalability conditions

$$E = \frac{T_{serial}}{p \cdot T_{parallel}} = \frac{T_{serial}}{T_{serial} + T_{overhead}} = \frac{1}{1 + T_{overhead} / T_{serial}}$$

If n=ct and p is increased ↗:

T_{serial}=ct and $T_{overhead}$ increases. → E decreases ↘

If p=ct and n is increased ↗:

T_{serial} increases and $T_{overhead}$ increases.

if rate of growth for $T_{overhead}$ is smaller than for T_{serial}

→ E increases ↗

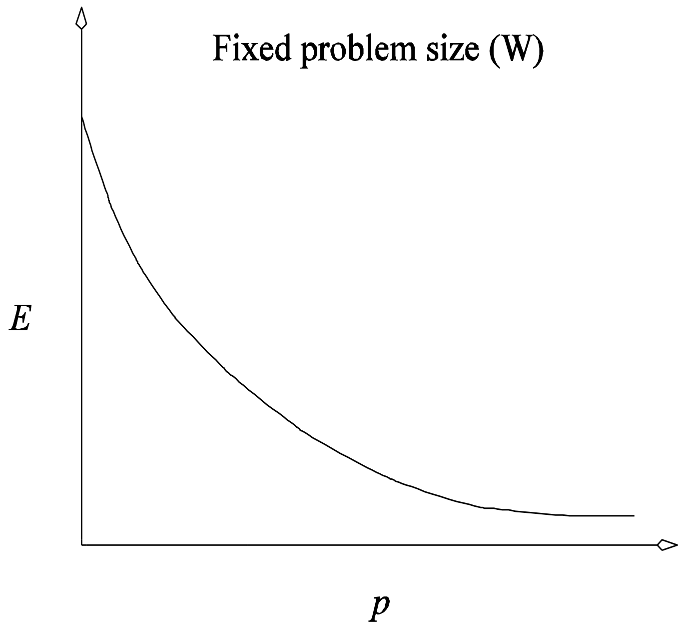
if rate of growth for $T_{overhead}$ is bigger than for T_{serial}

→ E decreases ↘

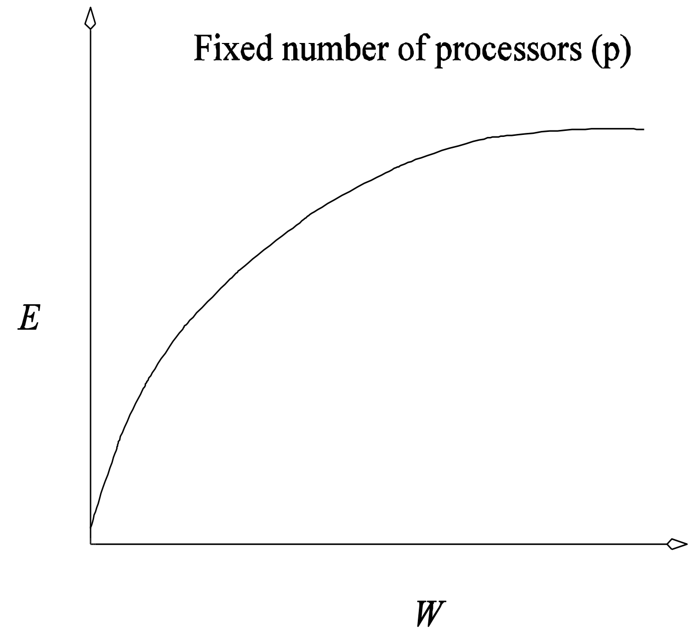
Ensuring scalability

- Actions that we can undertake in order to ensure scalability in practice: *we must ensure that the TOverhead has a growth rate smaller* than the growth rate of the serial time
  - Constant Efficiency and non-decreasing Speedup (the elements from the definition of scalability) result as a *consequence* of this!
  - if rate of growth for $T_{overhead}$ is smaller than rate of growth for T_{serial} then it is possible to increase simultaneoulsy n and p and keep E=ct

Efficiency and Scalability



(a)



(b)

- Variation of efficiency: (a): as the number of processing elements p is increased for a given problem size W
- The phenomenon illustrated in graph (a) is *common to **all** parallel systems*

- Variation of efficiency: (b): as the problem size W is increased for a given number of processing elements p.
- The phenomenon illustrated in graph (b) *is **not** common to all parallel systems – E increases **only for scalable systems***!

Conclusions

- Automatic parallelization works but with limitations
- In many cases the best parallelization may be obtained by designing an entirely new algorithm
- Steps for parallel algorithm design:
  - Decomposition(Division of work): Data parallelism or Task parallelism
  - Coordination: Communication and Synchronization
- Parallel performance metrics: Speedup, Efficiency, Scalability