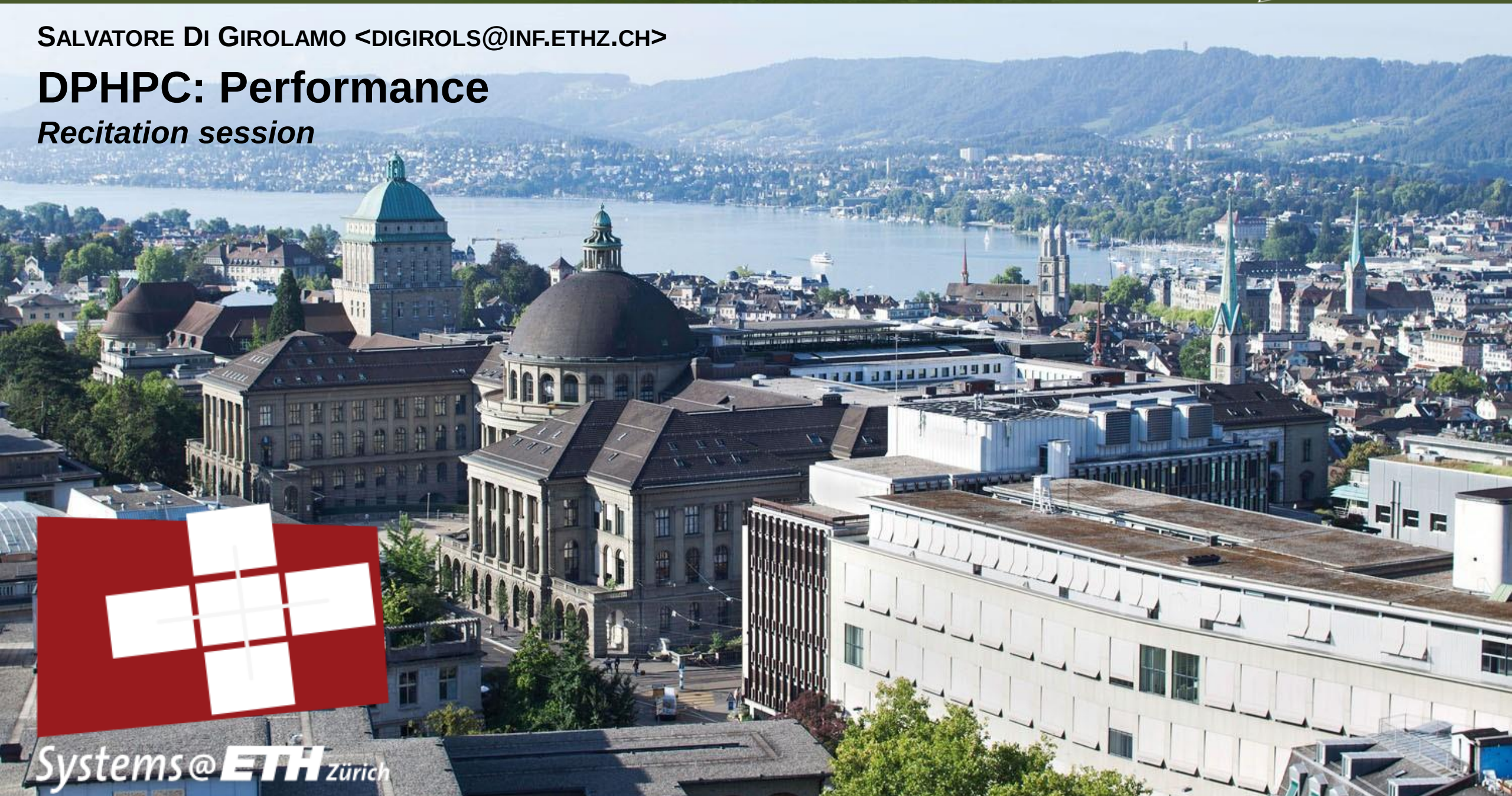
SALVATORE DI GIROLAMO <DIGIROLS@INF.ETHZ.CH>

DPHPC: Performance

Recitation session



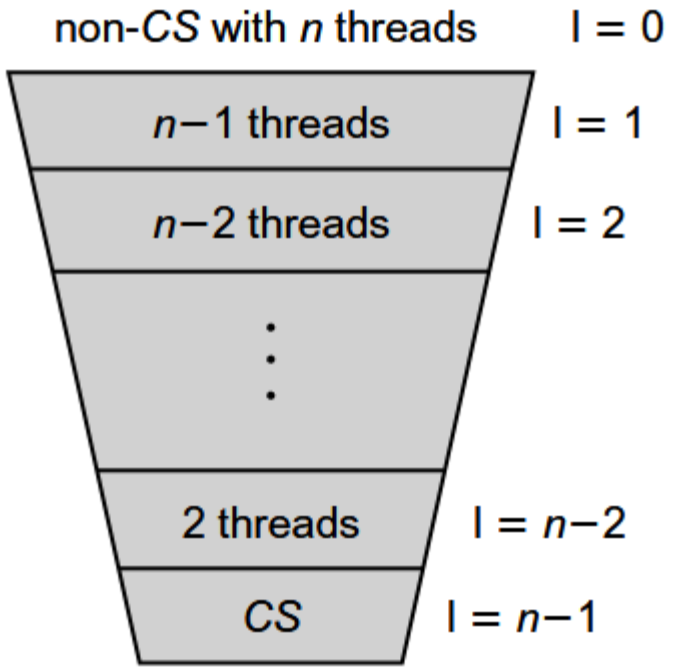
DCO Cluster

- **16 nodes**
- **Batch system: SLURM**
  - Submit with: sbatch <job script>
  - Check queue state with: squeue –u $USER
  - Cancel jobs with: scancel <jobid>
  - Login node: dco-node129
    ssh teamX@dco-node129.dco.ethz.ch
- **Wall time 20mins**
  - Ask if you need more
- **Nodes: AMD Opteron 6212 (8 cores)**
- **Network: 10Gbit**
- **1 account per team (email me to get login credentials)**

```
#!/bin/bash

#SBATCH --job-name=test
#SBATCH --output=slurm-%j.out

#SBATCH --nodes 4
#SBATCH --ntasks=16
#SBATCH --time=00:10:00

srun ./a.out
```

Assignment: Filter Lock

```
1  class Filter implements Lock {
2    int[] level;
3    int[] victim;
4    public Filter(int n) {
5      level  = new int[n];
6      victim = new int[n]; // use 1..n-1
7      for (int i = 0; i < n; i++) {
8         level[i] = 0;
9      }
10   }
11   public void lock() {
12     int me = ThreadID.get();
13     for (int i = 1; i < n; i++) { //attempt level 1
14       level[me] = i;
15       victim[i] = me;
16       // spin while conflicts exist
17       while ((∃k != me) (level[k] >= i && victim[i] == me)) {};
18     }
19   }
20   public void unlock() {
21     int me = ThreadID.get();
22     level[me] = 0;
23   }
24 }
```



From: The Art of Multiprocessor Programming

False sharing

- **Why does it happen?**

Amdahl's Law

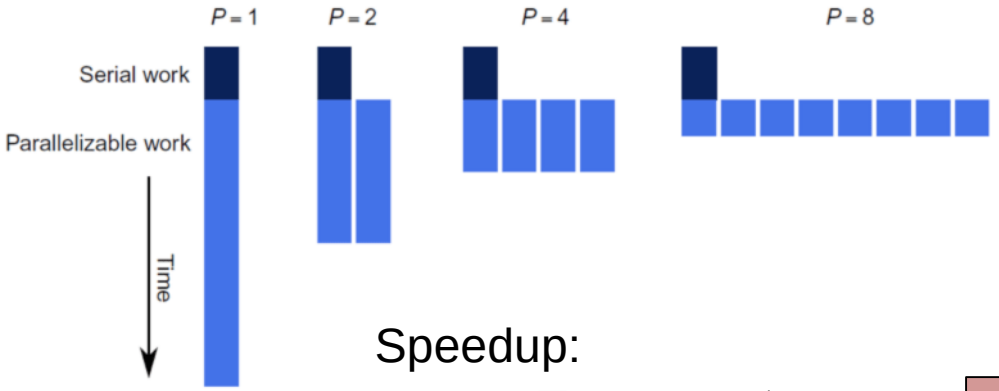
Time of sequential program with f as the fraction not affected by the parallelization:

$$T_1 = fT_1 + (1-f)T_1$$

Time of parallel program:

$$T_P \geq fT_1 + \frac{(1-f)T_1}{P}$$

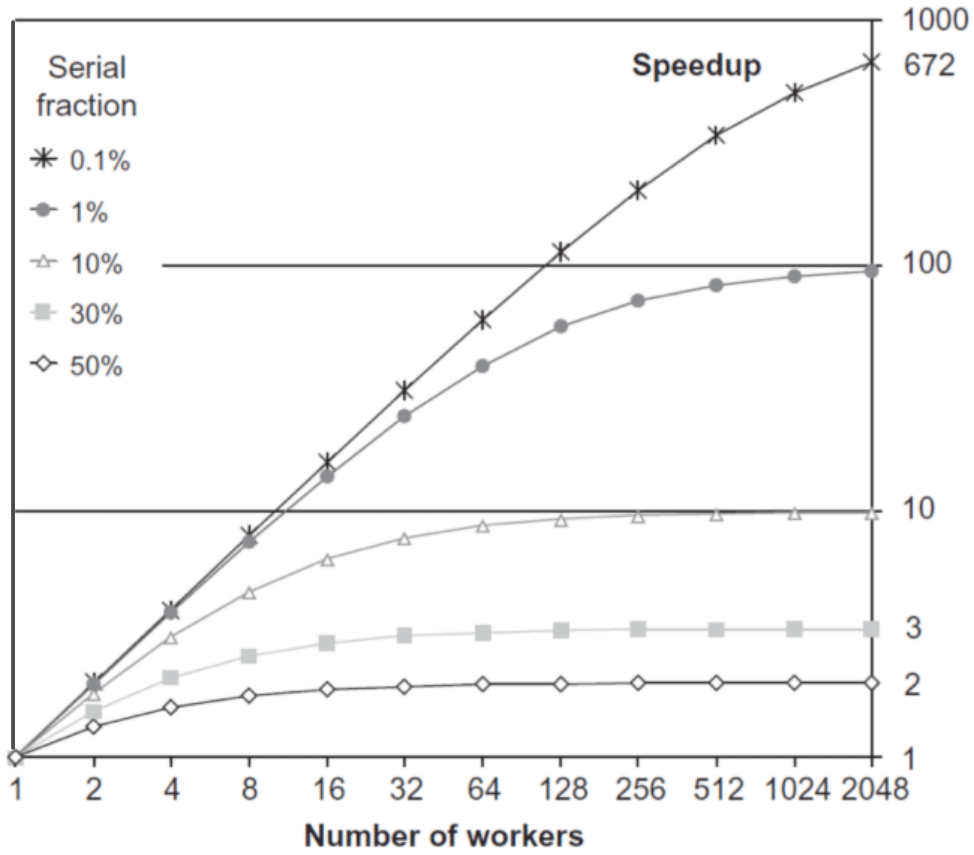
$$T_\infty = fT_1$$



Speedup:

$$S_P = \frac{T_1}{T_P} \leq \frac{1}{\frac{1-f}{P} + f}$$

$$S_\infty \leq \frac{1}{f}$$



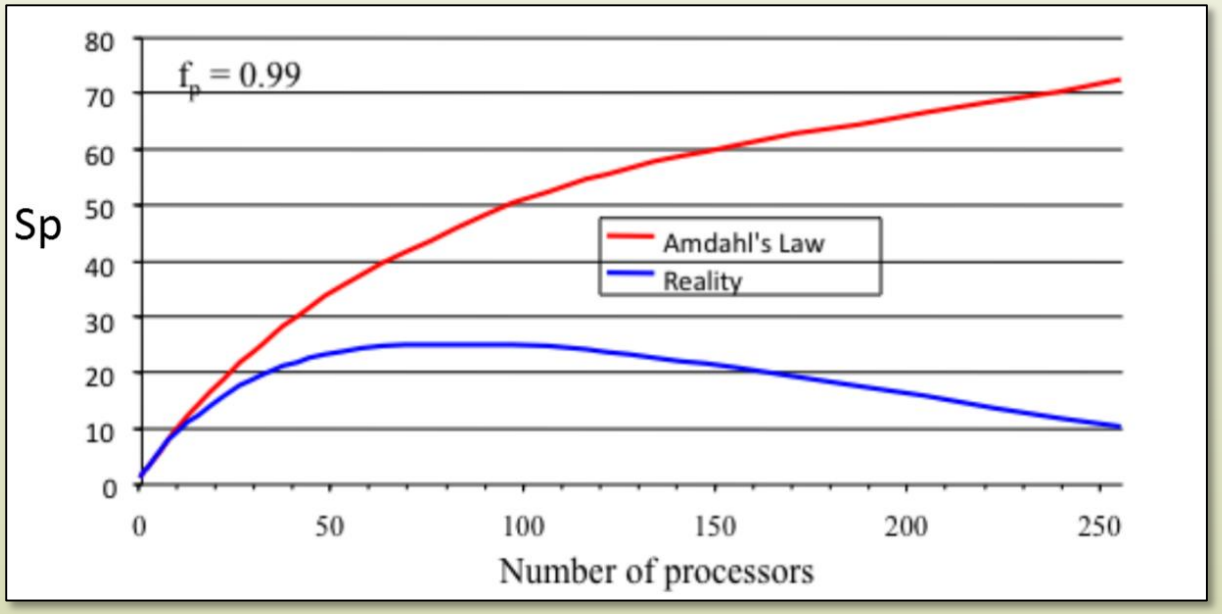
Amdahl's Law

Time of sequenti
by the parallel

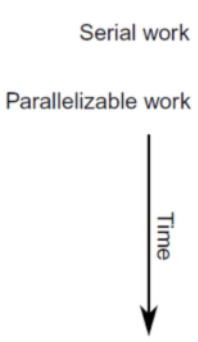
Time of parall

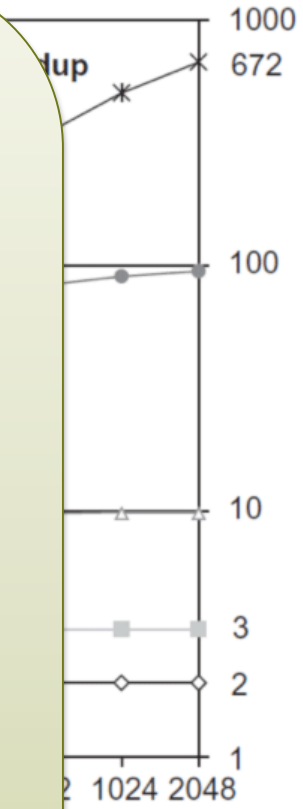
It's like to see the glass as half empty but…

It could be even worse!



Possible factors: *load balancing, communication costs, I/O, scheduling*





$$\frac{}{P} + f$$

Amdahl's Law vs Gustafson-Barsis' Law

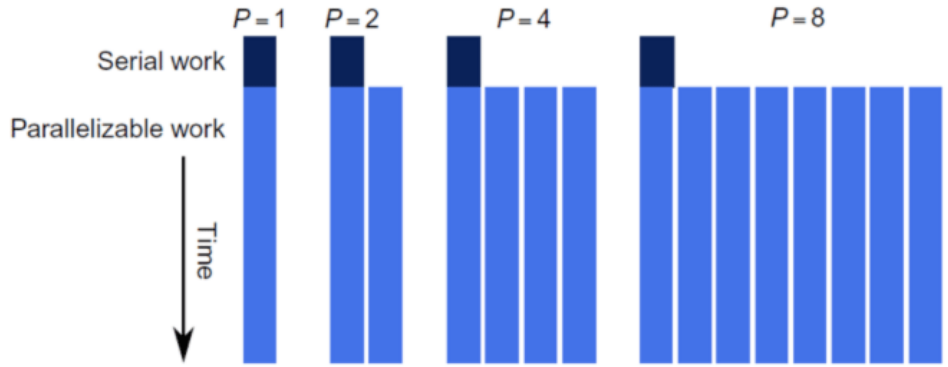
> *…speedup should be measured by scaling the problem to the number of processors, not by fixing the problem size.*
> — John Gustafson

Time of sequential program with α as the fraction not affected by the parallelization on P-processors machine:

$$T_1 = \alpha T_1 + (1 - \alpha) P T_1$$

Time of parallel program:

$$T_P = \alpha T_1 + (1 - \alpha) T_1$$



Speedup:

$$S_P = \frac{T_1}{T_P} \leq \alpha + P(1 - \alpha)$$

Note: no parallel overheads are taken into account here (as in Amdahl's)!

Quiz

- **Speedup**
  - How well something responds when adding more resources
  - **What's your base case?** The best serial version or a single parallel process?

- **Efficiency**
  - Gives an idea on the "utilization" degree of the computing resources

- **Strong Scaling**
  - Problem size stays fixed as the number of processing elements are increased

- **Weak Scaling**
  - Problem size increases as the number of processing elements are increased

Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! but at what cost?. In Proceedings of the 15th USENIX conference on Hot Topics in Operating Systems (HOTOS'15), George Candea (Ed.). USENIX Association, Berkeley, CA, USA, 14-14.

Exercise 1

Assume 1% of the runtime of a program is not parallelizable. This program is run on 61 cores of a Intel Xeon Phi. Under the assumption that the program runs at the same speed on all of those cores, and there are no additional overheads, what is the parallel speedup?

Amdahl's law assumes that a program consists of a serial part and a parallelizable part. The fraction of the program which is serial can be denoted as B — so the parallel fraction becomes $1 - B$. If there is no additional overhead due to parallelization, the speedup can therefore be expressed as

$$S(n) = \frac{1}{B + \frac{1}{n}(1 - B)}$$

For the given value of $B = 0.01$ we get $S(61) = 38.125$.

Exercise 2

Assume 0.1% of the runtime is not parallelizable. The program also invokes a broadcast operation, that add overhead depending on the number of cores involved. There are two broadcast implementations available. One adds a parallel overhead of $0.0001n$, the other one $0.0005 \log n$. For which number of cores do you get the highest speedup for both implementations?

$$S_1(n) = \frac{1}{0.001 + \frac{1}{n}0.999 + 0.0001n}$$

$$S_2(n) = \frac{1}{0.001 + \frac{1}{n}0.999 + 0.0005log(n)}$$

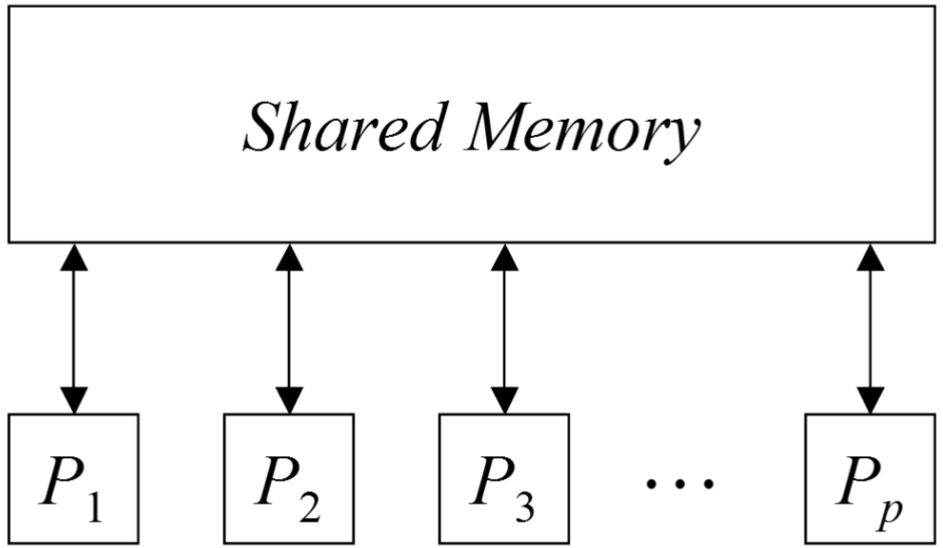
We can get the maximum of these terms if we minimize the term in denominator.

$$\frac{d}{dn}0.001 + \frac{1}{n}0.999 + 0.0001n = 0 \leftrightarrow 0.0001 - \frac{0.999}{n^2} = 0 \leftrightarrow n \approx 100$$

$$\frac{d}{dn}0.001 + \frac{1}{n}0.999 + 0.0005log(n) = 0 \leftrightarrow \frac{0.005n0.999}{n^2} = 0 \leftrightarrow n = 1998$$

PRAM: Parallel Random Access Machine

- **P processes with shared memory**
- **Ignores communications and synchronization**
- **Instruction are composed by 3 phases:**
  - Load data from shared memory (if needed)
  - Perform computation (if any)
  - Store data in shared memory (if needed)
- **Any process can read/write to any memory cell**
  - How are conflicts handled?



PRAM: Conflicting Accesses

- **EREW: Exclusive Read / Exclusive Write**
  - No two processes are allowed to read or write to the same memory cell simultaneously
- **CREW: Concurrent Read / Exclusive Write**
  - Simultaneous reads are allowed; only one process can write
- **CRCW: Concurrent Read / Concurrent Write**
  - Simultaneous reads and write to the same memory cell are allowed
  - Detecting CRCW: A special code for “detected collision” is written
  - Priority CRCW: processors assigned fixed distinct priorities, highest priority wins
  - Random CRCW: one randomly chosen write wins
  - Common CRCW: all processors are allowed to complete write if and only if all the values to be written are equal

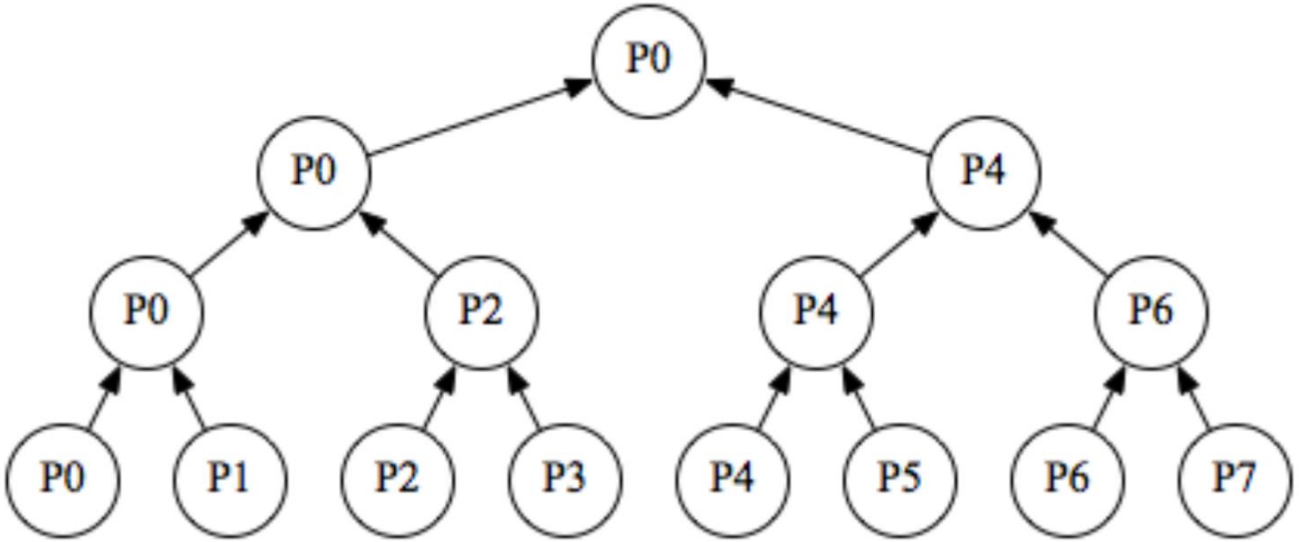
Weak **Strong**

EREW < CREW < CRCW-D < CRCW-C < CRCW-R < CRCW-P

http://homes.cs.washington.edu/~arvind/cs424/notes/l2-6.pdf

PRAM: Reduction

- Reduce p values on the p-processor EREW PRAM in $O(logp)$ time
- The algorithm uses exclusive reads and writes
- It's the basis of other EREW algorithms



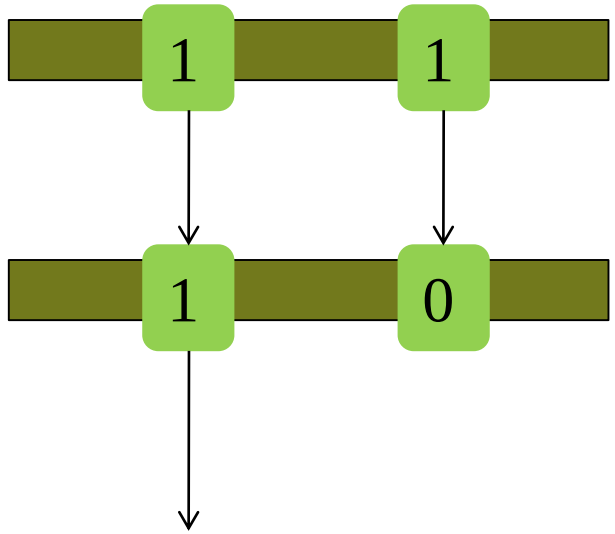
PRAM: First 1

- **Computing the position of the first "1" in the sequence of 0's and 1's in a constant time.**

Algorithm A

(2 parallel steps and n^2 processors)

```
for each 1≤ i<j ≤ n do in parallel
   if C[i] =1 and C[j]=1 then C[j]:=0
for each 1≤ i ≤ n do in parallel
   if C[i] =1 then FIRST-ONE-POSITION:=i
```

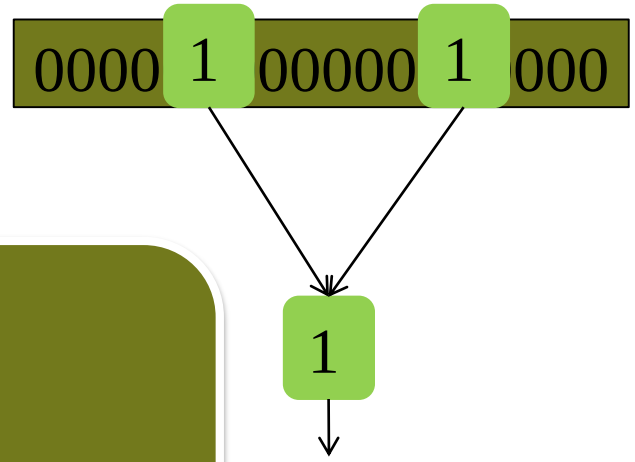


www.csc.liv.ac.uk/~igor/COMP308/ppt/Lect_5.ppt

PRAM: First 1 – Reducing Number of Processors

Algorithm B: it reports if there is any "1" in the table.

```
There-is-one:=0
for each 1≤ i ≤ n do in
   if C[i] =1 then The
```



How many processors we need?

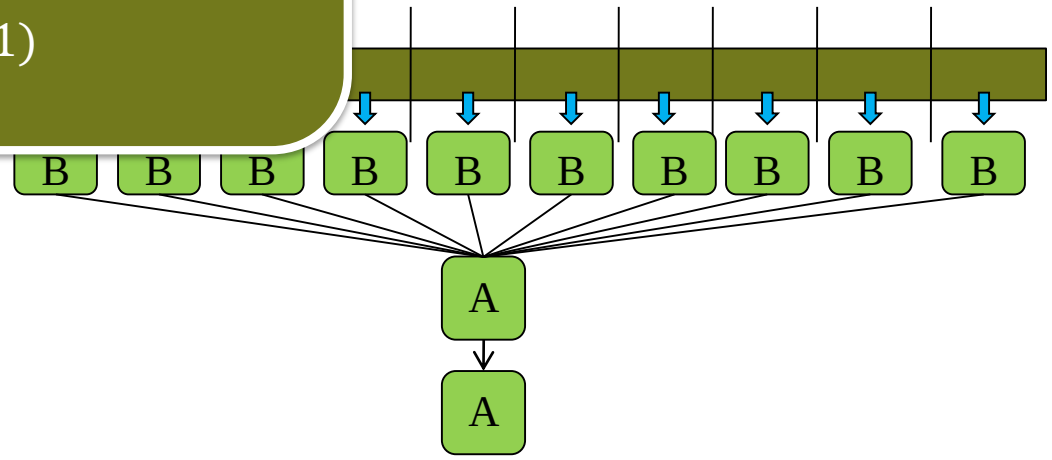
$(\sqrt{n})^2 = n$

What's the complexity?

$3\ parallel\ steps \rightarrow O(1)$

Merge A and B

1. Partition table C into se
2. In each segment apply t
3. Find position of the first one in these sequence by applying algorithm A
4. Apply algorithm A to this single segment and compute the final value



Exercise 3

How can we find the minimum from an unordered collection of n natural numbers on EREW-PRAM machine?

We can find the minimum from an unordered collection of n natural numbers by performing a reduction along a binary tree: In each round, each processor compares two elements, and the smaller element gets to the next round, the bigger one is discarded. What is the work and depth of this algorithm?

The dependency graph of this computation is a tree with $log_2(n)$ levels. Therefore the longest path, which is equal to the depth/span has length $log_2(n)$. The tree contains $2n - 1$ nodes, which is equal to the work.

Exercise 4

Develop an algorithm which can find the minimum in an unordered collection of n natural numbers in $O(1)$ time on a CRCW-PRAM machine.

- Assume the list is stored in an array A.
- Create an additional array $tmp[n]$ initialiazed with $true$.
- We use $O(n^2)$ processors, labelled $p(i,j)$ with $0 \leq i,j \leq n$.
- Each processor $p(i,j)$ checks if $A[i] > A[j]$.
  - If true then tmp$[i]$ is set to false (it cannot be the minimum)
  - Otherwise nothing is done
- At the end we have only one element of tmp set to true, say $tmp[k]$. The minimum element of A is $A[k]$.

A:	3	5	1	8
tmp:	T	T	T	T

P(0, 1):
If (A[0] > A[1]) tmp[0] = F;

P(0, 2):
If (A[0] > A[2]) tmp[0] = F;

P(0, 3):
If (A[0] > A[3]) tmp[0] = F;

P(1, 0):
If (A[1] > A[0]) tmp[1] = F;

P(1, 2):
If (A[1] > A[2]) tmp[1] = F;

P(1, 3):
If (A[1] > A[3]) tmp[1] = F;

P(2, 0):
If (A[2] > A[0]) tmp[2] = F;

P(2, 1):
If (A[2] > A[1]) tmp[2] = F;

P(2, 3):
If (A[2] > A[3]) tmp[2] = F;

P(3, 0):
If (A[3] > A[0]) tmp[3] = F;

P(3, 1):
If (A[3] > A[1]) tmp[3] = F;

P(3, 2):
If (A[3] > A[2]) tmp[3] = F;