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

DPHPC: Performance

Recitation session

Systems@ETH zürich

Administrativia

- **Reminder: Project presentations next Monday**
  - 10min each
  - Presentations order as teams are displayed on the web-page
  - Send me an e-mail by Sunday if you have particular time constraints (already got some)
  - Send slides at digirols@inf.ethz.ch by Sunday 11/6 11:59pm
  - Rough guidelines:

    Present your plan

    Related work

    Preliminary results (milestones)

  - **Main goal:** gather feedbacks (so present some details)
  - Ideally one presenter (make sure to rotate for other presentations!)

Amdahl's Law

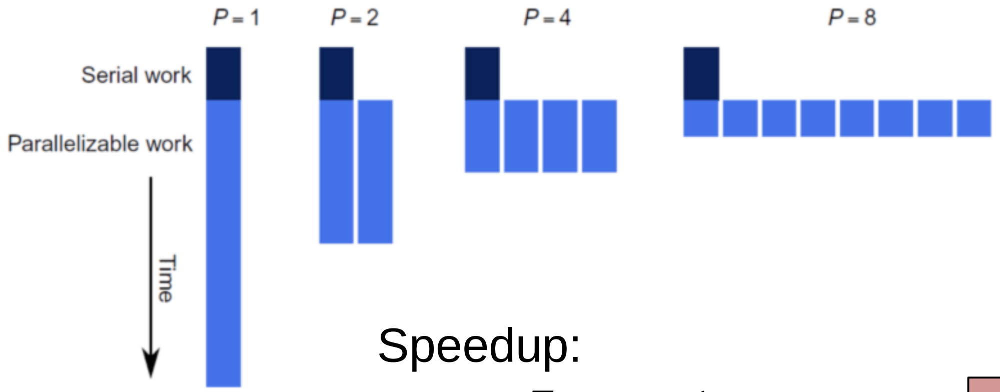
Time of sequential program with f as the fraction not affected by the parallelization:

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

Time of parallel program:

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

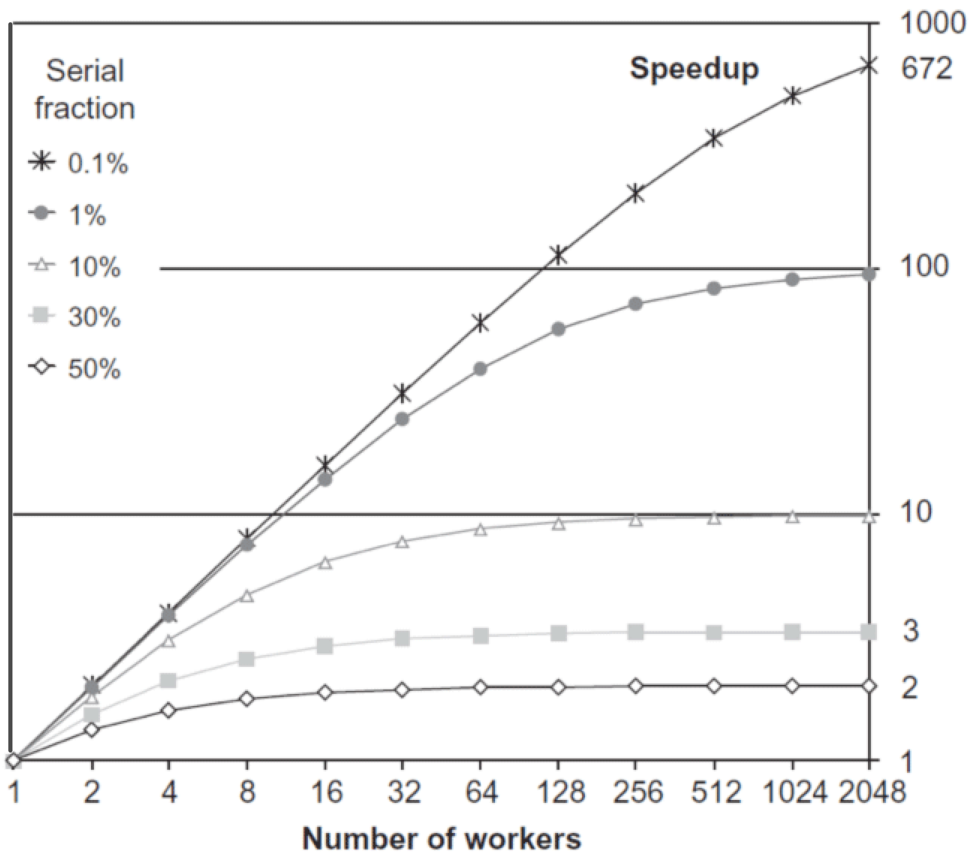
$$\boxed{T_\infty = fT_1}$$



Speedup:

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

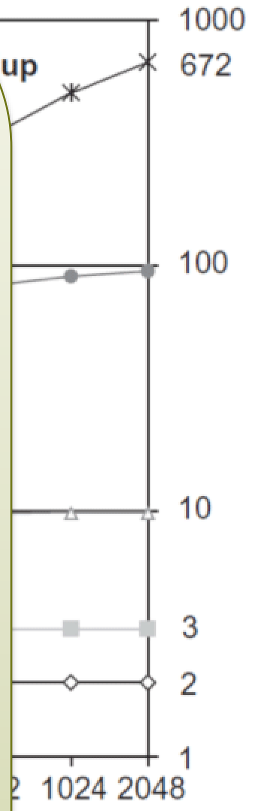
$$\boxed{S_\infty \leq \frac{1}{f}}$$

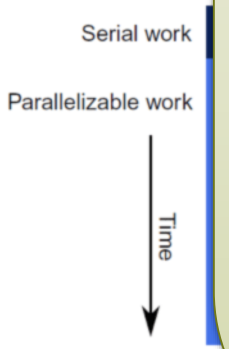


Amdahl's Law

Time of sequenti
by the paralleli

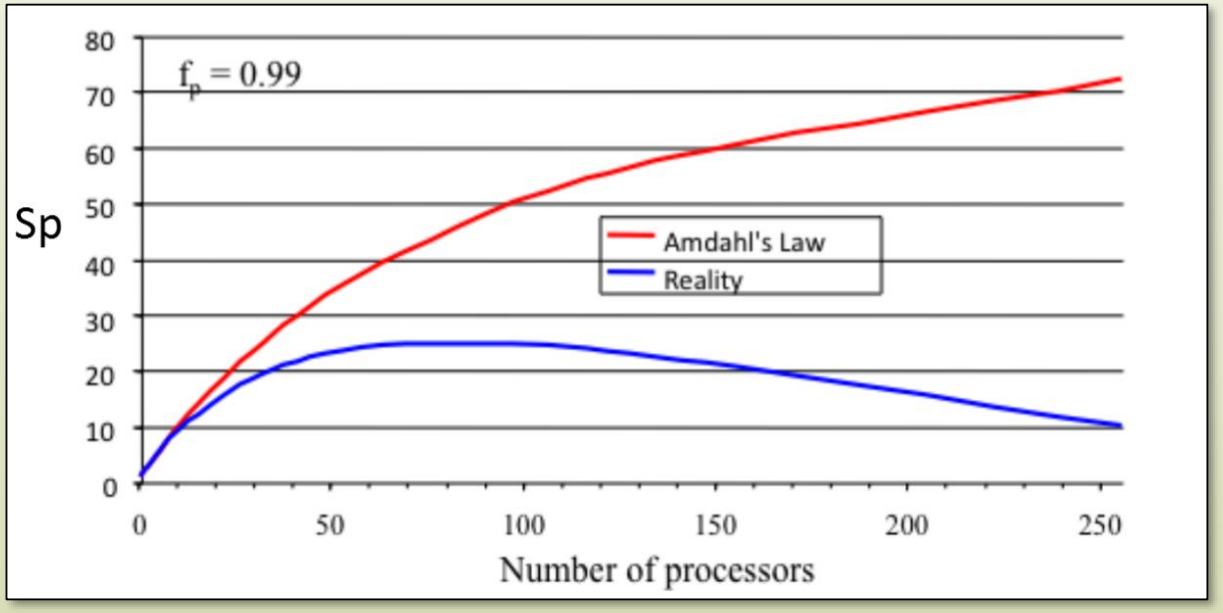
Time of paralle





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*

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 f as the fraction not affected by the parallelization on P-processors machine:

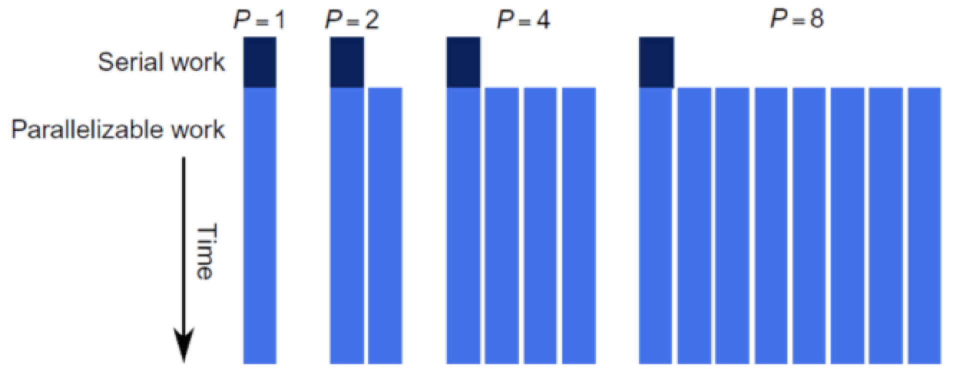
$$T_1 = \alpha T_1 + (1-\alpha)PT_1$$

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!

Quiz

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

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?

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

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

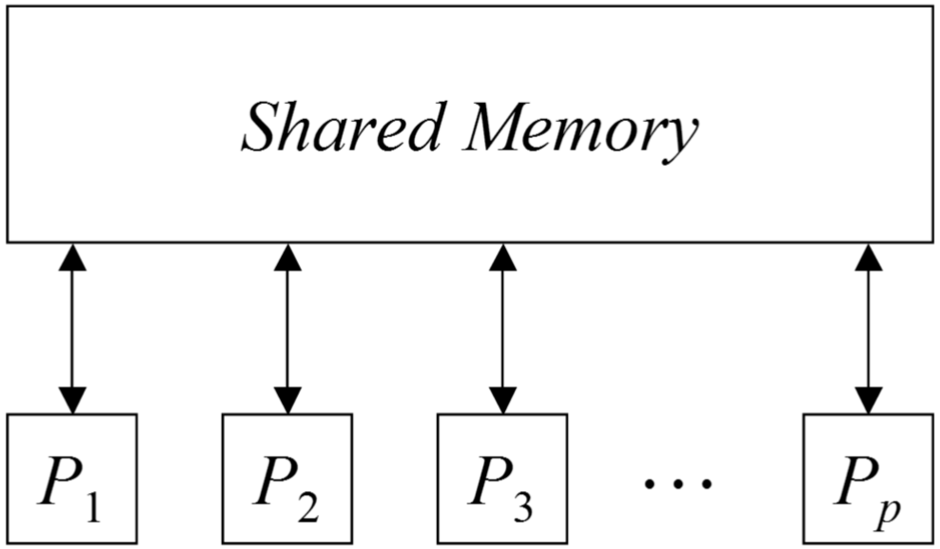
We can get the maximum of these terms if we minimize the term in denominator.

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

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

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 conflicts are handled?



PRAM: Conflicting Accesses

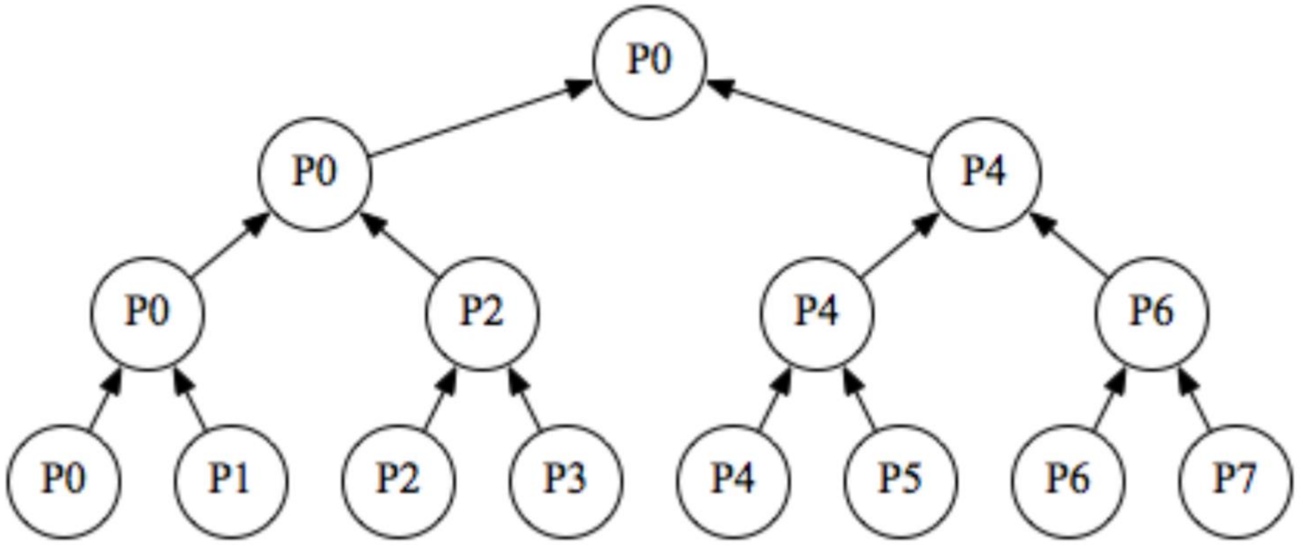
- **EREW: Exclusive Read / Exclusive Write**
  - No two processes are allowed to read or write to the same memory cell simultaneously
- **CREW: Concurrent Read / Exclusive Write**
  - Simultaneous reads are allowed; only one process can write
- **CRCW: Concurrent Read / Concurrent Write**
  - Simultaneous reads and write to the same memory cell are allowed
  - Priority CRCW: processors assigned fixed distinct priorities, highest priority wins
  - Arbitrary 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

PRAM: Reduction

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



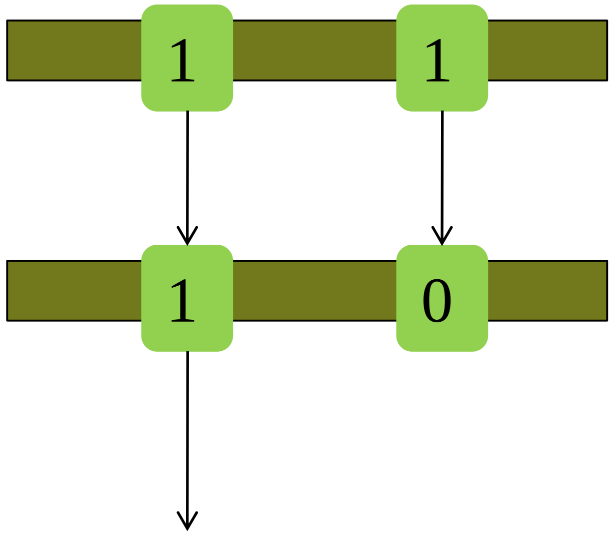
PRAM: First 1

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

Algorithm A

(2 parallel steps and n^2 processors)

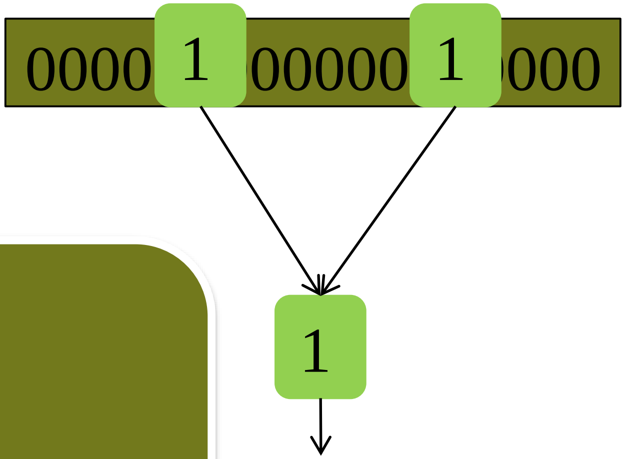
```
for each 1≤ i<j ≤ n do in parallel
   if C[i] =1 and C[j]=1 then C[j]:=0
for each 1≤ i ≤ n do in parallel
   if C[i] =1 then FIRST-ONE-POSITION:=i
```



PRAM: First 1 – Reducing Number of Processors

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

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

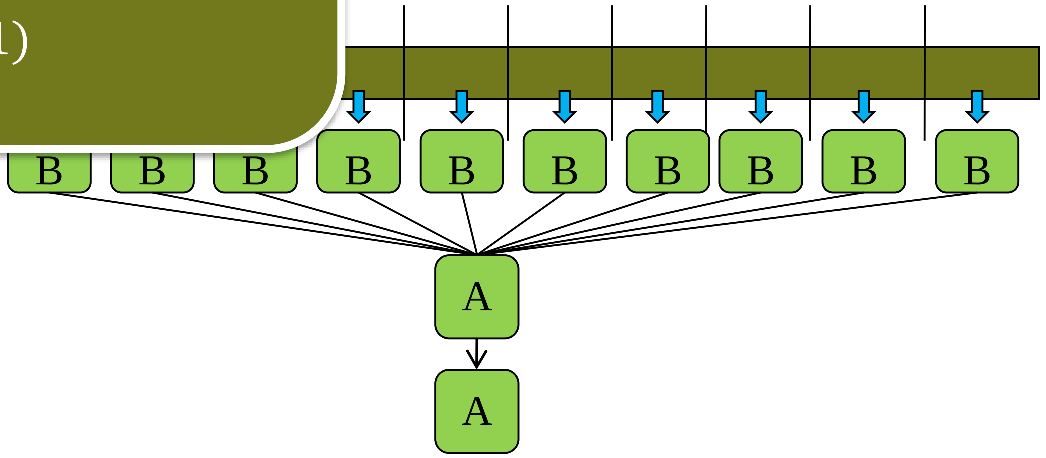


Merge A and B

1. Partition table C into se
2. In each segment apply
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

How many processors we need?
$(\sqrt{n})^2 = n$

What's the complexity?
$3 \; parallel \; steps \; \rightarrow O(1)$



PRAM: Odd-Even Merge Sort

- **Odd-Even Merge**

```
Merge(x_1, ..., x_n, y_1 ... y_n)

Merge(x_1, x_3, x_5, y_2, y_4, y_6 ...) to get a_1 ... a_n
Merge(x_2, x_4, x_6, y_1, y_3, y_5 ...) to get b_1 ... b_n
for i=1 to n do z_2i-1=min(a_i, b_i)
                z_2i  =max(a_i, b_i)
```

$O(\log n)$ with n processors

- **Odd-Even Merge Sort**

```
Sort(x_1, ... x_n,)

Merge(Sort(x_1, ... x_n/2), Sort(x_n/2, ... x_n))
```

$O(\log^2 n)$ with n processors

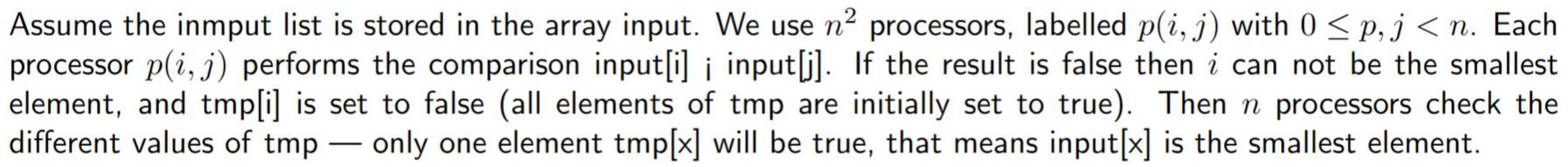
Exercise 3

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 inmput list is stored in the array input. We use n^2 processors, labelled $p(i,j)$ with $0 \leq p, j < n$. Each processor $p(i,j)$ performs the comparison input[i] ¡ input[j]. If the result is false then i can not be the smallest element, and tmp[i] is set to false (all elements of tmp are initially set to true). Then n processors check the different values of tmp — only one element tmp[x] will be true, that means input[x] is the smallest element.

Public Lecture: *Scientific Performance Engineering in HPC*

Invitation to a lecture by Prof. Dr. Torsten Hoefler (Scalable Parallel Computing Lab at ETH Zurich)

Date: Tuesday, November 8, 2016
Time: 17:15
Location: HG F 5, ETH Zurich

Abstract:
We advocate the usage of mathematical models and abstractions in practical high-performance computing. For this, we show a series of examples and use-cases where the abstractions introduced by performance models can lead to clearer pictures of the core problems and often provide non-obvious insights. We start with models of parallel algorithms leading to close-to-optimal practical implementations. We continue our tour with distributed-memory programming models that provide various abstractions to application developers. A short digression on how to measure parallel systems shows common pitfalls of practical performance modeling. Application performance models based on such accurate measurements support insight into the resource consumption and scalability of parallel programs on particular architectures. We close with a demonstration of how mathematical models can be used to derive practical network topologies and routing algorithms. In each of these areas, we demonstrate newest developments but also point to open problems. All these examples testify to the value of modeling in practical high-performance computing. We assume that a broader use of these techniques and the development of a solid theory for parallel performance will lead to deep insights at many fronts.