Multithreading - part 1

Lecture 4 We are Legion, for we are many

Stanisław Gepner

14th April, 2023

Some general things to start with

Introduction

During next three meetings we will consider the bread and butter of HPC - the parallelisation.

Parallelization, parallel or concurrent execution or processing (or whatever the name) is based on the use of more than one processing unit (CPU) to perform a task at hand. For us, coming from the computational sciences background this means applying a number of computing cores to shorten the time needed to obtain results. Sometimes, when problems are very large, parallelization might be the only way to handle the computational problem. We note, that for programmers working on the development of operating systems, user interfaces or general purpose user applications, while general efficiency might still be important, parallel processing might have a different goal. It might be used to enable responsiveness of the system to the user when under load. But this remains beyond our simple, computing needs.

Why parallel?

We will start with stating that the true power of computing comes not from the processing power of a single CPU, but from the ability of processors working together in a joint effort. There are many reasons why it is more convenient to connect many processors to work together rather than to produce a single all powerful processor (or computer) to solve our problem.

Why parallel?

Let's start with an illustrative example, and show that some problems are simply to large to be handled on a single computer.

Now is the time to show the submarine example.

Why parallel?

The reason we wish to use some form of parallelism is to be able to perform tasks faster, or to be able to perform them at all. Some tasks are easy to parallelize, some less so.

Example:

- I/O operations
- Processing pixels
- others ...

Different types of parallelization

In the world of parallel processing concurrency of execution can be achived by various means. The most general clasification of parallelism is:

- Symmetric Multiprocessing - SMP
  here processors remain under the control of a single OS instance and share the main memory. (One computer - many cores).
- Massively Parallel Processing - MPP in the cluster sense meaning a large number of separate processors (computers), each with a separate OS, but connected via network.
- Hybrid approach uses the mix of the two.

Scaling - How to know if it is faster?

- Scaling and Speedup
- Weak vs Strong scaling
- Amdahl's law, Gustafson's law

Scaling

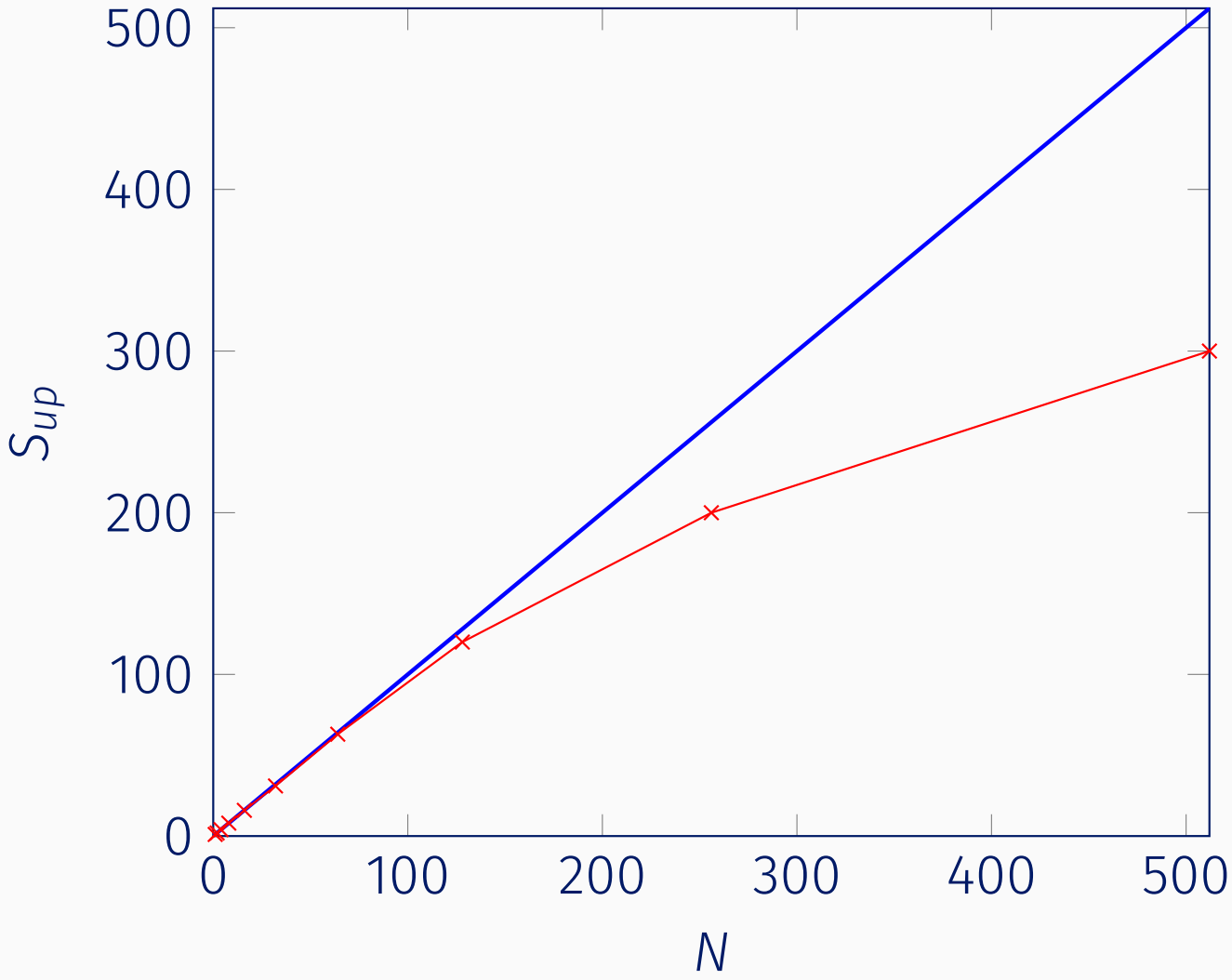
Scaling or scalability is ability of the system to handle more work, either as the size of the computer (number of processing units) or the problem being solved grows. For the computer system, scalability is important, and means proportional increase in computing power as resources are added. From the perspective of a program scalability means ratio between the actual and theoretical acceleration resulting from increased resource allocation. The often used measure of acceleration is **speedup**:

$$S_{up} = \frac{t_1}{t_N} \tag{1}$$

Sometimes the notion of parallel efficiency is used, defined as $n = \frac{S_{up}}{N}$

Scaling and Speedup

Mention how do we measure $S = \frac{t_1}{t_N}$



Weak and Strong Scaling

Strong scaling, corresponds to the situation where the number of processors is increased, while the size of the problem remains the same, leading to the reduced load per processor. This type of scaling is considered for applications that are of fixed size, and we wish to run them faster, yet utilising the available resources reasonably. As we split the problem into smaller portions and add more processors N, at one moment the communication overhead will dominate the computational time. In short $N \to \infty$ is not possible and at some level we need to use a more powerfull CPU to speed-up the solution.

Weak and Strong Scaling

Weak scaling, corresponds to the situation where both the number of processors and the work load are increased, such that the load per processor is constant. This type of scaling is considered for problems limited by the size (memory and not CPU-time), and usually restricted by avalible resources. In this case our goal is not to do calculations faste, but to perform larger problems in the same time and using larger computers. A problem that scales well weakly may effectively utilise increasing number of processors, so for the perfect weak scaling $N \to \infty$ is possible (our problem grows with N).

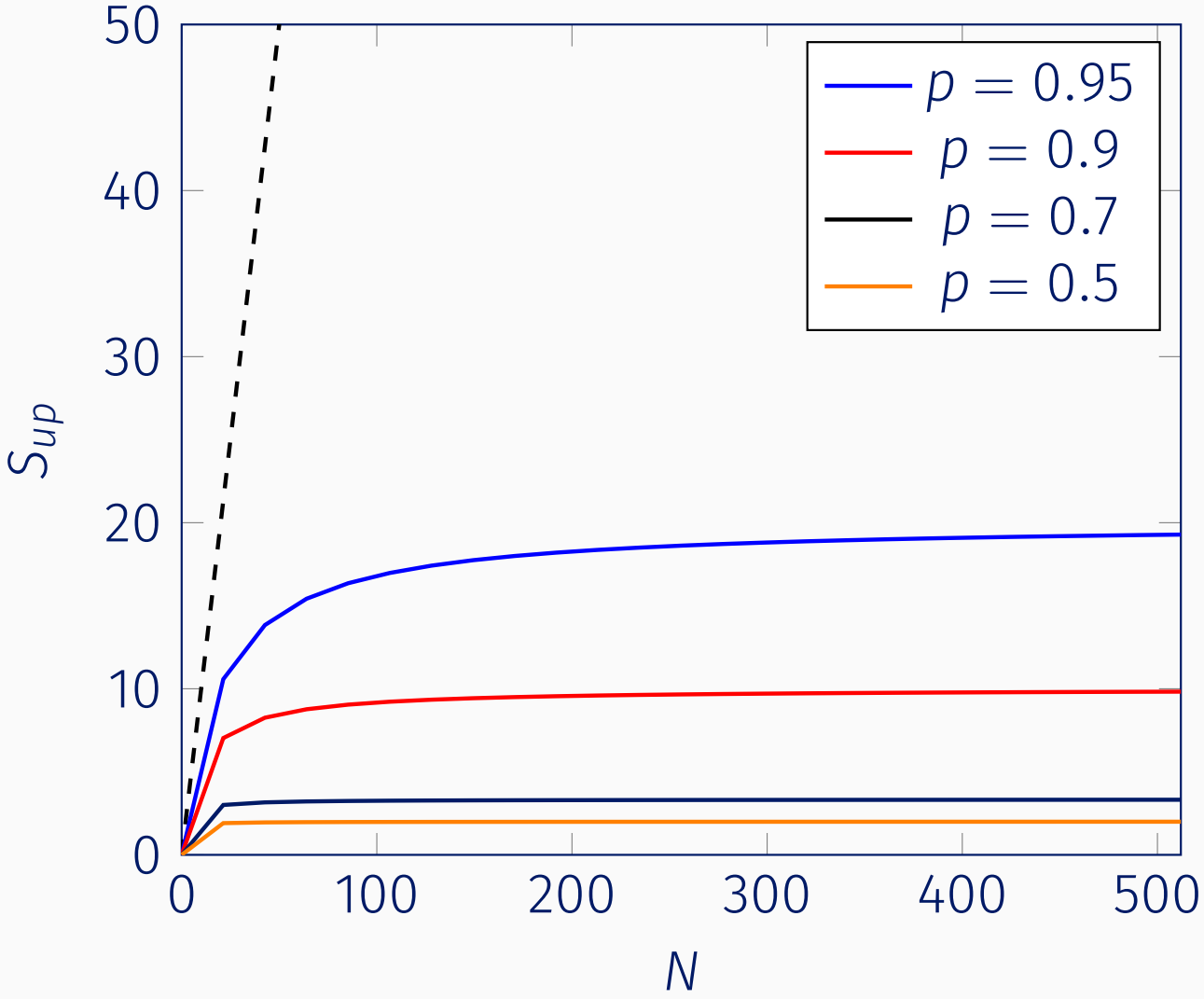
Amdahl's law and the Strong Scaling (Amdahl 1967)

Amdahl postulates that speedup of a parallel application is limited by the serial fraction of the program, that can not be parallelized. With p-the part spent to perform the part of the program that can be accelerated the remaining part is $1 - p$ (e.g. I/O operations), with s - the acceleration factor - assumed e.g. equal to the number of cores N:

$$S_{up} = \frac{1}{(1-p) + \frac{p}{s}} \tag{2}$$

Note: Strong scaling and Amdahl's law is a rather pessimistic concept.

Amdahl's law



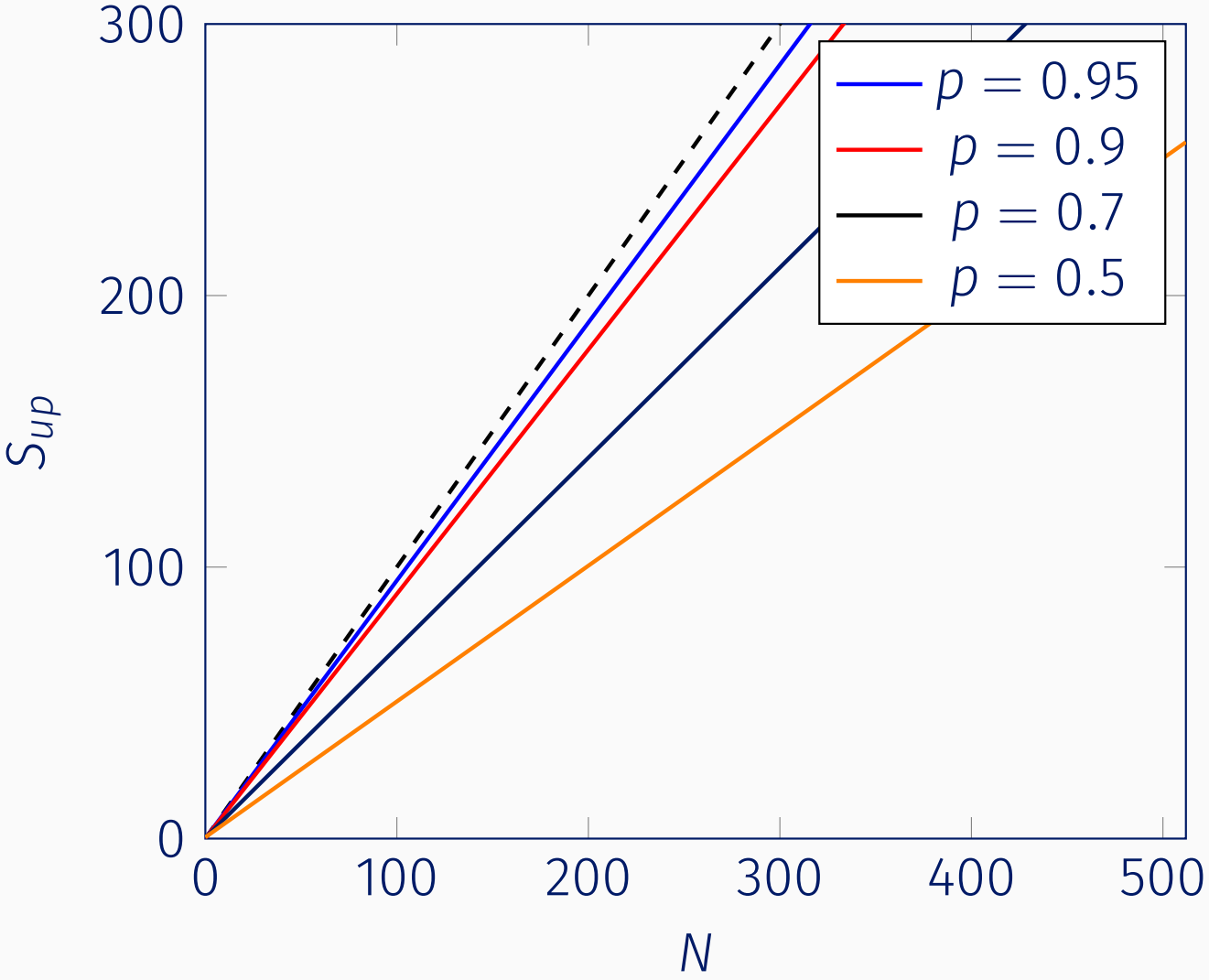
Gustafson's law (1988) and the Weak Scaling

Gustafson looks at a problem differently, and postulates, that as the available resources increase, so is the size of the problem one wants to solve. Consequently speedup S follows

$$S_{up} = (1 - p) + p \times N, \tag{3}$$

with p representing the part of the program that can be accelerated.

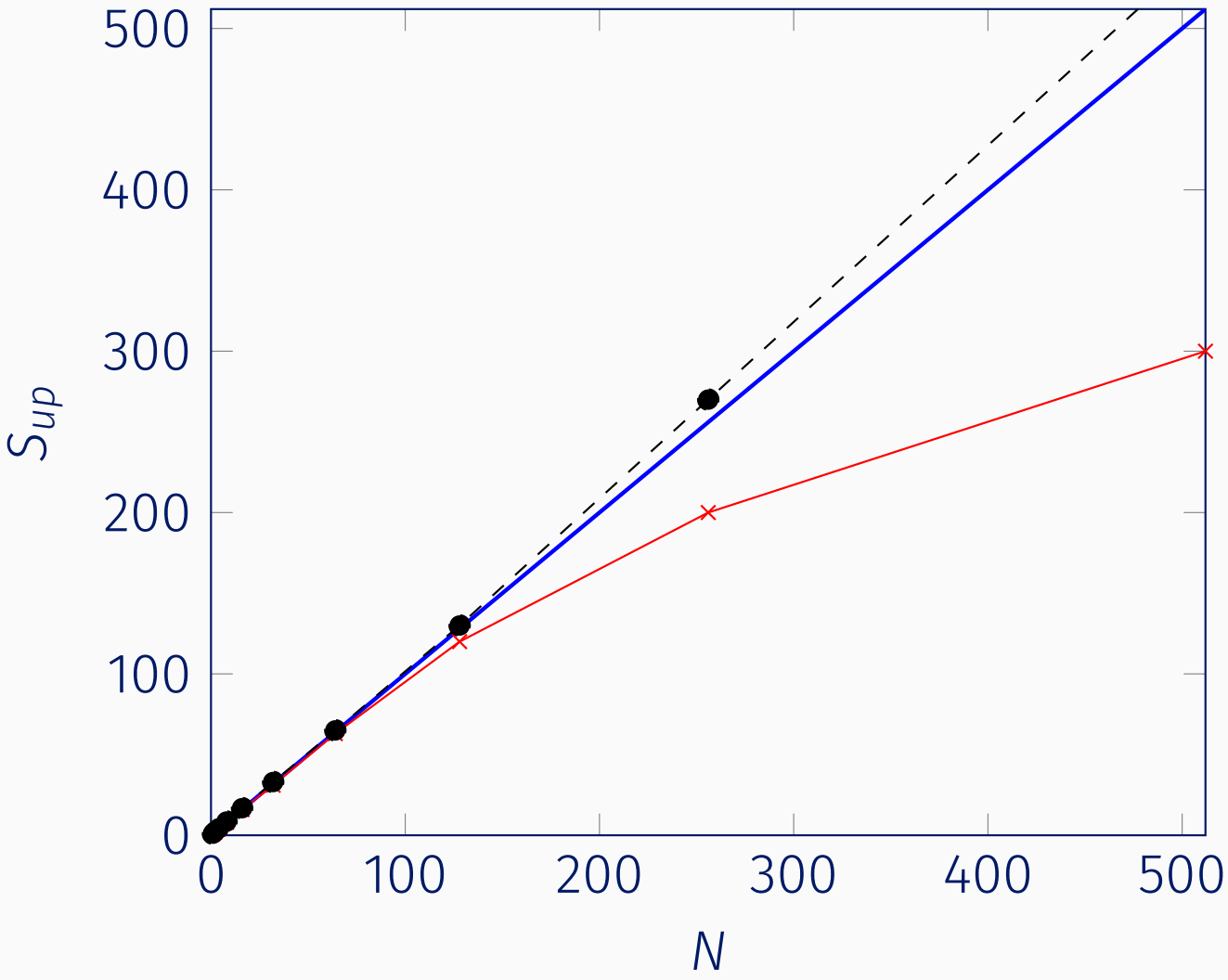
Gustafson's law



Super Linear speedup?

Sometimes things work better than expected. In case of parallel efficiency this means better than theoretical performance. This can happen for a number of reasons. The most embarrassing one is sub-optimal serial implementation, but also because of the non homogenic memory layout (L1, L2, etc.). The problem once partitioned and distributed might require less memory calls, and consequently super-linear speedup can be observed.

Super Linear speedup?

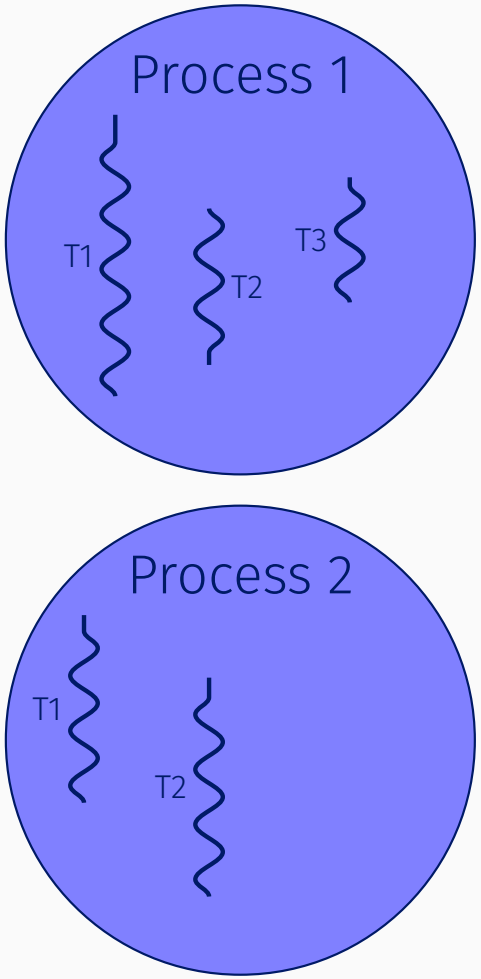


Thread and Process

Thread and Process

Process

- Independent from others
- Possesses a separate addressing and memory
- Any form of communication is handled by system calls (shared memory, semaphores, files, network, etc.)
- For us communication will be handled by an MPI (Message Passing Interface) implementation



Thread

- Exists within a process
- Threads share addressing space within a process

Thread

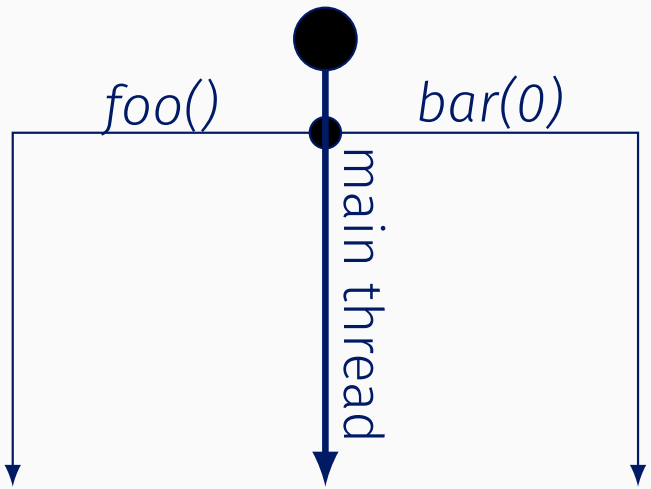
We will start our adventure with getting more than one core to work by looking at threads. The objective is to get familiar with the parallel way of looking at a problem using simple, available within the C++ standard tools and than to move onto a dedicated library that would handle thread prallelism for us.

```
#include <thread>
```

But first we need to ”see” the threads and processes that we create. We will use **pstree** command line tool to do just that.

Starting a thread

```
#include <thread>
void foo(){
  for(int i=0; i<10; ++i){
   cout << "foo sleeps " << i << endl;
   sleep ( 1 );
  }
}
void bar(int x){
  for(int i=0; i<10; ++i){
   cout << "bar x=" << x << " bar sleeps "
   << i << endl;
   sleep ( 1 );
  }
}
...
std::thread first (foo);
std::thread second (bar,0);
```



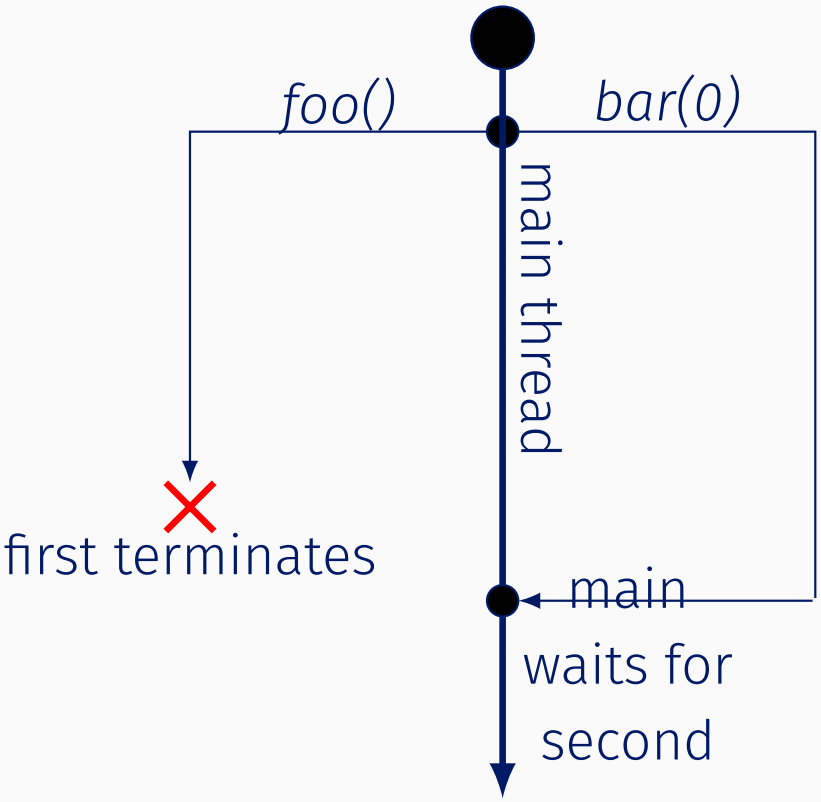
Note: It is bad practice to initiate threads and letting them run without guaranteeing they will terminate before the main thread terminates - use join.

Threads - *join & detach*

- join is used block the current thread until the thread being joined finishes
- detach separates the thread oject alowing the execution past scope

```
#include <thread>

...
std::thread first (foo);
std::thread second (bar,0);
// detach first from main
first.detach()
...
//main thread does sth
...
// wait for second to terminate
second.join()
```



Examine and run the code

See the code in the repository, and try to compile and run thread1.cpp and thread2.cpp. Is all working as it should?

With thread power comes thread responsibility

Uncle Ben

Problems with threads and how to avoid them

- Race condition.
- Atomic operations (RMW)
- Mutex - mutual exclusion lock.

Accessing resources - race condition

- Threads spawn within a process and share memory and other resources.
- The most dreaded problem is the so called ”race condition”.
- A data race (race condition) occurs if multiple threads access an object and at least one of them modifies it.
- Thread synchronisation is critical to avoid UB and crashes.

Compile and run thread3.cpp

Atomic operations (operacja niepodzielna)

Atomic operation are not interrupted by concurrent operations. That is Read-Modify-Write trio is performed - Kuba was talking about grabbing stuff from memory to cache operating on and putting back. This is enforced by hardware and guarantees that once an atomic operation starts it is finished before interruption. This will allow to place 'locks' in the code.

Compare the code below using https://godbolt.org/

```
#include <atomic>

void a(int &a)
{
    a++;
}
```

```
#include <atomic>

void b(std::atomic<int> &a)
{
    a++;
}
```

Do you notice difference?

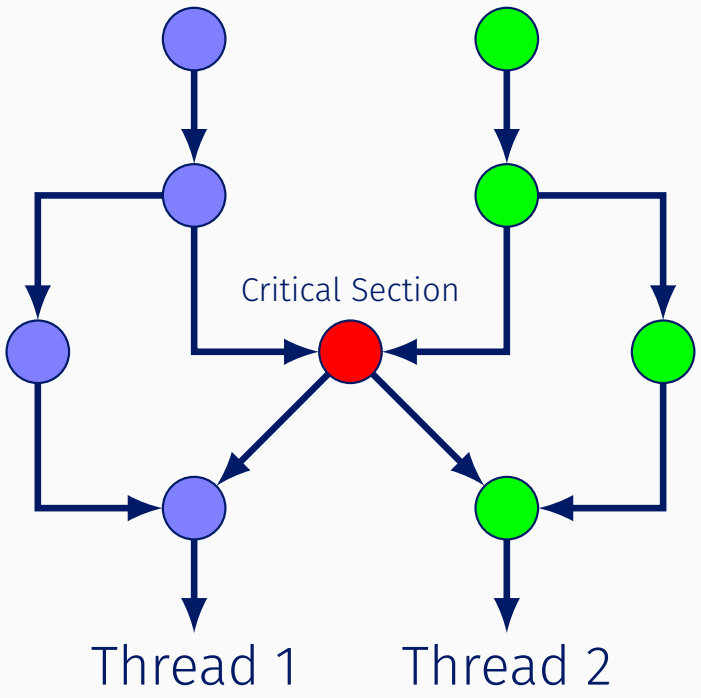
Atomic operations

The main takeaway from this experiment is that modern CPUs have direct support for atomic integer operations, for example the LOCK prefix in x86, and std::atomic basically exists as a portable interface to those instructions. Note: std::atomic works for integers, see the effect on doubles, does it work.

Task: Use thread3.cpp as a tempate and modify it using std::atomic. Ist the UB still a problem? Does it work for doubles?

Mutex - mutual exclusion lock

We have seen that concurrent access to a shared resource can lead to UB and must be avoided. The region that this can happen is called **critical section** and should be protected from being accessed at the same time. The locking mechanism is called mutex. See thread4.cpp, compile it and run. Mutex allows to define the critical section and protects it from concurrent access.



TBB

Thread Building blocks

TBB is a threading library designed to be easy to use. In a sense it is to threads what STL is to data containers, a template library to ease creation of threads, maintaince, load balancing and scheduling of work.

TBB is free, portable, relatively easy and in principle allows to add parallelism to existing codes.

We will have a look at two TBB constructs parallel_for and parallel_reduce and apply both to parallelize two types of problems.

Tasks

An embarrassingly parallel problem

We are going to start with a problem, that is considered to be "embarrassingly parallel". This means it is, by it's nature very easy to perform in parallel, mostly because of *data locality* and independence of operations performed by each of the working parties.

Example: Consider a very long vector y.

1. Fill it with values that are the result of the following (making no sense, but appropriately complicated) expression: $y_i = sin(x_i^\pi) * cos(x_i^\pi) * tan(x_i^\pi) - atan(exp(a)/(a+1))$, for some evenly distributed x.
2. Find the average off all y_i

Use serial.cpp

parallel_for

We will now parallelize the first part of our problem using the tbb::parallel_for.

```
tbb::parallel_for( range, kernel );
```

range - A range over which to iterate.

kernel - a function that takes as argument the range to iterate through.

Note: Do a race condition, show performance degeneration.

False Sharing

We will now illustrate a situation which can lead to performance degradation and stems from the fact that values used by different threads might be placed to close to each other in memory.

See the tbb_flase_sharing.cpp

parallel_reduce

Taks: Suplement the code by adding summation over all elements of y.

```
double sum = 0;
for (double val : y)
{
    sum += val;
}
```

This summation can not be parallelized using parallel_for since now there is a shared variable. We could use a mutex, or make use of tbb::parallel_reduce.

```
auto result =
  tbb::parallel_reduce( range, identity_value,
                        kernel, reduction_function );
```

range - as before
identity_value - identity value for reduction, 0 for summation, 1 for multiplication reduction_function - standard reduction function (a class with member function operator())
kernel - function that performs the reduction

More complex example

...