Introduction to Parallel Computing

Performance, Parallel Computing Objectives

Jesper Larsson Träff
TU Wien
Institute of Computer Engineering
Parallel Computing

Parallel computing objectives

(Traditional) Objective:
Solve given computational problem faster

- Than what?

- How do we account for better performance?
- How many parallel resources (processors) can be productively used?
- What are obstacles to good parallel performance?

Non-specific model:
p dedicated parallel processors collaborate to solve given problem of input size n.

- Processors work independently (local memory, program, MIMD), but start at the same time
- Processors are occupied until last processor finishes
- Collaboration can incur overheads (communication, coordination, algorithmic)

Recall: Parallel computing assumes dedicated processors:
We must “pay” for system time until all processors are done

Let a computational problem P with input I be given

- Seq: Sequential algorithm (or implementation) solving P(I)
- Par: Parallel algorithm (or implementation) solving P(I)

I(n) input of size n. P(n) short-hand for P(I(n)) in the worst case (either, we quantify over all inputs, or the input is not important…)

- Theory: Algorithm in given, specific model
- Practice: Concrete implementation for some specific type of parallel computer

Let

- Tseq(n): Time for 1 processor to solve P(n) using Seq
- Tpar(p,n): Time for p processors to solve P(n) using Par, time for last/slowest processor to finish

The gain in moving from sequential computation with algorithm Seq to parallel computation with algorithm Par is expressed as the <u>speed-up</u> of Par over Seq:

$$S_p(n) = Tseq(n)/Tpar(p,n)$$

Note: Both parameters p and n can be varied

Goal: Achieve as large speed-up as possible (for some n, all n) for as many processors as possible

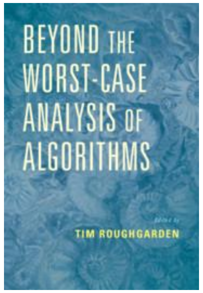
What exactly is Tseq(n), Tpar(p,n)?

- Time for some algorithm for solving problem?
- Time for a specific algorithm (Seq, Par) for solving problem?

- Time for best known algorithm for problem?
- Time for best possible algorithm for problem?

- Time for specific input of size n, average case, worst case, …?
- Asymptotic time, large n, large p?

- Do constants matter, e.g. O(f(p,n)) or 25n/p+3ln (4 p/n) … ?



Tim Roughgarden: Beyond worst-case analysis. Comm. ACM 62(3): 88-96 (2019)

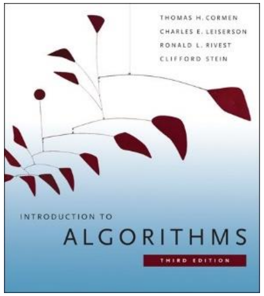
Choose sequential algorithm (theory), choose an implementation of this algorithm (practice)

Tseq(n):

- Theory: Number of instructions (or other critical cost measure) executed in the worst case for inputs of size n
- The number of instructions carried out is the required work

- Practice: Measured time (or other parameter) of execution over some inputs (experiment design)

Theory and practice:
Always state baseline sequential algorithm&implementation



Examples:

Tseq(n) = O(n), =Θ(n):	Finding maximum of n numbers in unsorted array; prefix-sums
Tseq(n,m) = Θ(n+m) :	Merging of two sequences; BFS/DFS in graph
Tseq(n) = O(n log n):	Comparison-based sorting
Tseq(n,m) = O(n log n + m):	Single-source Shortest Path (SSSP)
Tseq(n) = $O(n^3)$:	Matrix multiplication, input two nxn matrices
…	…

Can be solved in $o(n^3)$, by Strassen etc.

Standard, worst-case, asymptotic complexities

Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms. 3rd ed., MIT Press, 2009

Practice:

- Construct meaningful inputs to measure Tseq(n): experiment design, experimental methodology
- Worst-case not always possible, not always interesting; best case, average case? (what is that?)
- Experimental methods to get stable, accurate, repeatable Tseq(n): Repeat measurements many times (thumb rule: average over at least 30 repetitions. Be very careful!)

Experimental science:
Always some assumptions about realism, repeatability, regularity, determinism, …

Practice:

- Construct meaningful inputs to measure Tseq(n): experiment design, experimental methodology
- Worst-case not always possible, not always interesting; best case, average case? (what is that?)
- Experimental methods to get stable, accurate, repeatable Tseq(n): Repeat measurements many times (thumb rule: average over at least 30 repetitions. Be very careful!)

New issue with modern processors:
Clock speed may not be constant (turbomode, power capping, …), behavior can change with time

Modern software: Behavior can change with time… ("intelligent", adaptive software)

Example: AMD EPYC Series: Turbo-mode vs. all cores

One core: 3.2GHz. All cores: 2.20GHz

AMD EPYC PRODUCT FAMILY AND WORKLOAD AFFINITY

Model #	Cores	Threads	Base Freq. (GHz)	All Core Boost Freq. (GHz)	Max. Boost Freq. (GHz)	TDP (W)	L3 Cache (MB)	DDR Channels	Max DDR Freq. (1DPC)	2-Socket Theoretical Memory Bandwidth GB/s	PCIe®	2P/1P	Workload Affinity
7601	32	64	2.20	2.70	3.20	180	64	8	2666	341	x128	2P/1P	• DBMS and Analytics • Capacity HPC
7551 7551P	32	64	2.00	2.55	3.00	180	64	8	2666	341	x128	2P/1P 1P	• VM Dense • VDI • DBMS and Analytics • Capacity HPC
7501	32	64	2.00	2.60	3.00	155/170	64	8	2400/2666	307/341	x128	2P/1P	• VM Dense • VDI • DBMS and Analytics • Web Serving
7451	24	48	2.30	2.90	3.20	180	64	8	2666	341	x128	2P/1P	• General Purpose
7401 7401P	24	48	2.00	2.80	3.00	155/170	64	8	2400/2666	307/341	x128	2P/1P 1P	• General Purpose • GPU/FPGA Accelerated • Storage
7371	16	32	3.10	3.60	3.80 (8C)	200	64	8	2666	341	x128	2P/1P	• Frequency Optimized • EDA and HPC • Video and Gaming
7351 7351P	16	32	2.40	2.90	2.90	155/170	64	8	2400/2666	307/341	x128	2P/1P 1P	• General Purpose • GPU/FPGA Accelerated • Storage
7301	16	32	2.20	2.70	2.70	155/170	64	8	2400/2666	307/341	x128	2P/1P	• General Purpose • License Cost Optimized
7281	16	32	2.10	2.70	2.70	155/170	32	8	2400/2666	307/341	x128	2P/1P	• General Purpose • License Cost Optimized
7261	8	16	2.50	2.90	2.90	155/170	64	8	2400/2666	307/341	x128	2P/1P	• General Purpose • License Cost Optimized
7251	8	16	2.10	2.90	2.90	120	32	8	2400	307	x128	2P/1P	• License Cost Optimized

LEARN MORE:

Definition (Absolute Speed-up, theory):

Let Tseq(n) be the (worst-case) time of the best possible/best known specific, sequential algorithm Seq for P, and Tpar(n,p) the (worst-case) time of a parallel algorithm Par. The absolute speed-up of Par on p processors over Seq is

$$S_p(n) = Tseq(n)/Tpar(p,n)$$

Observation (proof follows):
Best-possible, absolute speed-up is linear in p

Goal: Obtain (linear) absolute speed-up for as large p as possible (as function of problem size n), for as many n as possible

Goal: Obtain (linear) absolute speed-up for as large p as possible (as function of problem size n), for as many n as possible

The difficult objective of parallel computing:
To develop algorithms and techniques (and interfaces and compilers) that allow us to ultimately <u>be faster than the best known sequential approaches</u>!

If this is not possible, why parallelize? Resources can be used better differently&elsewhere

For speed-up (and other complexity measures), distinguish:

- Problem P to be solved (mathematical specification)

- Some algorithm A to solve P
- Best possible (lower bound) algorithm A* for P, best known algorithm A+ for P: The complexity of P

- Implementation of A on some machine M

"Best possible" algorithm is most often not known. Lower bounds in computer science are somewhat rare and difficult to establish. Must therefore settle for "best known".

Work

- The work of a sequential algorithm Seq on input of size n is the total number of operations (integer, FLOP, memory, …; that which matters most, according to model) carried out when Seq runs to completion on n
- The work of a parallel algorithm Par on input of size n is the total number of operations carried out by all assigned processors, not including idle or waiting times
- The work required for some problem P is the work by a best possible algorithm (complexity of P)

A (parallel) algorithm that performs work proportional to a best possible (sequential) algorithm is called work optimal

Example: "Data parallel" (SIMD) computation

Complexity: $Tseq(n) = \Theta(n)$

Seq algorithm/
implementation:

```
for (i=0; i<n; i++) {
   a[i] = b[i]+c[i];
}
```

Problem: sum of two n-element vectors

Subproblem: Sum of n/p elements	Subproblem: sum of n/p elements	Subproblem: sum of n/p elements	Subproblem: sum of n/p elements

$Tpar(p,n) = Tseq(n)/p$

$S_p(n) = n/(n/p) = p$

Best possible parallelization:
sequential work divided evenly across
p processors

Example: “Data parallel” (SIMD) computation

Seq algorithm/
implementation:

```
for (i=0; i<n; i++) {
   a[i] = b[i]+c[i];
}
```

Problem: sum of two n-element vectors

Subproblem: Sum of n/p elements

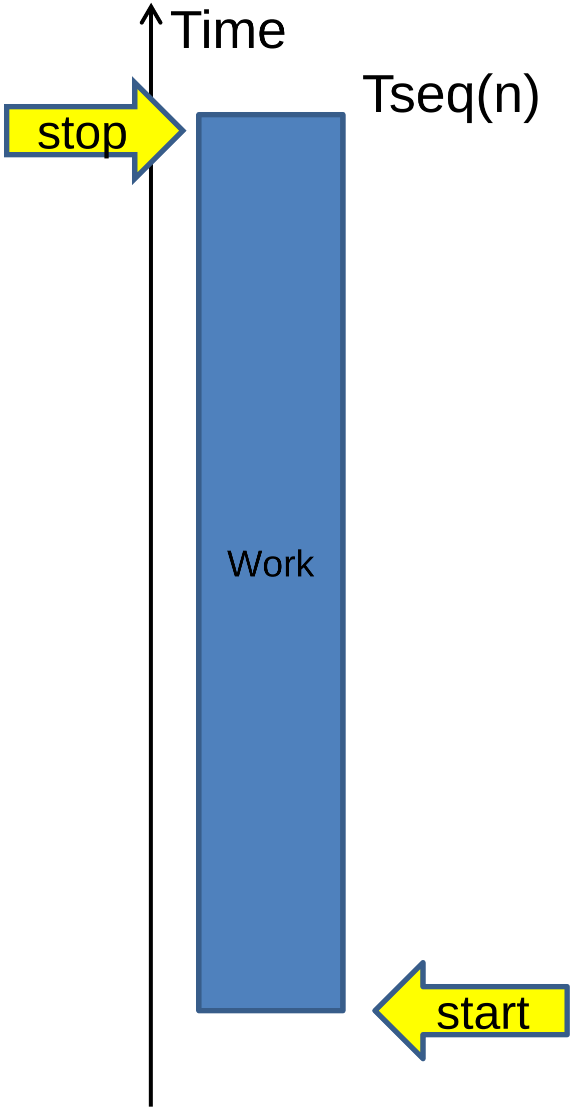
Subproblem: sum of n/p elements

Subproblem: sum of n/p elements

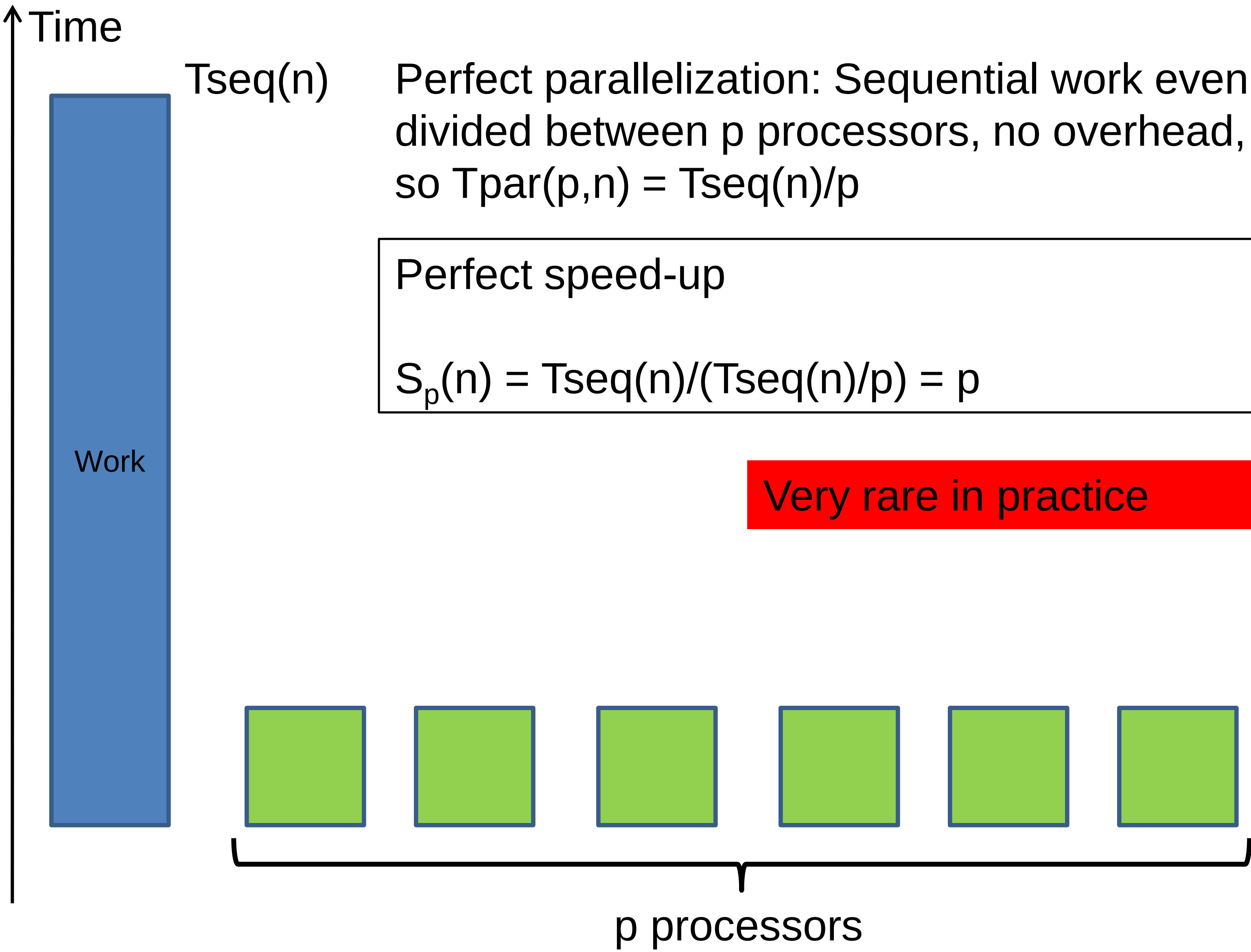
Subproblem: sum of n/p elements

- Perfect speed-up
- “Embarrassingly parallel”
- “Pleasantly parallel”

Tpar(p,n) = c(n/p) for constant c≥1



The <u>work</u>, measured in instructions and/or time that has to be carried out for problem of size n (worst-case)



Perfect parallelization: Sequential work evenly divided between p processors, no overhead, so Tpar(p,n) = Tseq(n)/p

Perfect speed-up

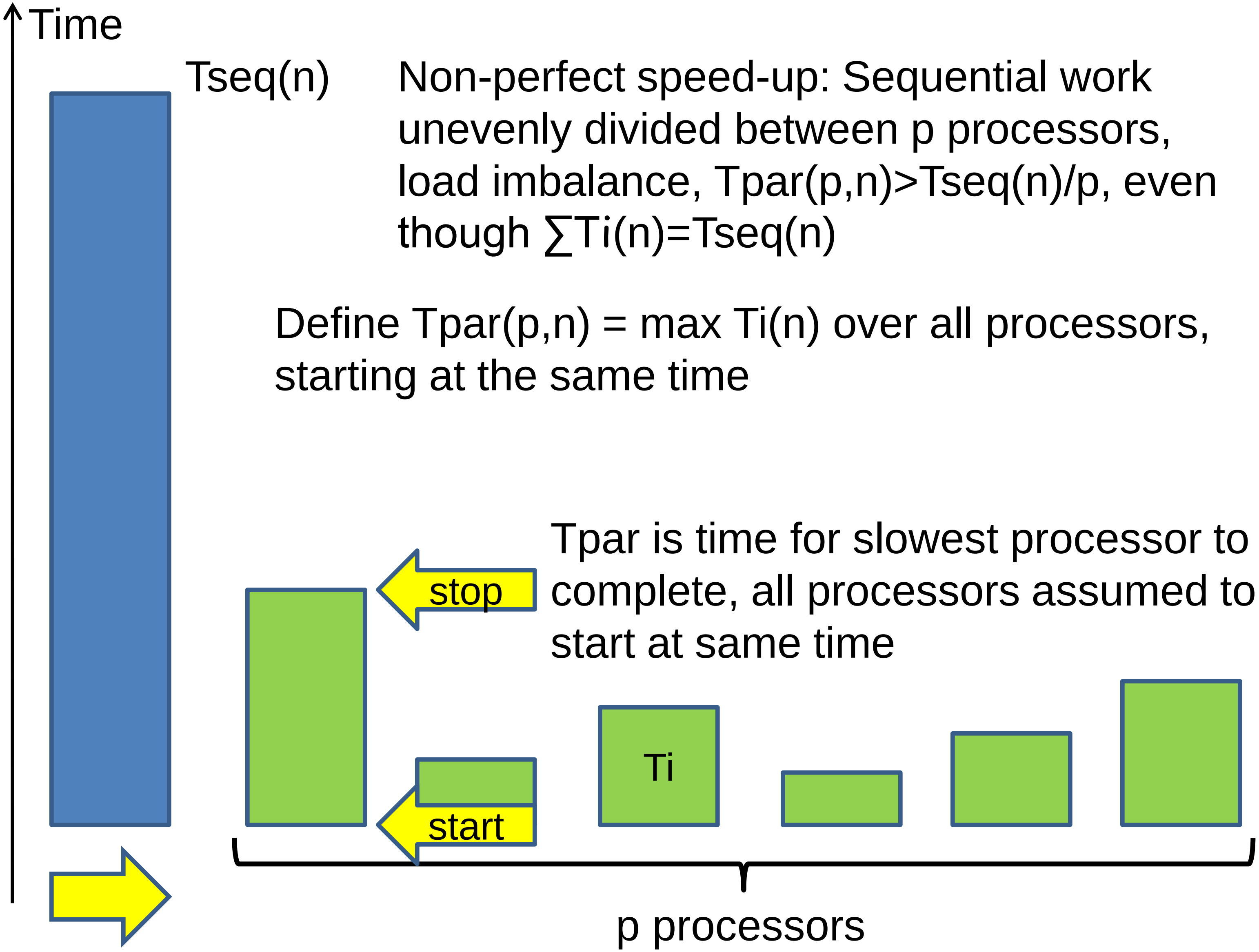
$S_p(n) = Tseq(n)/(Tseq(n)/p) = p$

Very rare in practice

Non-perfect speed-up: Sequential work unevenly divided between p processors, load imbalance, Tpar(p,n)>Tseq(n)/p, even though ∑Ti(n)=Tseq(n)

Define Tpar(p,n) = max Ti(n) over all processors, starting at the same time

Tpar is time for slowest processor to complete, all processors assumed to start at same time

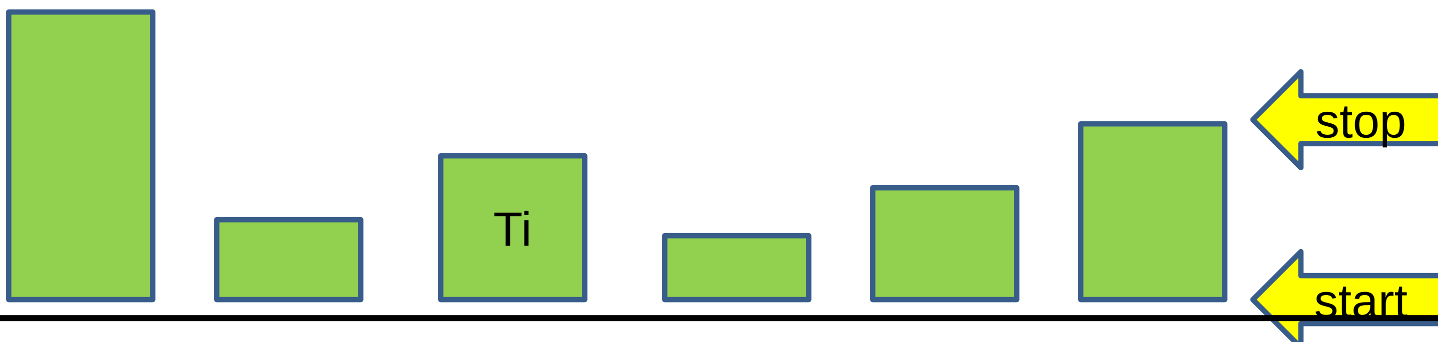


Measuring parallel time, Tpar(p,n)

Time of slowest processor-core to finish, assuming all processors start at the same time (recall definition of parallel computing: dedicated resources)

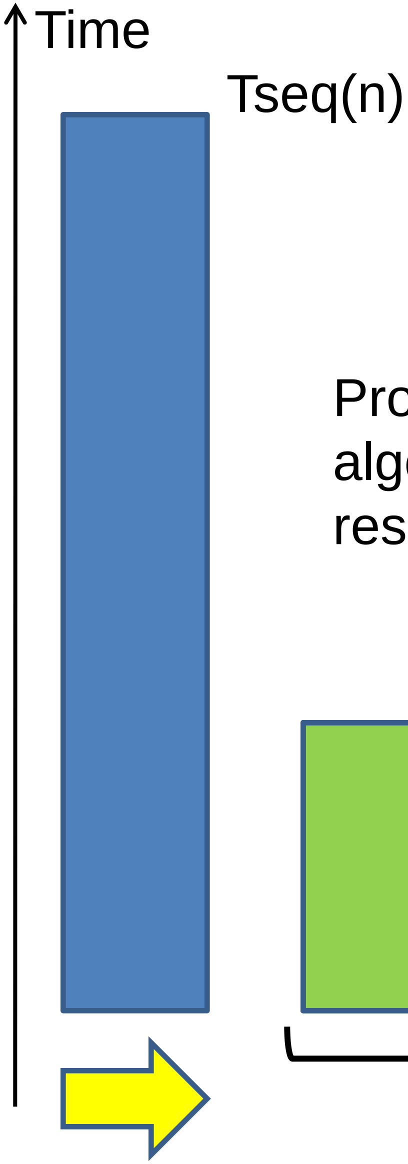
“barrier” sync.

```
for (number of repetitions) {
  // synchronize processors, all start at same time
```



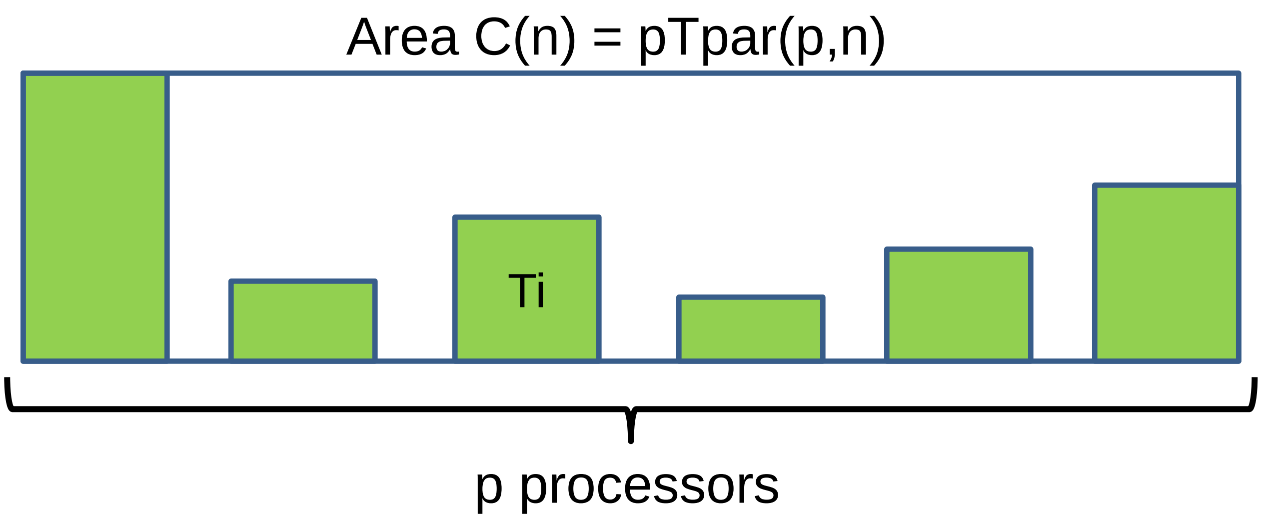
“barrier” sync.

```
  Ti = stop-start
  Tpar(p,n) = max_{0≤i<p}Ti
}
// Do statistics
```



Wpar(n) = ∑Ti(n) is the <u>work of the parallel algorithm</u>, total number of instructions performed by the p processors

Product C(n) = pTpar(p,n) is <u>the cost of the parallel</u> algorithm, total time in which the p processors are reserved (: have to be paid for)

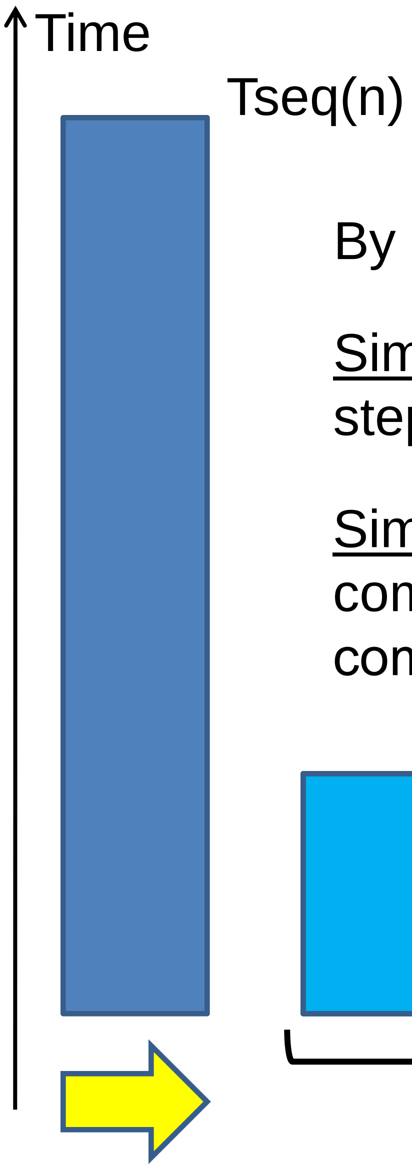


"Theorem:"
Linear (perfect) speed-up $S_p(n) = cp$ is best possible and cannot be exceeded (for some constant c, $0<c\leq 1$).

Advantage of a theoretical model: Using the PRAM, a technical proof with all details can be given

"Proof":
A sequential algorithm can be constructed from a parallel algorithm by simulating the parallel algorithm on a single processor. The instructions of the p processors have to be carried out in some correct order on the sequential processor. The time for the simulation is $Tsim(n) \leq pTpar(p,n)$.
Assume $S_p(n) > p$ for some n. Now $Tseq(n)/Tpar(p,n) > p$ implies $Tseq(n) > pTpar(p,n) \geq Tsim(n)$, and contradicts that Tseq(n) was best possible/known time.

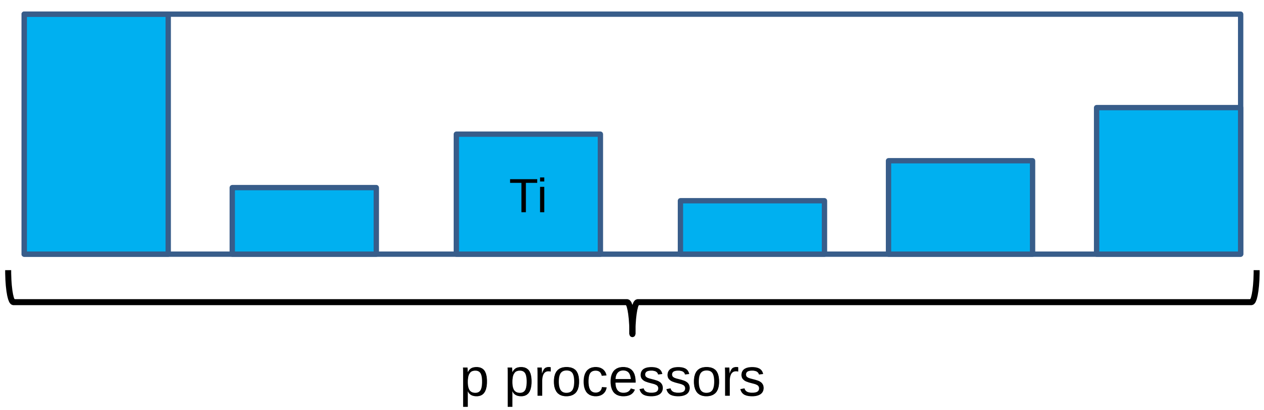
Reminder: Speed-up is calculated (measured) relative to "best" sequential algorithm (implementation)

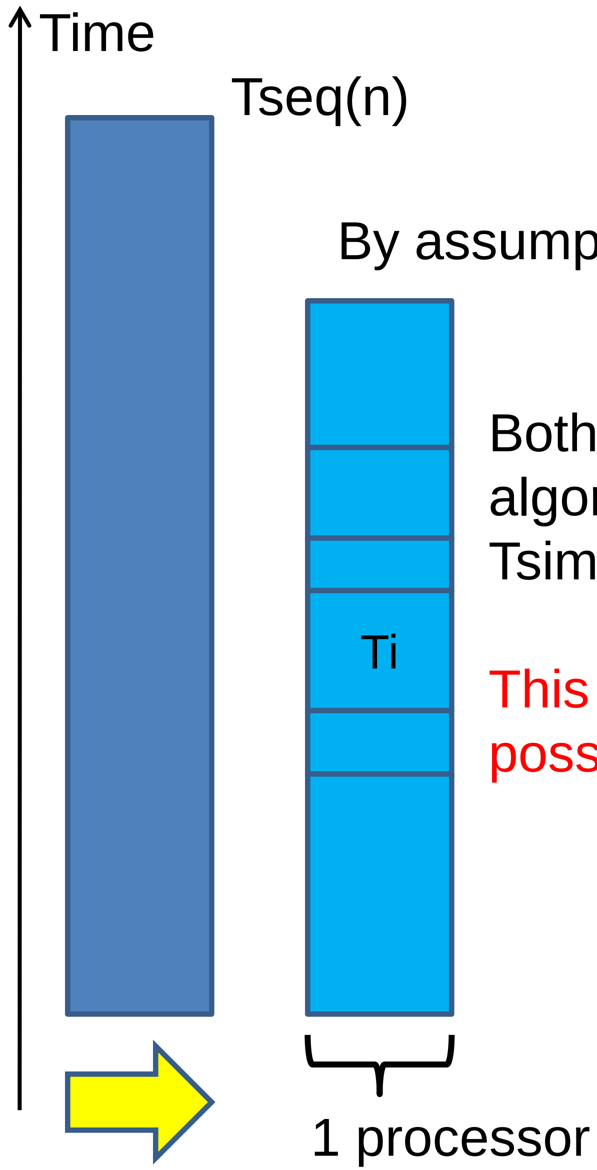


By assumption C(n) = pTpar(p,n) < Tseq(n)

<u>Simulation A</u>: one step of P1, one step of P2, …, one step of P(p-1), one step of P1, …, for C(n) iterations

<u>Simulation B</u>: steps of P1 until communication/synchronization, steps of P2 until communication/synchronization, …

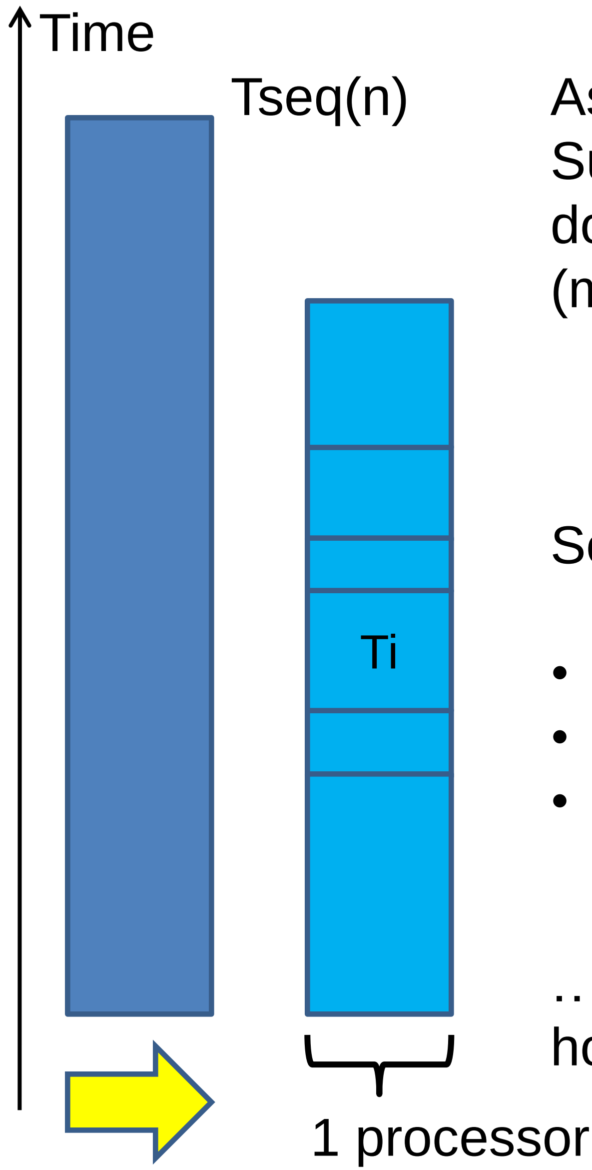




By assumption C(n) = pTpar(p,n) < Tseq(n)

Both simulations yield a new, sequential algorithm Tsim(n) with Tsim(n)≤pTpar(p,n)<Tseq(n)

This contradicts that Tseq(n) was time of best possible/best known sequential algorithm



Aside:
Such simulations are actually sometimes done, and can be very useful to understand (model) and debug parallel algorithms

Some simulation tools:

- SimGrid (INRIA)
- LogPOPSim (Hoefler et al.)
- …

…Or when running parallel program at home on one (or a few) processors

Simulation construction shows that the total parallel work must be at least as large as the sequential work Tseq, otherwise, better sequential algorithm can be constructed.

Crucial assumptions: Sequential simulation is possible (enough memory to hold problem and state of parallel processors), sequential memory behaves as parallel memory, …

This is NOT TRUE for real systems and real problems

Lesson:
Parallelism offers only “modest potential”, speed-up cannot be more than p on p processors

Given the simulation, the definitions of linear and perfect speed-up can be strengthened to:

Definitions:
A parallel algorithm Par(p,n) has linear absolute speed-up relative to a best-known sequential algorithm Seq(n) if

$S_p(n) = \Theta(p)$

A parallel algorithm Par(p,n) has "perfect" absolute speed-up relative to a best-known sequential algorithm Seq(n) if

$S_p(n) \approx p$

"Perfect" speed-up is the rare case where the actual (measured or theoretically proven) speed-up is actuall close to p (constant close to 1)

The product C(n) = pTpar(p,n) is the <u>cost</u> of the parallel algorithm: Total time in which the p processors are occupied

Definition:
Parallel algorithm is called <u>cost-optimal</u> if C(n) = O(Tseq(n)). A cost-optimal algorithm has linear (perhaps perfect) speed-up

Wpar(p,n) = ∑Ti(n) is the parallel <u>work</u> of the parallel algorithm: total number of instructions performed by p processors

Definition:
Parallel algorithm is called <u>work-optimal</u> if Wpar(p,n) = O(Tseq(n)). A work-optimal algorithm has potential for linear speed-up (for some number of processors)

Examples:

Let Tseq(n) = O(n) for some (best known) algorithm Seq.

Any parallel algorithm with Tpar(p,n) = O(n/p) is cost-optimal, since for some constant c, p O(n/p) ≤ p (c(n/p)) = c n = O(n).

Parallel algorithms with Tpar(p,n) = O(n/√p) or Tpar(p,n) = O(n/(p/log p)) = O((n log p)/p) are not cost-optimal.

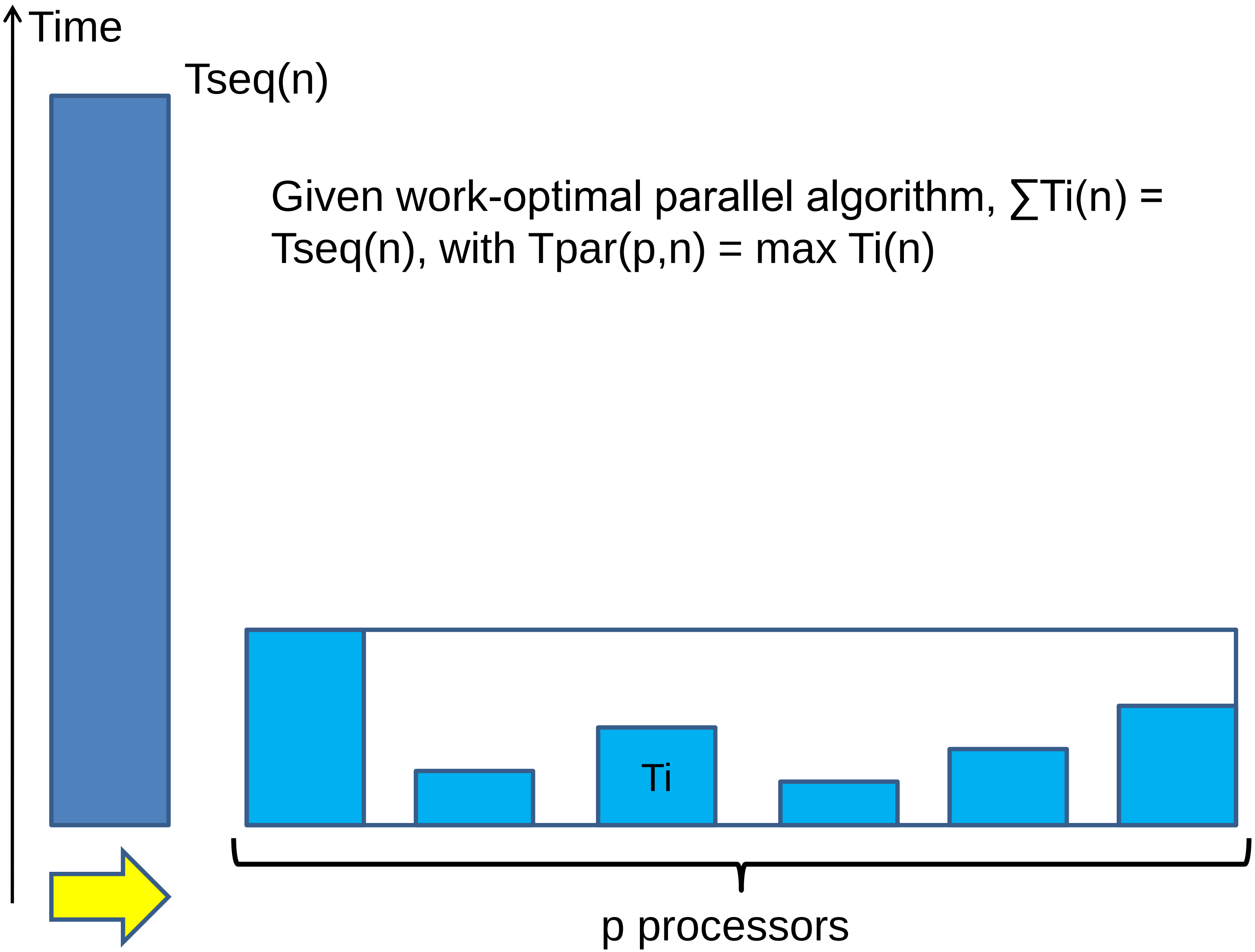
We have p c(n/√p) = c √p n which not O(n) since √p is not constant (bounded). Likewise, p c (n log p)/p = c n log p is not O(n).
Such algorithms cannot have linear speed-up.

Proof (linear-speed up of cost-optimal algorithm):

Given cost-optimal parallel algorithm with pTpar(p,n) = cTseq(n) = O(Tseq(n)). This implies Tpar(p,n) = cTseq(n)/p, so

$$S_p(n) = Tseq(n)/Tpar(p,n) = p/c$$

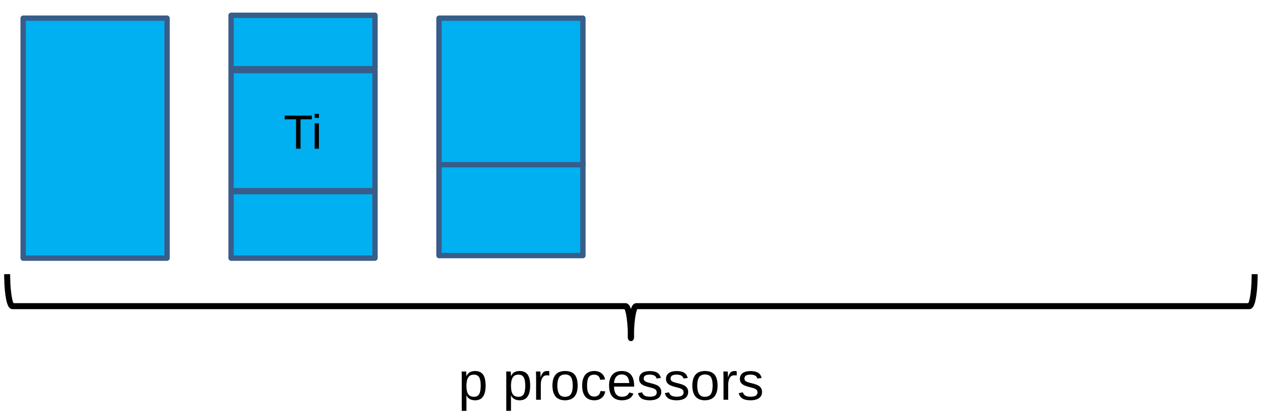
The constant factor c captures the load imbalance and overheads (see later) of the parallel algorithm relative to best sequential algorithm. The smaller c, the closer the speed-up to perfect



Given work-optimal parallel algorithm, $\sum Ti(n) = Tseq(n)$, with $Tpar(p,n) = \max Ti(n)$

Given work-optimal parallel algorithm, ∑Ti(n) = Tseq(n), with Tpar(p,n) = max Ti(n)

Execute on smaller number of processors p’, such that ∑Ti(n) = p’Tpar(p’,n) = O(Tseq(n))



Proof idea (work-optimal algorithm can have linear speed-up):

1. Work-optimal algorithm
2. Schedule work-items Ti(n) on p' processors, such that p'Tpar(p',n) = O(Tseq(n))
3. With this number of processors, algorithm is cost-optimal
4. Cost-optimal algorithms have linear speed-up

The scheduling in Step 2 is possible in principle, but may not be trivial in concrete terms

Parallel algorithms' design goal:
Work-optimal parallel algorithm with as small Tpar(p,n) as possible (and therefore large parallelism: many processors can be utilized)

Example: CRCW PRAM Maximum Finding algorithm

```
par (0<=i<n) b[i] = true;      // a[i] could be
par (0<=i<n, 0<=j<n)
  if (a[i]<a[j]) b[i] = false; // a[i] is not
par (0<=i<n) if (b[i]) x = a[i];
```

$O(n^2)$ operations (work), but sequential maximum finding requires only $O(n)$ operations

Not work-optimal

Speed-up with perfect parallelization

$$S_p(n) = O(n)/O(n^2/p) = O(p/n)$$

Bad!

Only small (linear, for fixed n) speed-up, and decreasing with n

Example: Another not work-optimal algorithm

Given DumbSort(n) with $T(n) = O(n^2)$ that can be perfectly parallelized, $Tpar(p,n) = O(n^2/p)$

Well-known that $Tseq(n) = \Theta(n \log n)$, many algorithms and good implementations, so

$$S_p(n) = O(n \log n)/O(n^2/p) = O(p (\log n)/n)$$

Linear speed-up for fixed n but not independent of n (decreasing)

Not work-optimal algorithm: Speed-up decreases with n

Break-even:
How many processors are needed for parallel algorithm to be faster than sequential algorithm?

- PRAM Maximum Finding: Tpar(p,n) < Tseq(n) ⇔ $n^2/p < n$ ⇔ $p > n$
- DumbSort: Tpar(p,n) < Tseq(n) ⇔ $n^2/p < n \log n$ ⇔ $n/p < \log n$ ⇔ $p > n/\log n$

Bad! (Almost) as many processors needed as problem size n to be as fast as sequential algorithm.

Lesson:
It does not make sense to parallelize an inferior algorithm (although sometimes much easier). Almost never…

But parallelizing an efficient, best known sequential algorithm can be difficult.

Efficient, sequential algorithm often has:

- No redundant work (because efficient)
- Tight dependencies, forcing things to be done in a specific, sequential order: One thing (and not many) after the other

Lesson from much hard work in (e.g., PRAM) theory and practice: Work/cost-optimal parallel solution of a given problem often requires a new algorithmic idea!

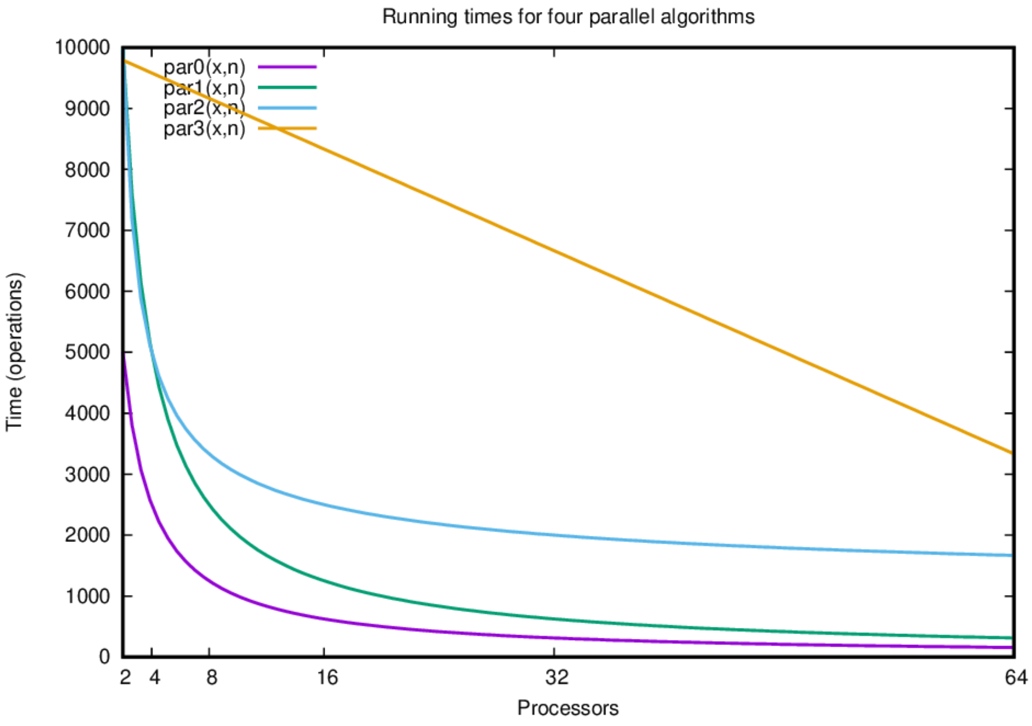
Parallel computing is a creative endeavor!

Lesson from much hard work in (e.g., PRAM) theory and practice: Work/cost-optimal parallel solution of a given problem often requires a new algorithmic idea!

But:
Many sequential algorithms often have a lot of potential for easy parallelization (loops, independent functions, …). Why not exploit this?

Also:
Non-work optimal algorithms can sometimes be useful, as subroutine

Example: Time and speed-up for four linear work algorithms

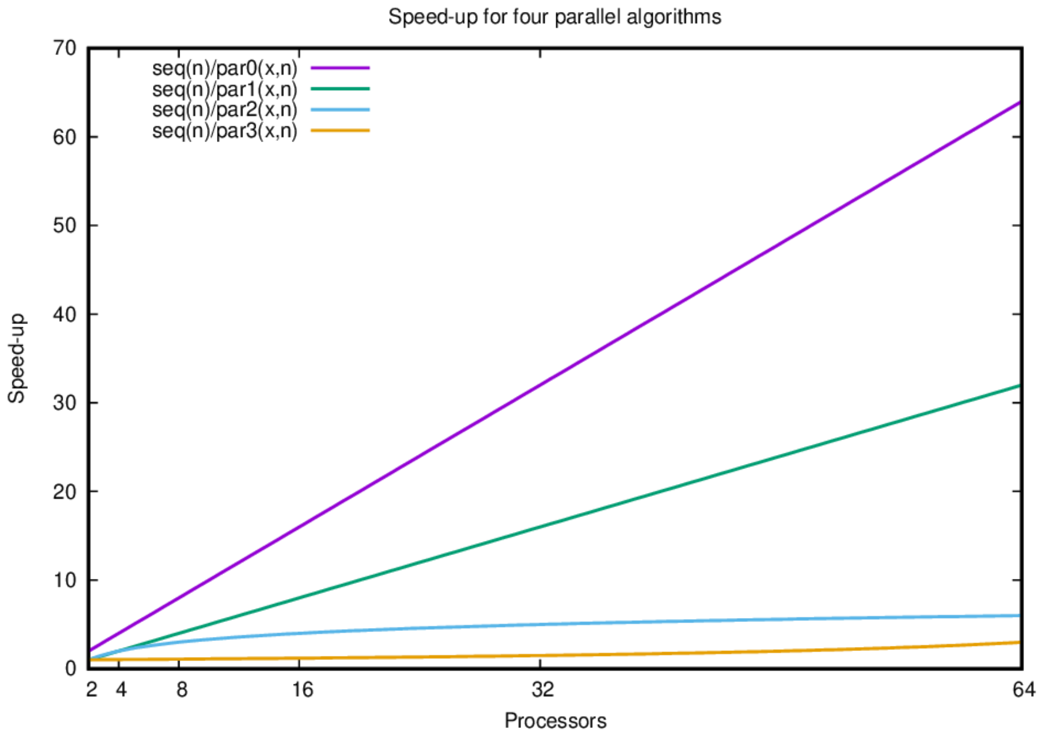


Tseq(n) = O(n)
Fix n=10000, ignore (normalize) constants

What are the parallel running times?
Which ones have linear speed-up?

Tpar0(p,n), Tpar1(p,n), Tpar2(p,n) look similar

Example: Time and speed-up for four linear work algorithms



Tpar0(p,n) = Tseq(n)/p

Tpar1(p,n) = Tseq(n)/p/2

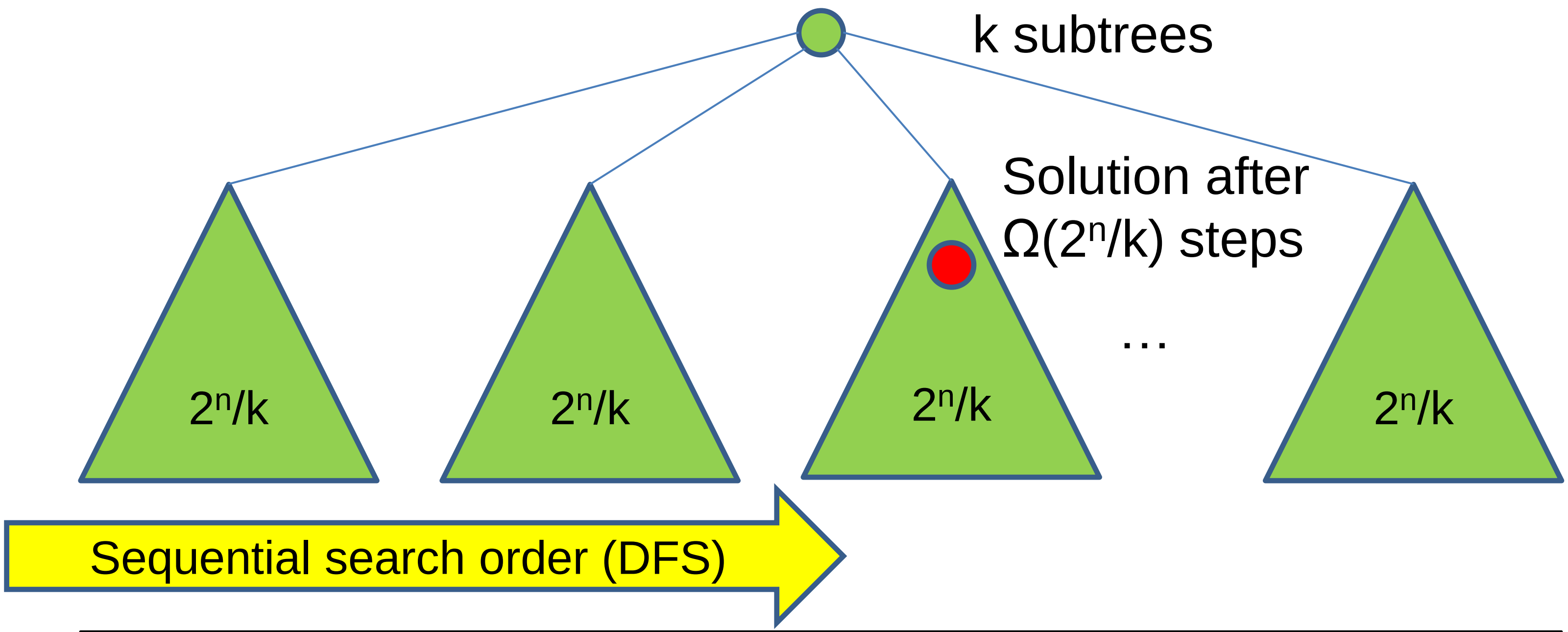
Tpar2(p,n) = Tseq(n)/$\log_2$p

Tpar3(p,n) = Tseq(n)(1-p/96)

Careful with looking at time alone

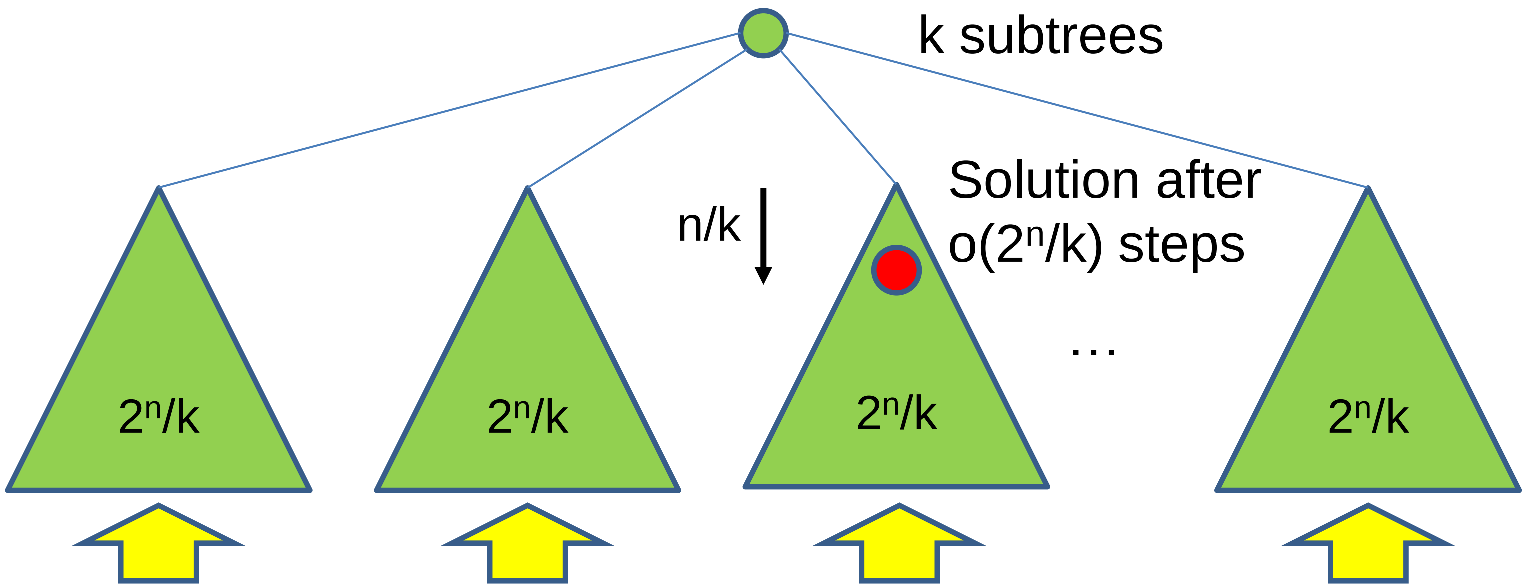
Disproof (I): Is super-linear speed-up possible?

Combinatorial problems are often solved by clever tree-search



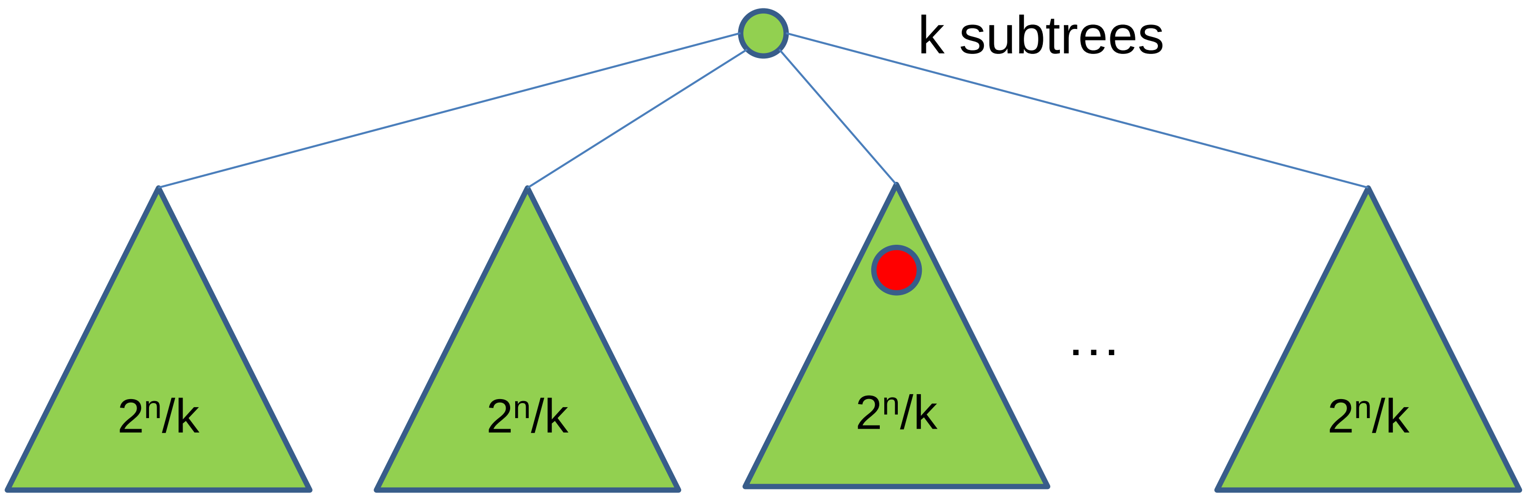
Marijn J. H. Heule, Oliver Kullmann: The science of brute force. Commun. ACM 60(8): 70-79 (2017)

Combinatorial problems are often solved by clever tree-search



Parallelization: k trees in parallel on k processors. Now solution in $o(2^n/k)$ steps, say n/k. Lucky processor finds solution fast

Combinatorial problems are often solved by clever tree-search



Speed-up is c $2^n/(n/k) = k\ 2^n/n \gg k$ for k processors and large n

But this does not contradict that linear/perfect speed-up is best possible.

The parallel and the sequential algorithms are just different. In the example, DFS is not the best search strategy, the parallel algorithm does a mix of BFS and DFS, which might be better (and hard to know in advance).

Reasons for “algorithmic” super-linear speed-up:

- Different algorithms
- Randomization, luck
- Non-determinism

Other factors can also lead to super-linear speed-up. See later

Absolute vs. relative speed-up

Definition (Relative speed-up): The ratio

$$SRel_p(n) = Tpar(1,n)/Tpar(p,n)$$

is the relative speed-up of algorithm Par. Relative speed-up expresses how well Par utilizes p processors (scalability)

Relative speed-up not to be confused with absolute speed-up. Absolute speed-up expresses how much can be gained over the best (known/possible) sequential implementation by parallelization.

Absolute speed-up is what ultimately matters

Beware:
Literature (research papers and books) is not always clear about the distinction between Absolute and Relative speed-up.

It is easier to achieve and document good relative speed-up.

Reporting speed-up relative to an inferior, sequential implementation is misleading and technically incorrect (goal: achieve speed-up over a best known algorithm/implementation)

Goal:
Obtain (linear) absolute speed-up for as large p as possible (as function of problem size n), for as many n as possible

Definition:
$T_\infty(n)$: The smallest possible running time of parallel algorithm Par given arbitrarily many processors. Per definition $T_\infty(n) \leq Tpar(p,n)$ for all p. Relative speed-up is limited by

$$SRel_p(n) = Tpar(1,n)/Tpar(p,n) \leq Tpar(1,n)/T_\infty(n)$$

Definition:
The ratio $Tpar(1,n)/T\infty(n)$ is called the parallelism of the parallel algorithm Par

The parallelism is the largest number of processors that can be employed and still give linear, relative speed-up: Assume $Tpar(1,n)/T_\infty(n)<p'$, the equation above tells that $SRel_p(n) < p'$

Statements on speed-up, e.g.,

1. $S_p(n) = c_1 p$ for some $c_1 < 1$
2. $S_p(n) = c_2 \sqrt{p}$ for some $c_2 < 1$

etc. implicitly assumes some upper bound on the number of processors for which this holds. Often, this upper limit is not stated, but there is always a point for which it does not make sense to use additional processors.

The parallelism $Tpar(1,n)/T\infty(n)$ is one such limit

Example: CRCW PRAM Maximum Finding algorithm

```
par (0<=i<n) b[i] = true;       // a[i] could be
par (0<=i<n, 0<=j<n)
  if (a[i]<a[j]) b[i] = false; // a[i] is not
par (0<=i<n) if (b[i]) x = a[i];
```

$O(n^2)$ operations (work), but sequential maximum finding requires only $O(n)$ operations

$SRel_p(n) = O(n^2)/O(n^2/p) = O(p)$
Parallelism: $O(n^2)/O(1) = n^2$

This (terrible) parallel algorithm has linear relative speed-up for p up to n^2 processors (!). And great parallelism.

Example: CRCW PRAM Maximum Finding algorithm

```
par (0<=i<n) b[i] = true;       // a[i] could be
par (0<=i<n, 0<=j<n)
  if (a[i]<a[j]) b[i] = false; // a[i] is not
par (0<=i<n) if (b[i]) x = a[i];
```

This (terrible) algorithm has linear relative speed-up for p up to n^2 processors

Nevertheless: Useful as a building block

<u>Theorem</u>: There exist a work-optimal CRCW PRAM algorithm that runs in O(log log n) steps requiring O(n) parallel work

Advanced material. And last fact about PRAM in this lecture

An algorithm has good scalability and relative speed-up if Tpar(1,n)/Tpar(p,n) = $\Theta(p)$

Relative speed-up

Example:
Someone reports for algorithm Par that $0.1p \leq$ Tpar(1,n)/Tpar(p,n) $\leq 0.5p$ is reported. Sounds good!

But what if Tpar(1,n) = 100Tseq(n)?
Or Tseq(n) = O(n) but Tpar(p,n) = O((n log n)/p + log n)?

Even for work-optimal Tpar(1,n) = 100Tseq(n) = O(Tseq(n)) it would take at least 200 processors to break even with the sequential algorithm with the reported relative speed-up

Constants, as always, do matter (for the practitioner)

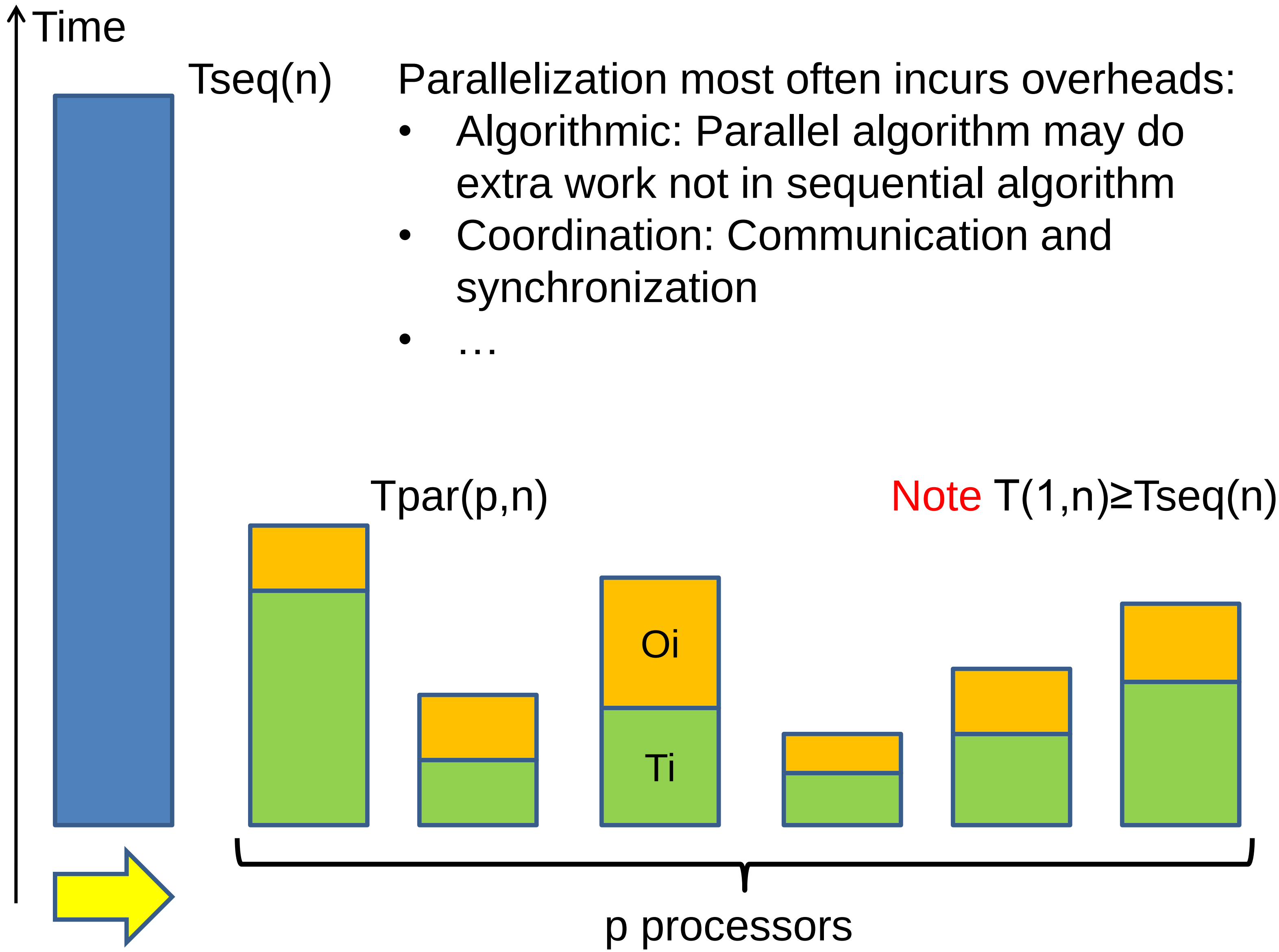
Again: Work-optimality is a strong property

Work-optimality property:

For work-optimal algorithms, absolute and relative speed-up coincide (asymptotically), since Tpar(1,n) = O(Tseq(n))

Parallelization most often incurs overheads:

- Algorithmic: Parallel algorithm may do extra work not in sequential algorithm
- Coordination: Communication and synchronization
- …



Parallelization overheads

Parallel overhead is the work that does not have to be done by a sequential algorithm

- Communication: Exchanging data, keeping data consistent
- Synchronization: Ensuring that processors have reached the same point in the computation (typically SPMD programs)
- Algorithmic: Extra or redundant computations

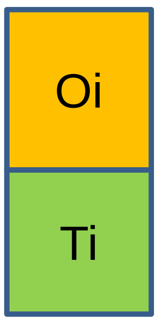
(Communication) Overheads for processor i sometimes modeled as

$$Toverhead(p,n_i) = \alpha(p) + \beta n_i$$

where $\alpha(p)$ is the latency (dependent on p), and β the cost per data item n_i that needs to be communicated by processor i. For synchronization operations, $n_i = 0$

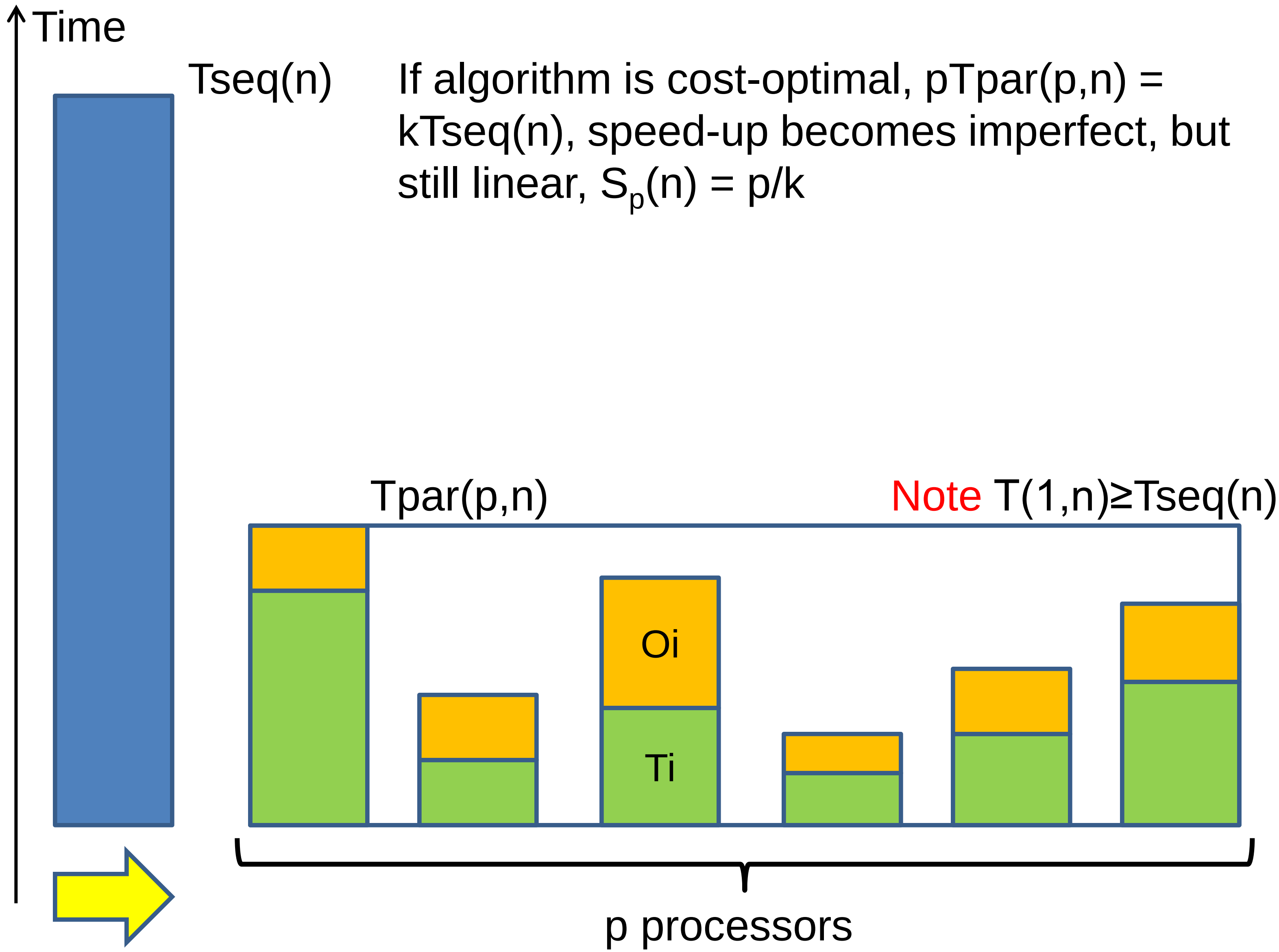
Overheads are counted as part of the parallel work (idle time is not counted, or time where processors are doing something else)

$\text{Wpar}(p,n) = \sum_{0 \le i < p} \text{Ti}(n) + \text{Oi}(n)$

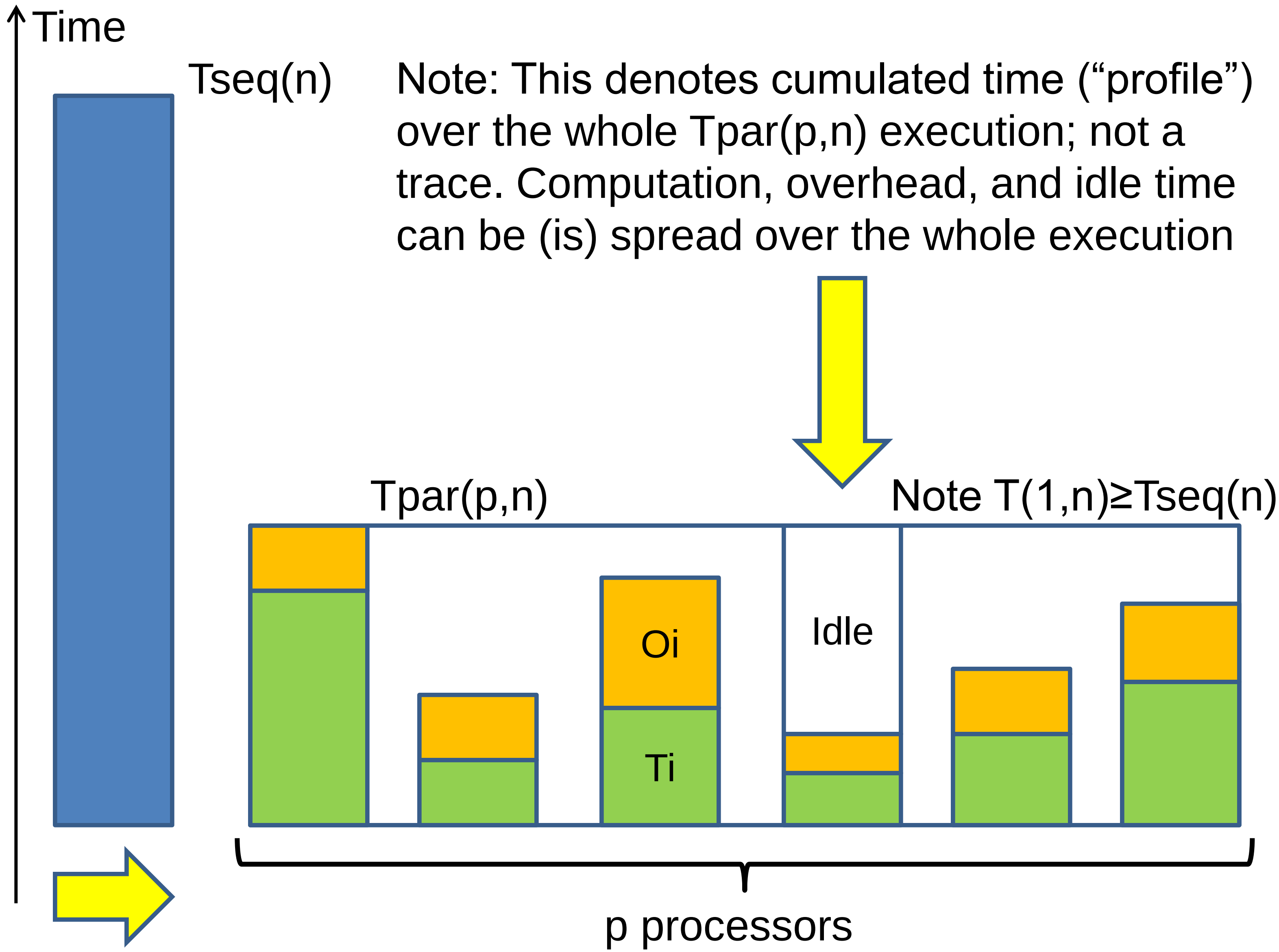


Parallel algorithm can still be work/cost-optimal if overheads are not too large, that is Wpar(p,n) = O(Tseq(n))

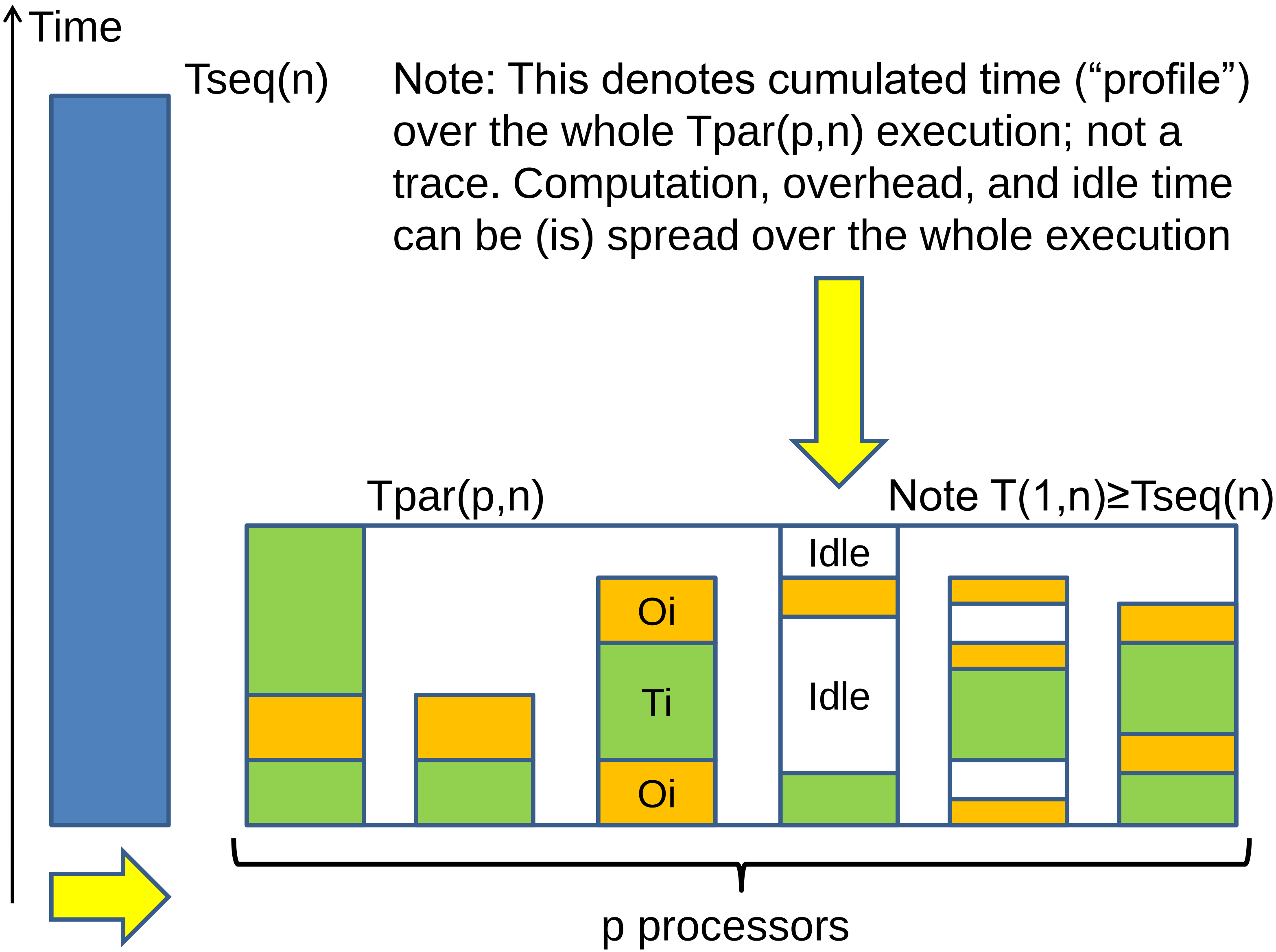
If algorithm is cost-optimal, $pTpar(p,n) = kTseq(n)$, speed-up becomes imperfect, but still linear, $S_p(n) = p/k$



Note: This denotes cumulated time (“profile”) over the whole Tpar(p,n) execution; not a trace. Computation, overhead, and idle time can be (is) spread over the whole execution

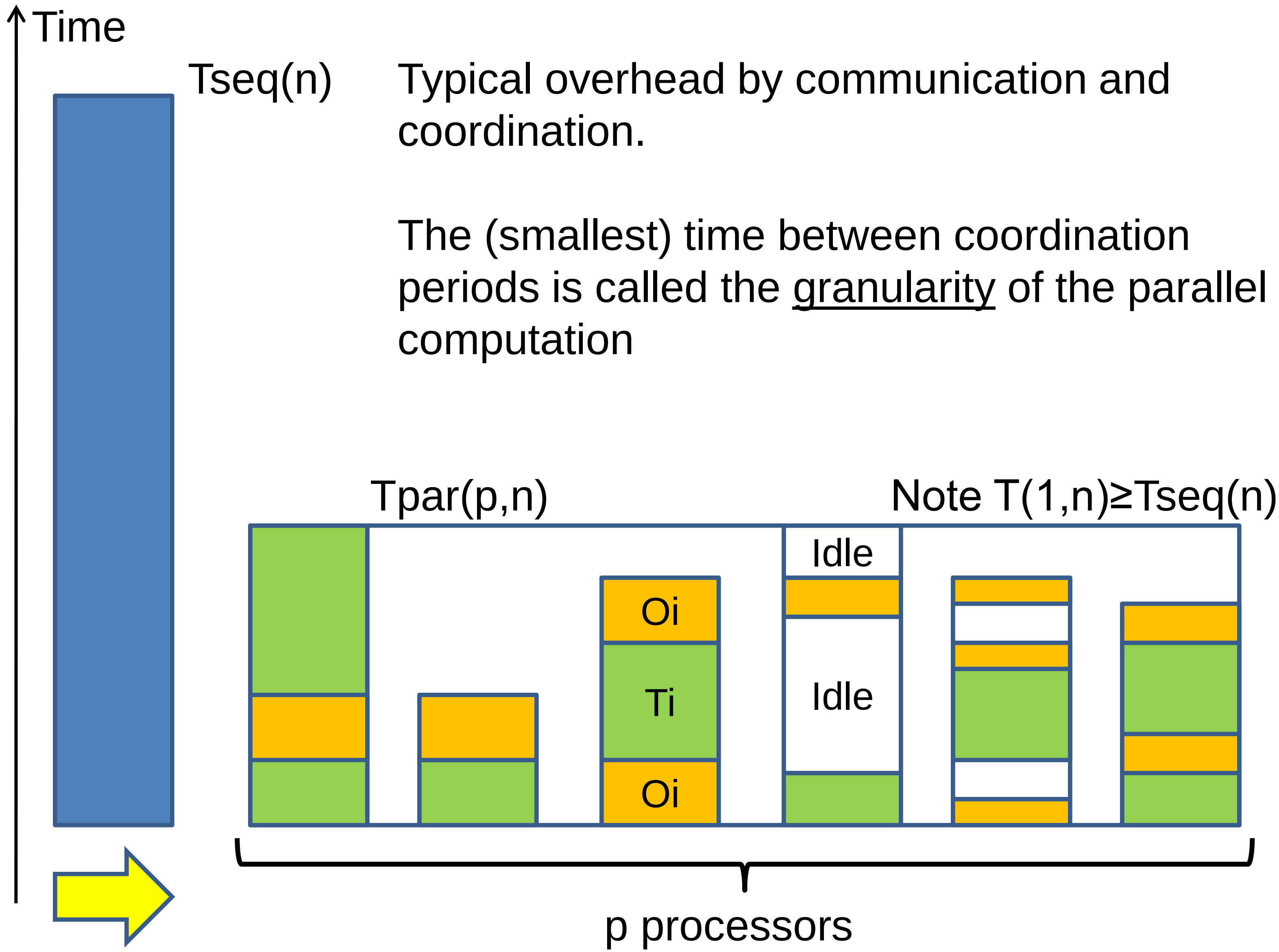


Note: This denotes cumulated time (“profile”) over the whole Tpar(p,n) execution; not a trace. Computation, overhead, and idle time can be (is) spread over the whole execution



Typical overhead by communication and coordination.

The (smallest) time between coordination periods is called the granularity of the parallel computation

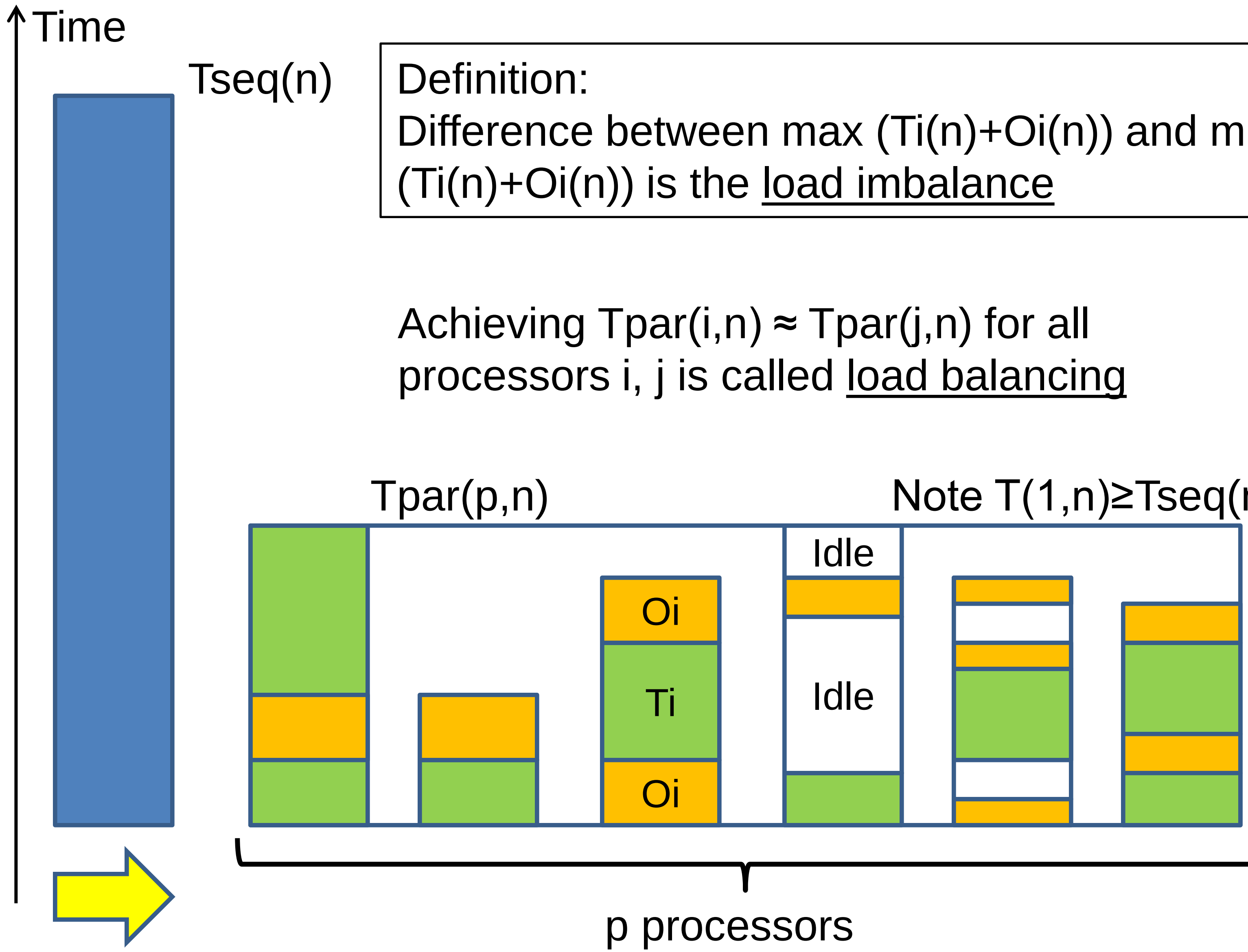


(Loose) Definition: Granularity of parallel computations:

- "Coarse-grained" parallel computation/algorithm: Time/number of instructions between coordination intervals (synchronization operations, communication operations) is large (relative to total time or work)
- "Fine-grained" parallel computation/algorithm: Time/number of instructions between… is small

Coarse-grained computation means less frequent coordination (with possibly larger data), potential for "hiding" coordination behind computation (: doing computation concurrently with communication)

Fine-grained computation requires more efficient coordination, otherwise coordination may dominate, algorithm could become non work-optimal

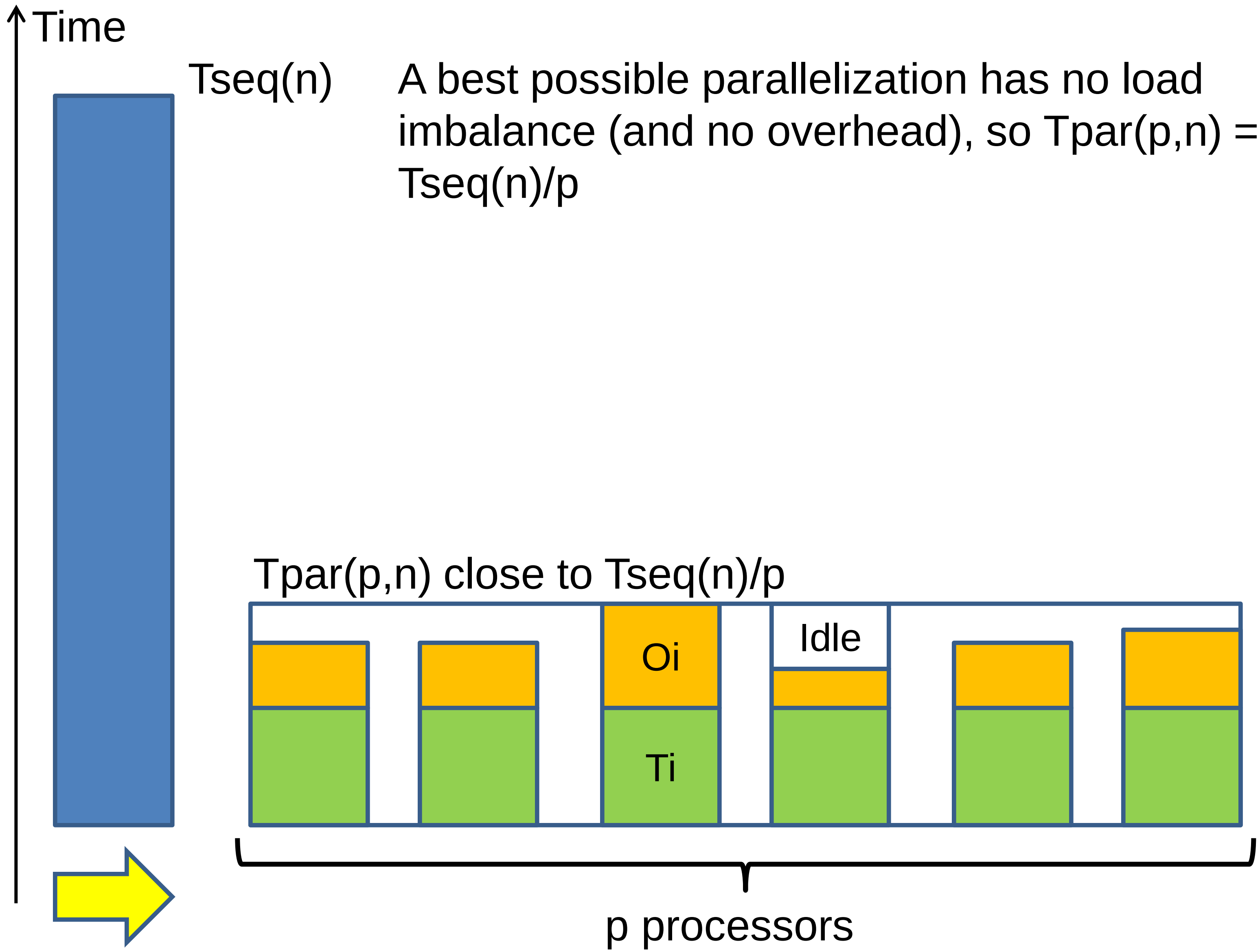


Definition:
Difference between max (Ti(n)+Oi(n)) and min (Ti(n)+Oi(n)) is the <u>load imbalance</u>

Achieving Tpar(i,n) ≈ Tpar(j,n) for all processors i, j is called <u>load balancing</u>

Note T(1,n)≥Tseq(n)

A best possible parallelization has no load imbalance (and no overhead), so Tpar(p,n) = Tseq(n)/p



Load balancing: Achieving for all processors, i, j, an even amount of work, Tpar(i,n) ≈ Tpar(j,n)

- Static, oblivious: Load balance achieved by splitting the problem into p pieces, regardless of the input (except its size n)
- Static, problem dependent, adaptive: Load balance achieved by splitting the problem into p pieces, using the (structure of) the input
- Dynamic: Load balance achieved by dynamically (during program execution) readjusting the work assigned to processors. Entails overheads (example: work stealing, see later)

Parallelizing sequential algorithm Seq

Perfectly parallelizable (static, oblivious load balancing): Tpar(p,n) = O(Tseq(n)/p)

Seq may have parts that cannot (easily) be parallelized, some fraction s(n), such that Tseq(n) = s(n)Tseq(n) + (1-s(n))Tseq(n)

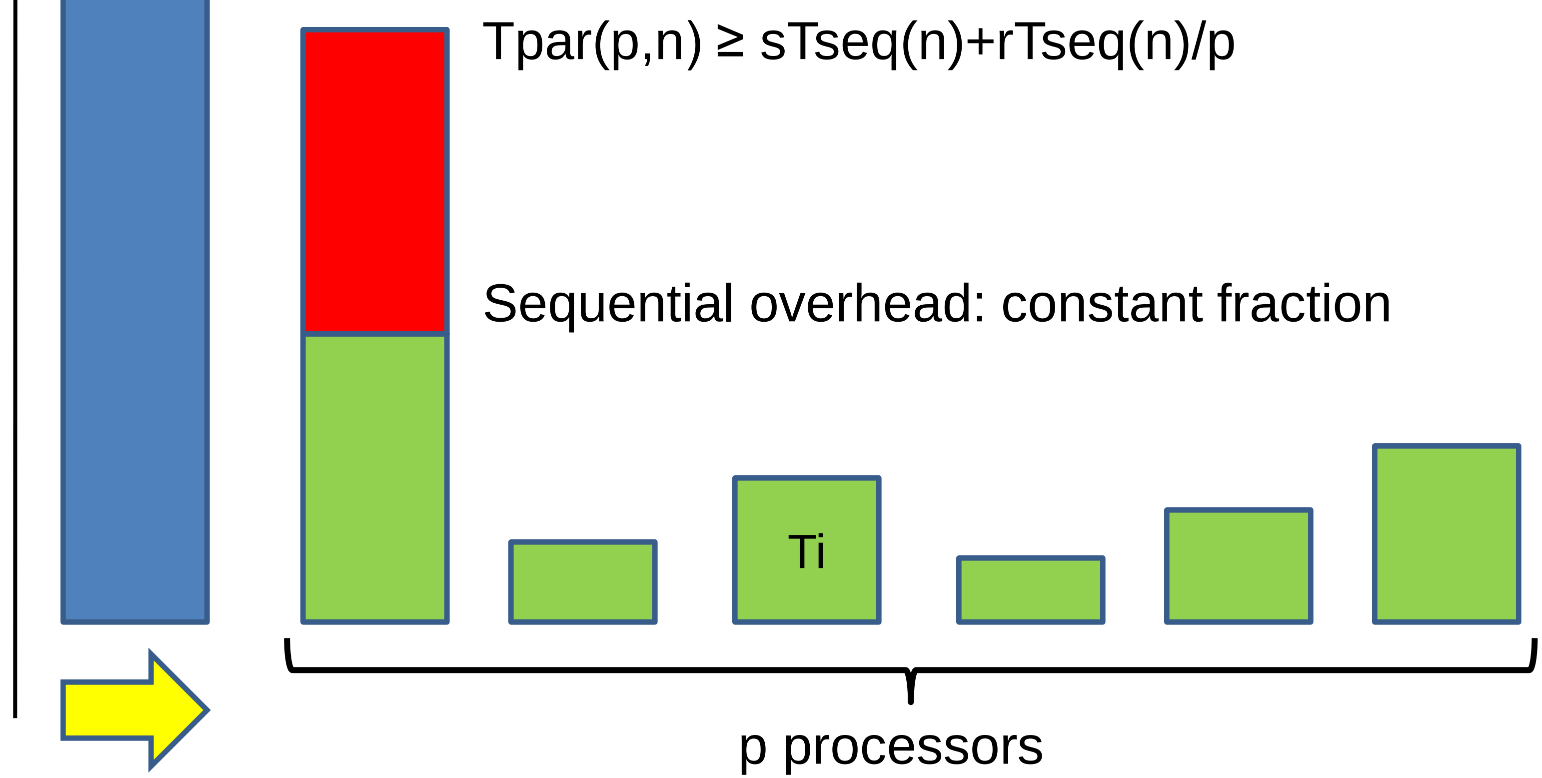
Part that could be parallelized

and Tpar(p,n) = s(n)Tseq(n) + (1-s(n))/p Tseq(n)

Time

Tseq(n) = (s+r)Tseq(n)

Maximum possible speed-up becomes severely limited if sequential part is a <u>constant fraction</u> of the total work (regardless of n)



<u>Amdahl's Law (parallel version)</u>:
Let a program Seq contain a fraction r that can be "perfectly" parallelized, and a fraction s=(1-r) that is "purely sequential", i.e., cannot be parallelized at all (s and r independent of n). The maximum achievable speed-up is 1/s, independently of n

Proof:

- Tseq(n) = (s+r)Tseq(n)
- Tpar(p,n) = sTseq(n) + rTseq(n)/p

$S_p(n)$ = Tseq(n)/(sTseq(n)+rTseq(n)/p)
= 1/(s+r/p) —> 1/s, for p —> ∞

G. Amdahl: Validity of the single processor approach to achieving large scale computing capabilities. AFIPS Spring Joint Conf., 483-485, 1967

Typical victims of Amdahl‘s law:

- Sequential input/output could be a constant fraction
- Sequential initialization of global data structures
- Sequential processing of „hard-to-parallelize“ parts of algorithm, e.g., shared data structures
- Everything that takes O(n) for input size n, and work O(n)…

Amdahl‘s law limits (kills!) speed-up in such cases, if they are a constant fraction of total time, independent of problem size

The hard work (alternative definition of parallel computing):
Find ways to avoid constant-fraction non-parallelizable work

Example:

1. Processor 0: Read input, some precomputation
2. Split problem into p parts (of size ≈n/p), send part i to processor i

9n

3. All processors i: Solve part i
4. All processors i: Send partial solution back to processor 0

10n

Amdahl: s=0.1, SU at most 10

Typical Amdahl, sequential bottleneck: Constant sequential fraction (3 out of 4 steps) limits speed-up)

When interested in parallel aspects, input-output and problem splitting is often explicitly not measured!

Example:

```
// Sequential initialization
x = (int*)calloc(n,sizeof(int));
…
// Parallelizable part
do {
   for (i=0; i<n; i++) {
       x[i] = f(i);
   }
   // check for convergence
   done = …;
} while (!done)
```

K iterations before convergence, (parallel) convergence check cheap, f(i) fast O(1)…

Tseq(n) = n+K+Kn

Tpar(p,n) = n+K+Kn/p

Sequential fraction ≈ 1/(1+K)

$S_p(n)$ —> 1+K

Problem: calloc(n) system function initializes memory and takes O(n) time

Example:

```
// Sequential initialization
x = (int*)malloc(n*sizeof(int));
…
// Parallelizable part
do {
   for (i=0; i<n; i++) {
       x[i] = f(i);
   }
   // check for convergence
   done = …;
} while (!done)
```

$S_p(n) \longrightarrow p$ when $n>p$ and $n \longrightarrow \infty$

K iterations before convergence, (parallel) convergence check cheap, f(i) fast O(1)…

Tseq(n) = 1+K+Kn

Tpar(p,n) = 1+K+Kn/p

Sequential part ≈ 1/(1+n)

Note:
A constant sequential part (not constant fraction) does not limit SU

Example:

```
// Sequential initialization
x = (int*)malloc(n*sizeof(int));
…
// Parallelizable part
do {
   for (i=0; i<n; i++) {
       x[i] = f(i);
   }
   // check for convergence
   done = …;
} while (!done)
```

$S_p(n)$ —> p when n>p, n —> ∞

K iterations before convergence, (parallel) convergence check cheap, f(i) fast O(1)…

Tseq(n) = 1+K+Kn

Tpar(p,n) = 1+K+Kn/p

Sequential part ≈ 1/(1+n)

Lesson:
Be careful with system functions (calloc, malloc), may need to be parallelized as well

Avoiding Amdahl: Scaled speed-up, efficiency

Sequential, strictly non-parallelizable part is most often not a constant fraction of the total execution time (number of instructions)

Indeed, the sequential part s(n) may decrease with problem size n.

Good speed-up can be maintained by increasing problem size with p

Recall Tpar(p,n) = s(n)Tseq(n) + (1-s(n))/p Tseq(n)

Not constant fraction

Assume

$$Tseq(n) = t(n)+T(n)$$

with sequential part t(n) and perfectly parallelizable part T(n), such that

$$Tpar(p,n) = t(n)+T(n)/p$$

Assume $t(n)/T(n) \rightarrow 0$ for $n \rightarrow \infty$

The speed-up as a function of p and n is

$$S_p(n) = (t(n)+T(n)) / (t(n)+T(n)/p)$$
$$= (t(n)/T(n)+1) / (t(n)/T(n)+1/p) \rightarrow 1/(1/p) = p \text{ for } n \rightarrow \infty$$

Definition:

Speed-up as function of p and n, with sequential and parallelizable times t(n) and T(n) is termed <u>scaled speed-up</u>

Lesson:
Depending on how fast t(n)/T(n) converges, linear speed-up can be achieved by increasing problem size n accordingly

With Tpar(p,n) = t(n)+T(n)/p, the fastest possible parallel time is T_∞(n) = t(n), and the parallelism is Tpar(1,n)/T∞(n) = (t(n)+T(n)) / t(n) = 1+T(n)/t(n).

Small t(n) relative to T(n) means large parallelism

Special case (Gustafson-Barsis “law”):

Assume the parallelizable part of the work increases linearly in p with T(n) = pt(n). Then

$$S_p(n) = (t(n)+T(n)) / (t(n)+T(n)/p)$$
$$= (t(n)+pt(n)) / (t(n)+t(n)) = (p+1)/2$$

John L. Gustafson: Reevaluating Amdahl's Law. Commun. ACM 31(5): 532-533 (1988)

(The paper actually says something different, makes the calculation somewhat similar to the proof of Amdahl’s law, in a way that doesn’t really make sense (in my opinion))

Definition:
The <u>efficiency</u> of parallel algorithm Par is the ratio of best possible parallel time to actual parallel time for given p and n:

$$E(p,n) = (Tseq(n)/p) / Tpar(p,n)$$
$$= S_p(n)/p = Tseq(n) / (p\ Tpar(p,n))$$

Cost, so efficiency is also ratio of sequential to parallel cost

Remarks:

- Tseq(n) time for best known/possible sequential algorithm

- $E(p,n) \leq 1$, since $S_p(n) = Tseq(n)/Tpar(n,p) \leq p$
- $E(p,n) = c$ (constant, ≤ 1): linear speed-up
- Cost-optimal algorithms have constant efficiency

Scalability

Definition:
A parallel algorithm/implementation is strongly scaling if $S_p(n) = \Theta(p)$ (linear, independent of (sufficiently large) n)

Definition:
A parallel algorithm/implementation is weakly scaling if there is a slowly growing function f(p), such that for $n = \Omega(f(p))$, E(p,n) remains constant. The function f is called the iso-efficiency function

Ananth Grama, Anshul Gupta, Vipin Kumar: Isoefficiency: measuring the scalability of parallel algorithms and architectures. IEEE Transactions Par. Dist. Computing. 1(3): 12-21 (1993)

Example:

Some work-optimal parallel algorithm runs in $O(n^2/p+\log^2 p)$. The iso-efficiency function for this algorithm ("how must problem size n increase as a function of p to maintain constant efficiency?") is

$e = n^2/(p(n^2/p+\log^2 p) = n^2/(n^2+p\log^2 p) \Leftrightarrow$

$n^2(1-e) = e\, p \log^2 p \Leftrightarrow$

$n = \sqrt{(e/(1-e))}\, \sqrt{p} \log p$

e ist the given efficiency

Efficiency e can be kept, if $n \geq \sqrt{(e/(1-e))}\, \sqrt{p} \log p$

Reminder:
$\log^2 n$ is shorthand for $(\log n)^2$, not log log n (iterated logarithm, which is written $\log^{(2)} n$)

Example:

If the algorithm instead runs in $O(n^2/p+\log^2 n)$, the iso-efficiency function for this algorithm ("how must problem size n increase as a function of p to maintain constant efficiency?") is

$e = n^2/(p(n^2/p+\log^2 n)) = n^2/(n^2+p\log^2 n) \Leftrightarrow$
$n^2(1-e) = e\, p \log^2 n \Leftrightarrow$
$n/\log n = \sqrt{(e/(1-e))}\sqrt{p}$

O-constants normalized to 1

No analytical solution

But we can maintain efficiency at least e, if $n/\log n \geq \sqrt{(e/(1-e))}\sqrt{p}$

Reminder:
$\log^2 n$ is shorthand for $(\log n)^2$, not $\log \log n$ (iterated logarithm, which is written $\log^{(2)} n$)

Example:

Parallel running time

$O(n^2/p+\log^2 n)$

Parallel "overhead" a function of problem size

vs.

$O(n^2/p+\log^2 p)$

Parallel "overhead" a function of number of processors, "caused by parallelization alone"

Both kind of algorithms/analyses occur frequently. Sometimes the latter is easier to handle (iso-efficiency), sometimes the former

Deriving the iso-efficiency function f(p)

Constant efficiency e in e = Tseq(n) / (p Tpar(p,n)), simplify, approximate, solve for n, gives function f(p) with the constant e somewhere that tells how n must grow with p to maintain constant e.

Technically, an algorithm is strongly scalable iff f(p) = O(1).

This is, technically speaking, never the case: All algorithms are at best weakly scalable, at least as much work is required as there are processors.
But often, constants and lower order terms can safely be ignored, so that the algorithm is strongly scalable for some range of n and p

Summary: Stating parallel performance

It is convenient to state parallel performance and scalability of a parallel algorithm/implementation as

$$Tpar(p,n) = O(T(n)/p+t(p,n))$$

T(n) represents the parallel part, t(p,n) the non-parallel part of the algorithm beyond which no improvement is possible, regardless of how many processors are used. The parallelism is 1+T(n)/t(p,n)

The cost of the algorithm is

$$W = O(p(T(n)/p+t(p,n))) = O(T(n)+pt(p,n))$$

The algorithm is cost-optimal when T(n) is O(Tseq(n)) and pt(p,n) is O(Tseq(n))

Example (again):

```
// Sequential initialization
x = (int*)malloc(n*sizeof(int));
…
// Parallelizable part
do {
   for (i=0; i<n; i++) {
       x[i] = f(i);
   }
   // check for convergence
   done = …;
} while (!done)
```

K iterations before convergence, (parallel) convergence check cheap, f(i) fast O(1)…

Tseq(n) = 1+K+Kn

Tpar(p,n) = 1+K+Kn/p

Sequential part ≈ 1/(1+n)

This code is <u>weakly scalable</u>, n has to increase as Ω(p log p) to maintain constant efficiency, Ω(log p) per processor (if the work in the iterations is load-balanced)

Speed-up in practice

Speed-up as an empirical quantity, “measured time”, based on experiment (benchmark)

Tseq(n): Running time for “reasonable”, good, best available, sequential implementation, on “reasonable” inputs

Tpar(p,n): Parallel running time, measured for a number of experiments with different, typical, relevant (worst-case? best-case?) inputs

$$S_p(n) = Tseq(n)/Tpar(p,n)$$

Empirical speed-up typically not independent of problem size n, and problem instance

Empirical, relative speed-up without absolute performance baseline (and comparison to reasonable, sequential algorithm and implementation) is grossly misleading

David H. Bailey: Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers. Supercomputing Review, Aug. 1991, pp. 54-55
Torsten Hoefler, Roberto Belli: Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. SC 2015: 73:1-73:12

Scalability analysis

- Strong scaling: Keep problem size (work) fixed, increase number of processors. Algorithm/implementation is strongly scaling, if Tpar(p,n) decreases proportionally to p (linear speed-up).
- Weak scaling (alternative definition): Keep average work (work per processor) fixed, that is increase problem size together with number of processors. Algorithm/implementation is weakly scaling if the running time remains constant (=Tseq(n') for non-scaled input of size n'). Let K = Tseq(n'), then the input size scaling function is n = $Tseq^{-1}(pK)$ = g(p)

For input of size n, the average work for p processors is Tseq(n)/p. In the weak scaling analysis, this is to be kept constant, e.g., Tseq(n')

Example:
Consider again algorithms for matrix-vector multiplication running in parallel time $O(n^2/p+\log n)$ with sequential work $O(n^2)$.

The average work (work per processor) should be kept fixed at $O(n'^2)$ for some n'. The total work with p processors is $O(p\, n'^2)$, for which input of size n in $O(n'\sqrt{p})$ is needed (solve $n^2 = (p\, n'^2)$).

The running time is $O((n'\sqrt{p})^2/p + \log(n'\sqrt{p})) = O(n'^2 + \log n' + \log\sqrt{p})$, which is $O(n'^2)$ as long as $\log\sqrt{p}$ is smaller than (some constant times) n'^2.

This algorithm is weakly scaling up to a very large number of processors (exponential in n').
The algorithm is also strongly scalable (linear speed-up) up to p in $O(n^2/\log n)$ processors (the parallelism)

Combining the two notions of weak scaling, a parallel algorithm Par is weakly scaling if the iso-efficiency function f(p) is growing no faster than the input size scaling function g(p)

The second, alternative notion of weak scaling (keep parallel time constant) puts an upper bound on the growth of the slowly growing iso-efficiency function. This is a (sometimes too) strong requirement.

Limitations of speed-up as an empirical measure

Empirical speed-up (speed-up in practice) assumes that Tseq(n) can be measured.

For very large n and p, this may not be the case: A large HPC system has much more (distributed) main memory than any single-processor system

Scalability measured by other means:

- Stepwise speed-up (1-1000 processors, 1000-10,000 processors, 10,000 to 100,000 processors, …)
- Other notions of efficiency

Sometimes Tseq(n) may be so large that it cannot be measured for real (large, combinatorially hard problems)

Examples (Tpar, Speed-up, Optimality, Efficiency, Iso-efficiency):

Linear time computation, Tseq(n) = n (constants ignored)

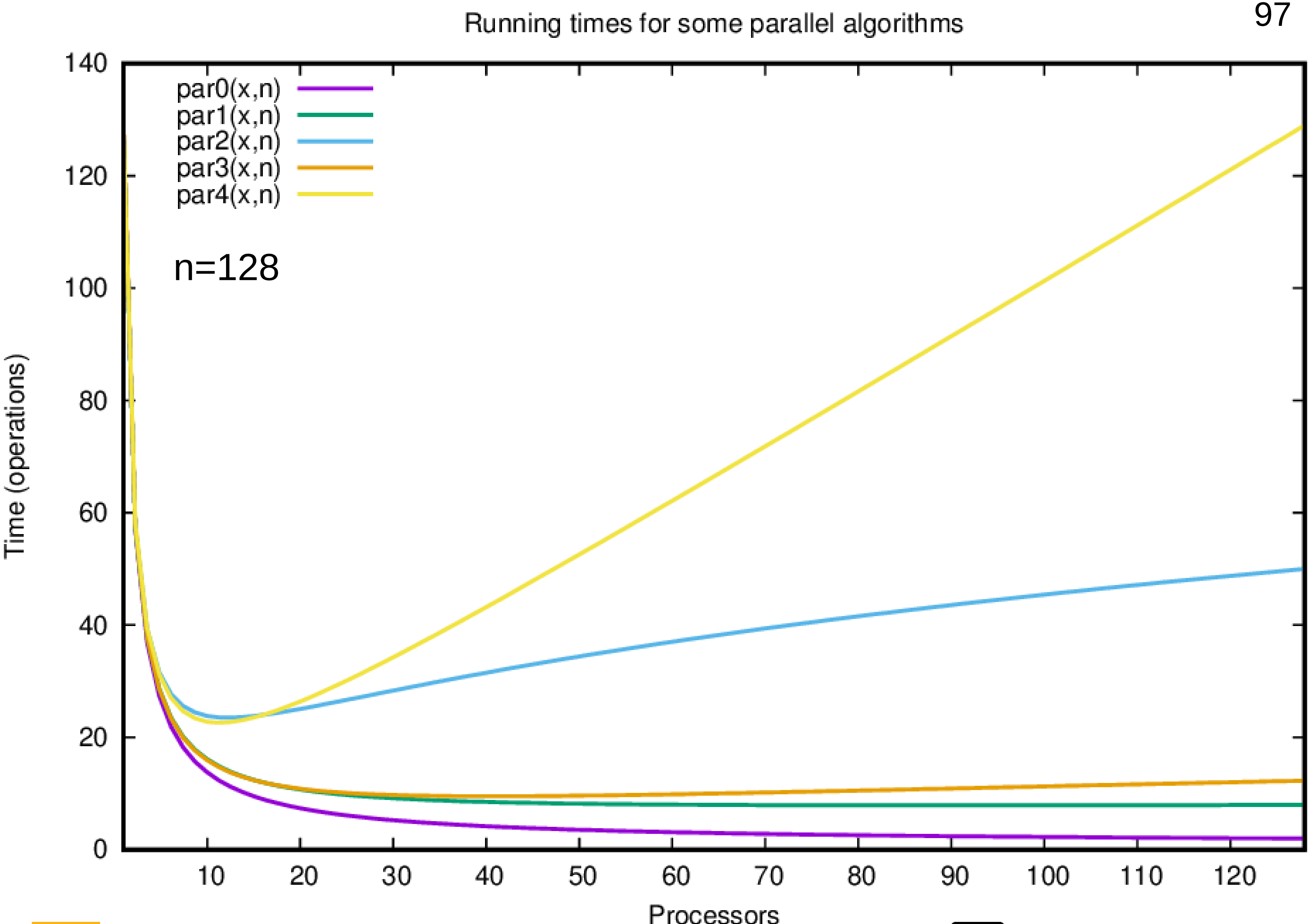
Typical, good, work-optimal parallel algorithms

1. Tpar0(p,n) = n/p+1
2. Tpar1(p,n) = n/p+log p
3. Tpar2(p,n) = n/p+$\log^2$ p
4. Tpar3(p,n) = n/p+$\sqrt{p}$
5. Tpar4(p,n) = n/p+p

Embarrassingly "data parallel" computation, constant overhead

logarithmic overhead, e.g. convergence check

Linear overhead, e.g. data exchange


Running times for some parallel algorithms
par0(x,n)
par1(x,n)
par2(x,n)
par3(x,n)
par4(x,n)
n=128
Time (operations)
Processors
0
20
40
60
80
100
120
140
10
20
30
40
50
60
70
80
90
100
110
120

Cost-optimality:

1. $\text{Tpar0}(p,n) = n/p+1$: $p\text{Tpar}(p,n) = n+p = O(n)$ for $p=O(n)$

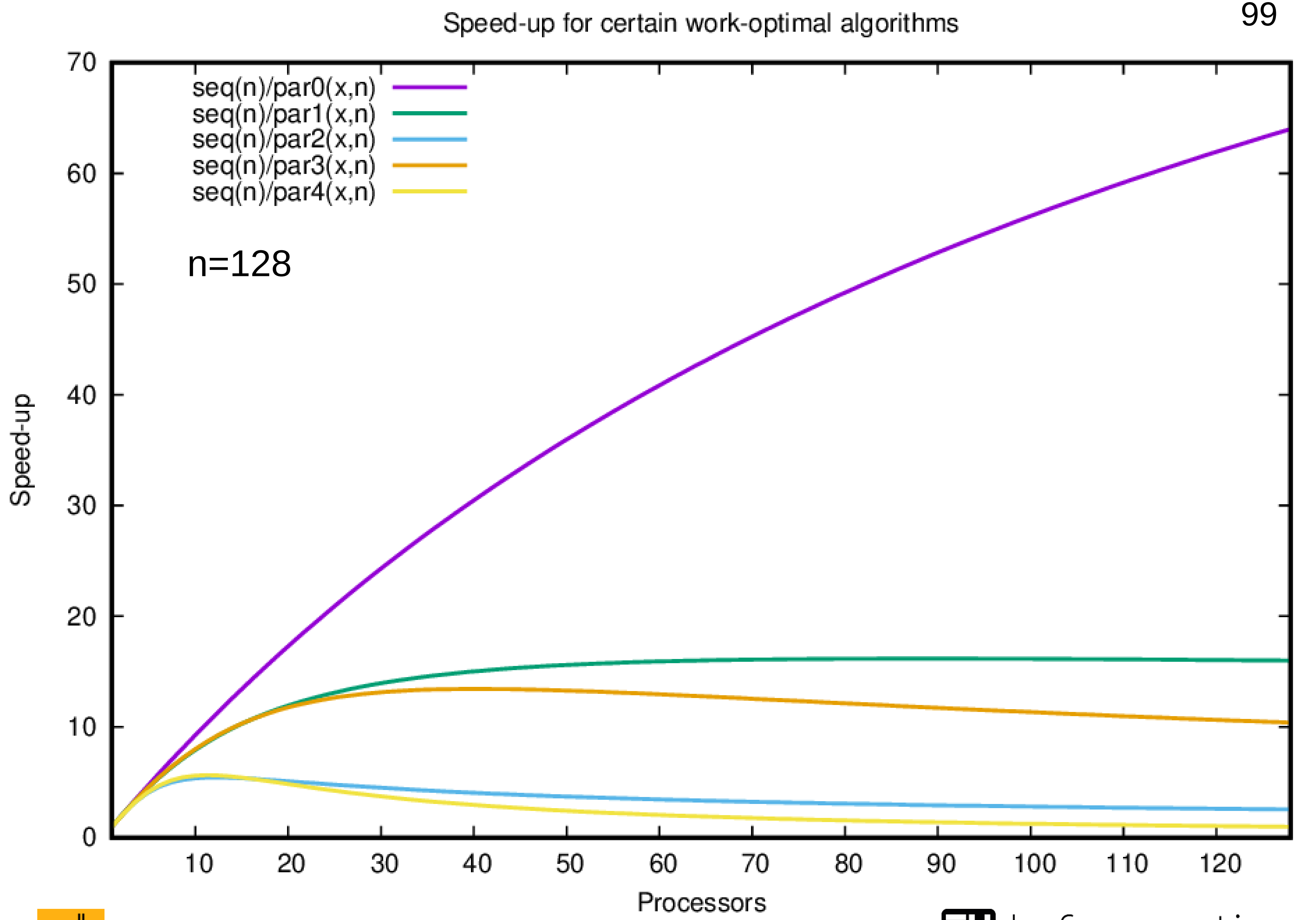
2. $\text{Tpar1}(p,n) = n/p+\log p$:
$p\text{Tpar}(p,n) = n+p \log p = O(n)$ for $p \log p = O(n)$

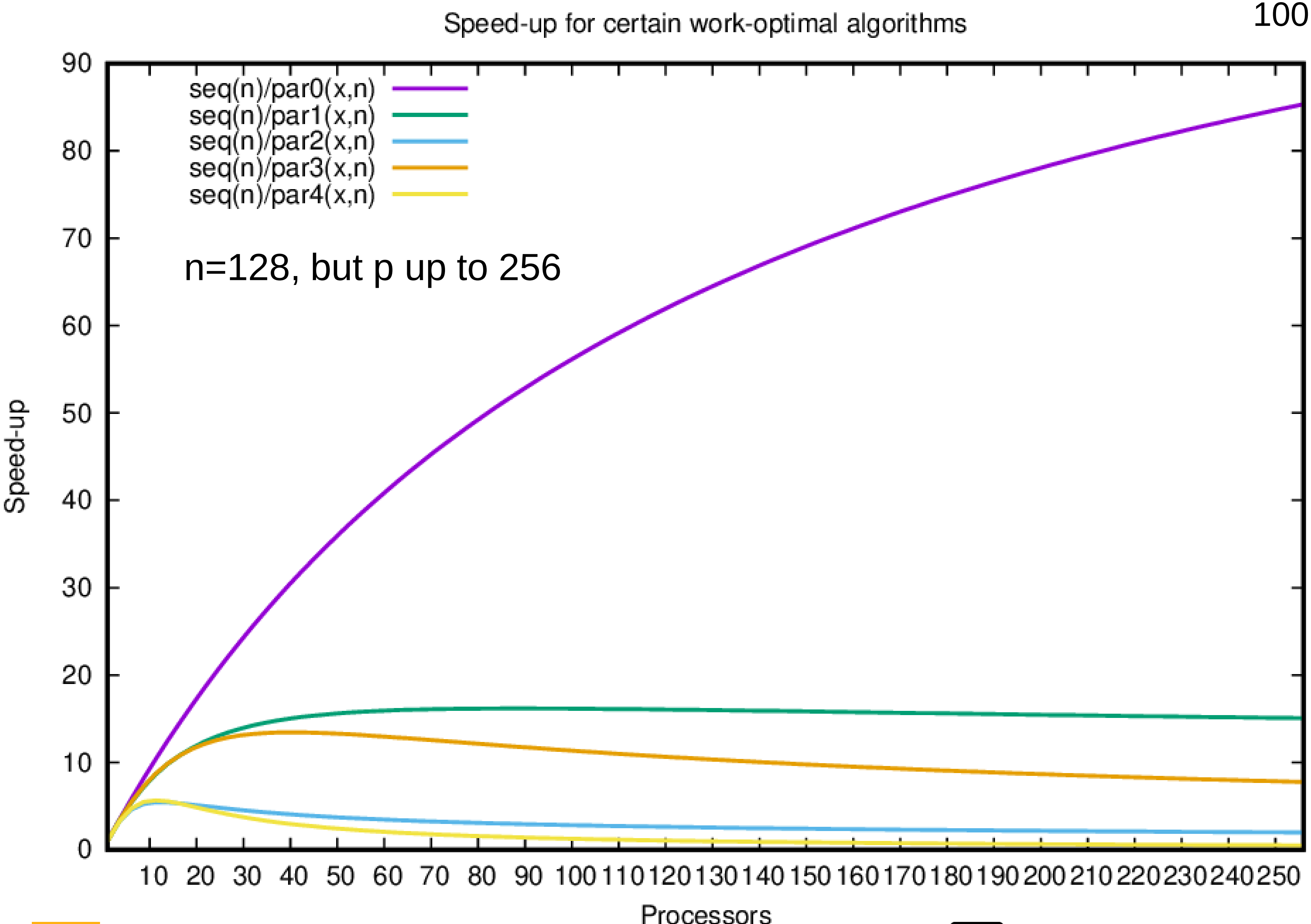
3. $\text{Tpar2}(p,n) = n/p+\log^2 p$:
$p\text{Tpar}(p,n) = n+p \log^2 p = O(n)$ for $p \log^2 p=O(n)$

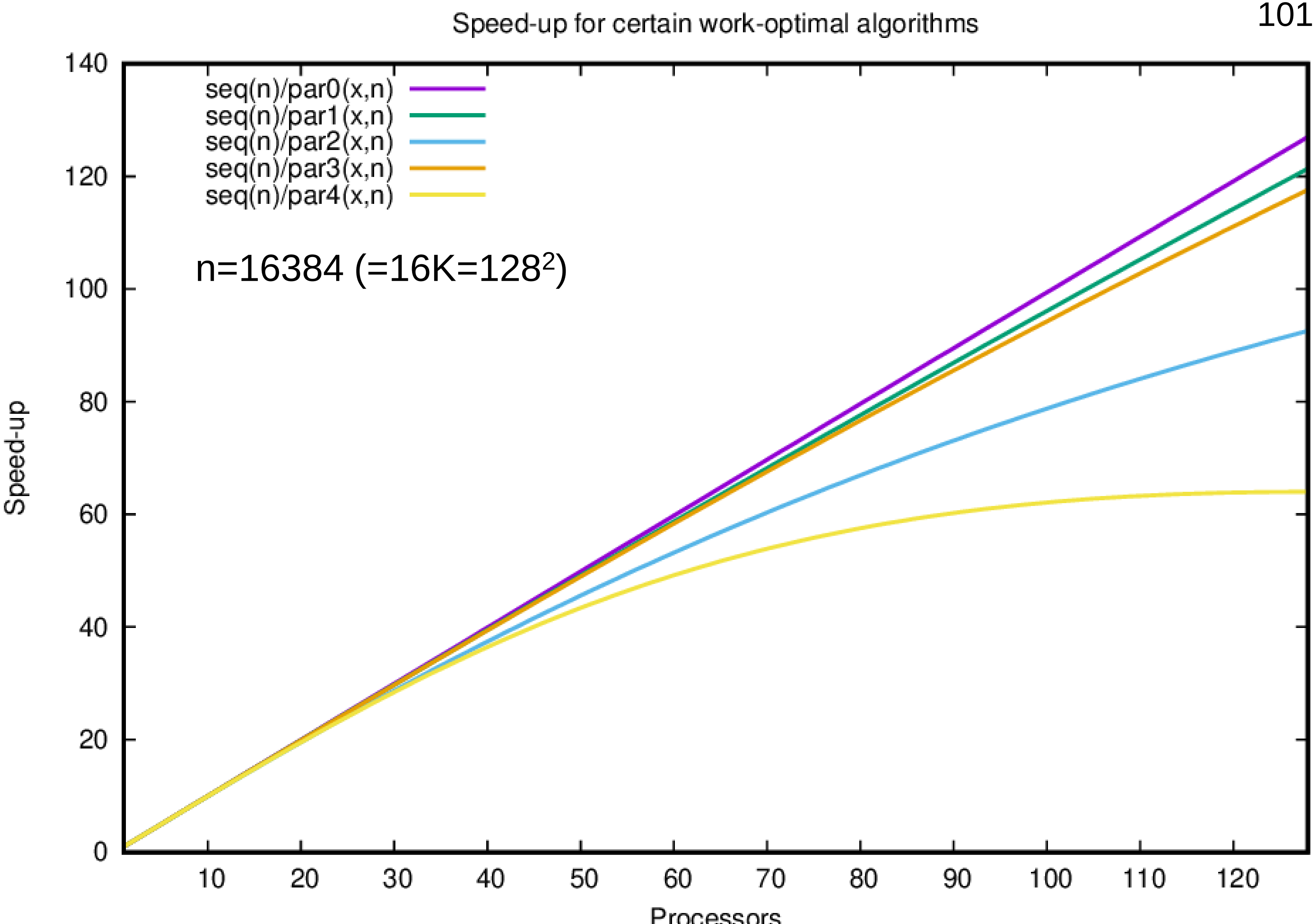
4. $\text{Tpar3}(p,n) = n/p+\sqrt{p}$: $p\text{Tpar}(p,n) = n+p\sqrt{p} = O(n)$ for $p\sqrt{p}=O(n)$

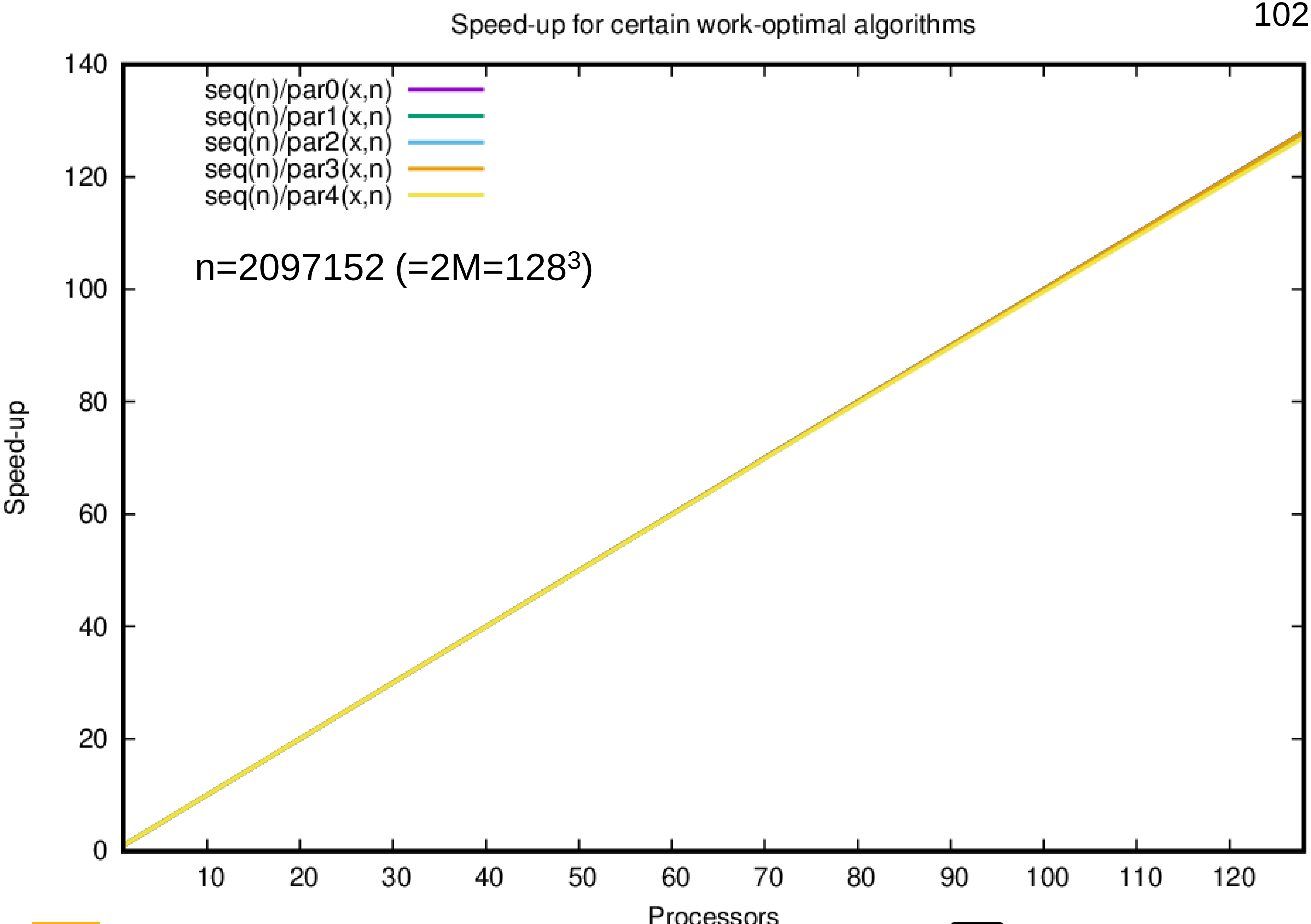
5. $\text{Tpar4}(p,n) = n/p+p$: $p\text{Tpar}(p,n) = n+p^2 = O(n)$ for $p^2=O(n)$

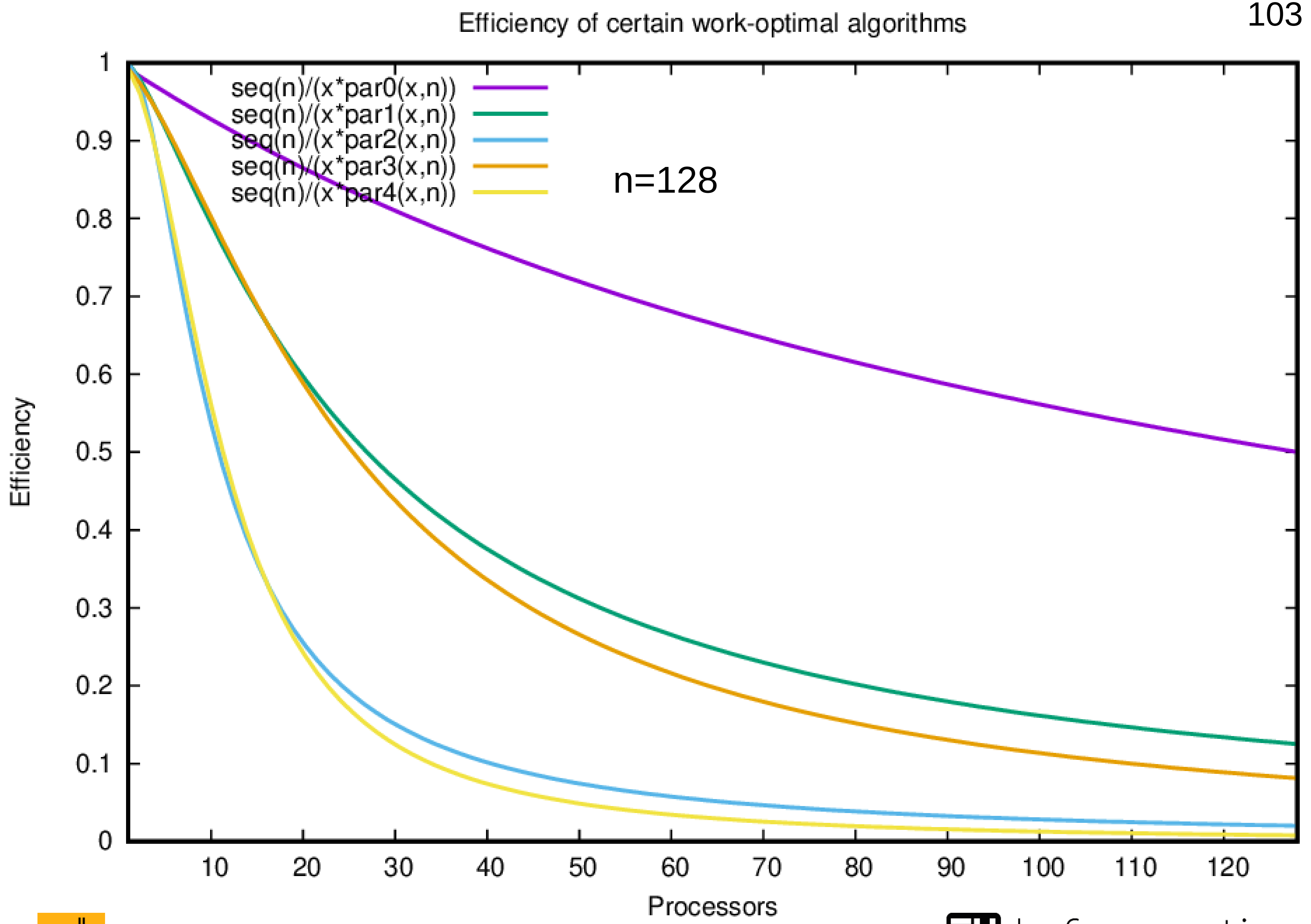
All five algorithms have potential for linear speed-up up to the calculated number of processors

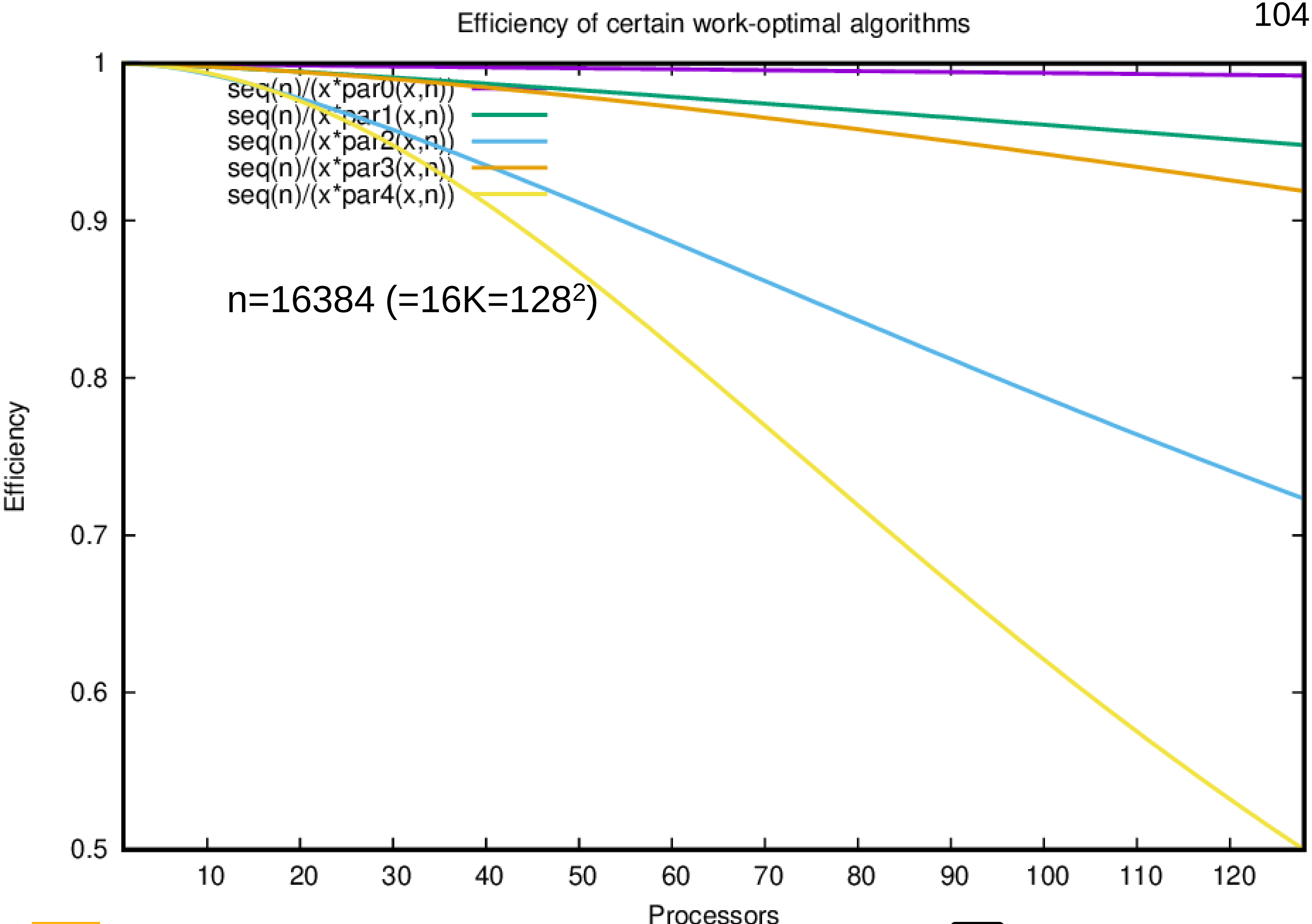

Speed-up for certain work-optimal algorithms
seq(n)/par0(x,n)
seq(n)/par1(x,n)
seq(n)/par2(x,n)
seq(n)/par3(x,n)
seq(n)/par4(x,n)
n=128
Speed-up
0
10
20
30
40
50
60
70
10
20
30
40
50
60
70
80
90
100
110
120
Processors

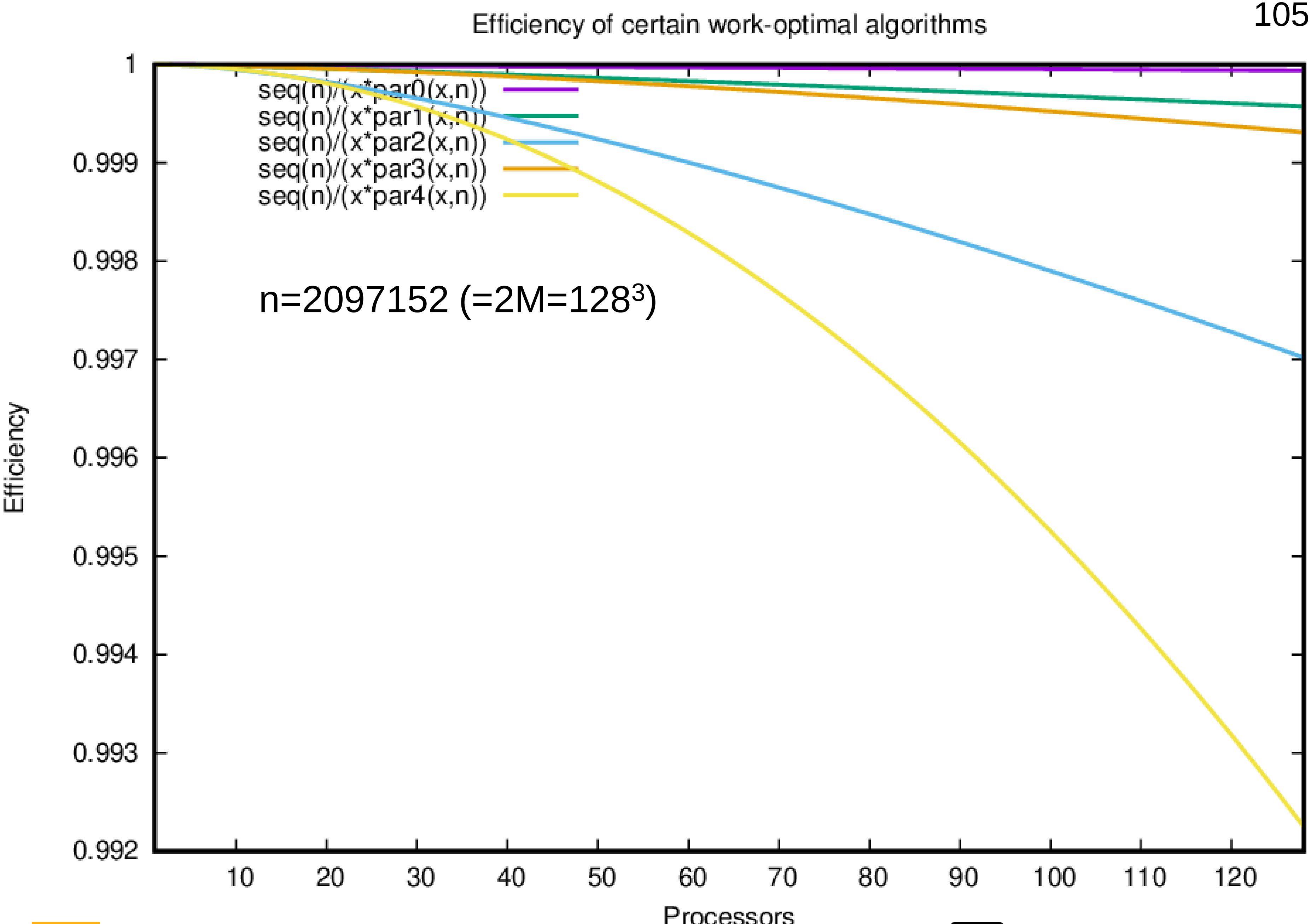

Speed-up for certain work-optimal algorithms
seq(n)/par0(x,n)
seq(n)/par1(x,n)
seq(n)/par2(x,n)
seq(n)/par3(x,n)
seq(n)/par4(x,n)
n=128, but p up to 256
Speed-up
0
10
20
30
40
50
60
70
80
90
10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200 210 220 230 240 250
Processors


Speed-up for certain work-optimal algorithms
seq(n)/par0(x,n)
seq(n)/par1(x,n)
seq(n)/par2(x,n)
seq(n)/par3(x,n)
seq(n)/par4(x,n)
n=16384 (=16K=128²)
Speed-up
Processors
0
20
40
60
80
100
120
140
10
20
30
40
50
60
70
80
90
100
110
120


Speed-up for certain work-optimal algorithms
seq(n)/par0(x,n)
seq(n)/par1(x,n)
seq(n)/par2(x,n)
seq(n)/par3(x,n)
seq(n)/par4(x,n)
n=2097152 (=2M=128³)
Speed-up
Processors
0
20
40
60
80
100
120
140
10
20
30
40
50
60
70
80
90
100
110
120


Efficiency of certain work-optimal algorithms
seq(n)/(x*par0(x,n))
seq(n)/(x*par1(x,n))
seq(n)/(x*par2(x,n))
seq(n)/(x*par3(x,n))
seq(n)/(x*par4(x,n))
n=128
Efficiency
Processors
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
10
20
30
40
50
60
70
80
90
100
110
120

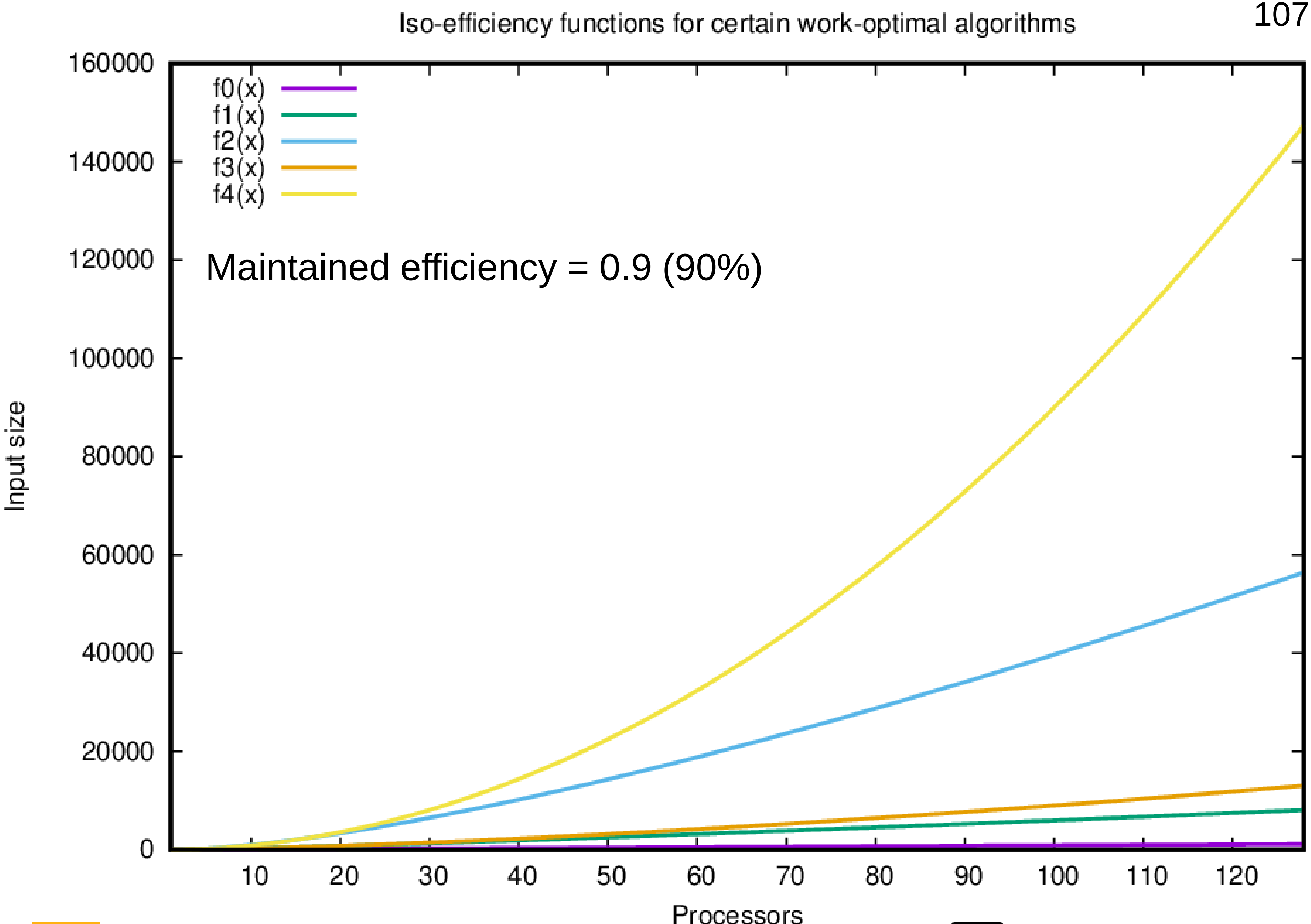

Efficiency of certain work-optimal algorithms
Efficiency
Processors
1
0.9
0.8
0.7
0.6
0.5
10
20
30
40
50
60
70
80
90
100
110
120
seq(n)/(x*par0(x,n))
seq(n)/(x*par1(x,n))
seq(n)/(x*par2(x,n))
seq(n)/(x*par3(x,n))
seq(n)/(x*par4(x,n))
n=16384 (=16K=128²)


Efficiency of certain work-optimal algorithms
seq(n)/(x*par0(x,n))
seq(n)/(x*par1(x,n))
seq(n)/(x*par2(x,n))
seq(n)/(x*par3(x,n))
seq(n)/(x*par4(x,n))
n=2097152 (=2M=128³)
Efficiency
Processors
1
0.999
0.998
0.997
0.996
0.995
0.994
0.993
0.992
10
20
30
40
50
60
70
80
90
100
110
120

Efficiency, iso-efficiency, weak scaling:

To maintain constant efficiency e=Tseq(n)/(pTpar(p,n)), n has to increase as

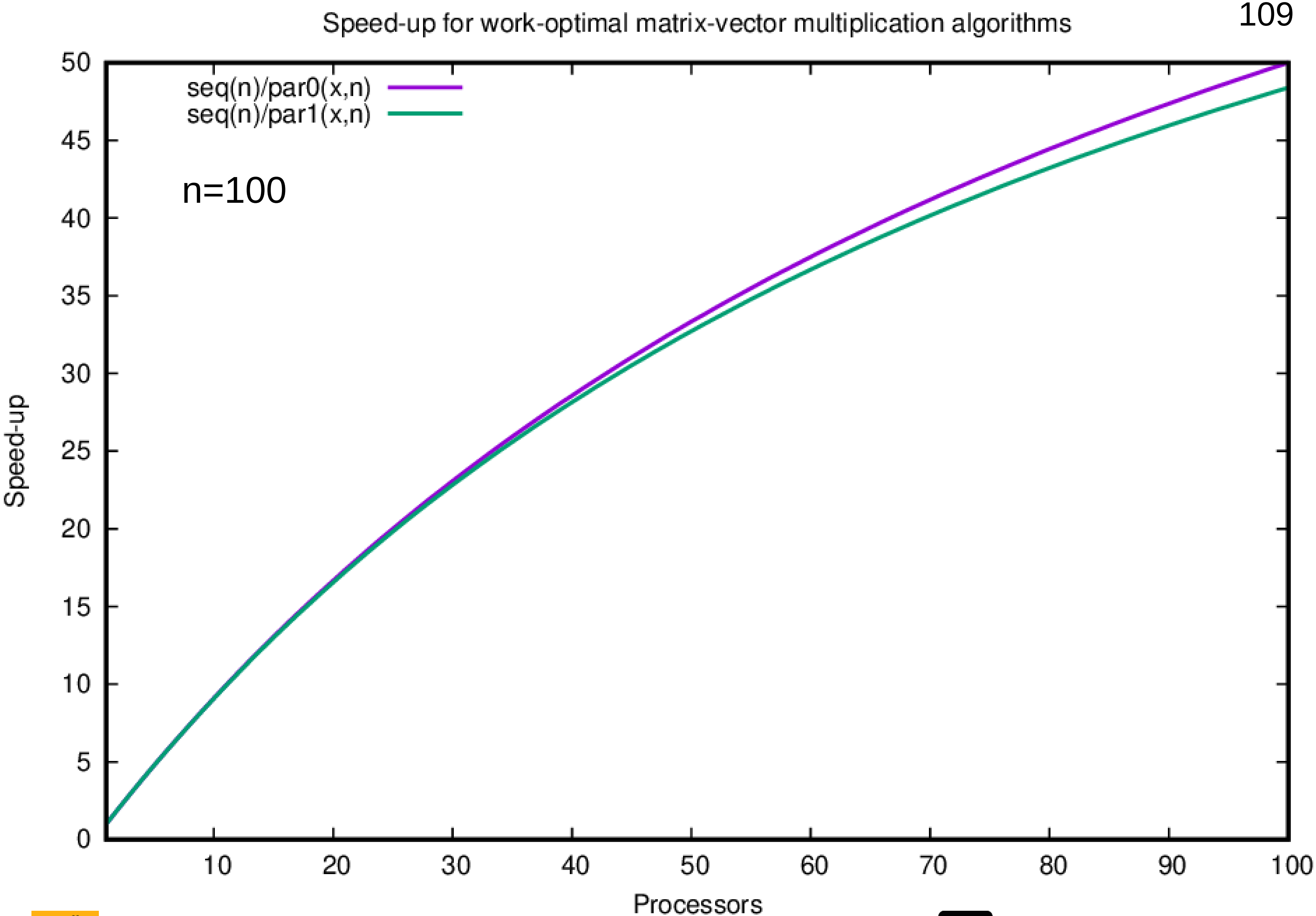
1. Tpar0(p,n) = n/p+1: f0(p) = [e/(1-e)] p
2. Tpar1(p,n) = n/p+log p: f1(p) = [e/(1-e)] (p log p)
3. Tpar2(p,n) = n/p+$\log^2$ p: f2(p) = [e/(1-e)] (p $\log^2$ p)
4. Tpar3(p,n) = n/p+$\sqrt{p}$: f3(p) = [e/(1-e)] (p$\sqrt{p}$)
5. Tpar4(p,n) = n/p+p: f4(p) = [e/(1-e)] p^2

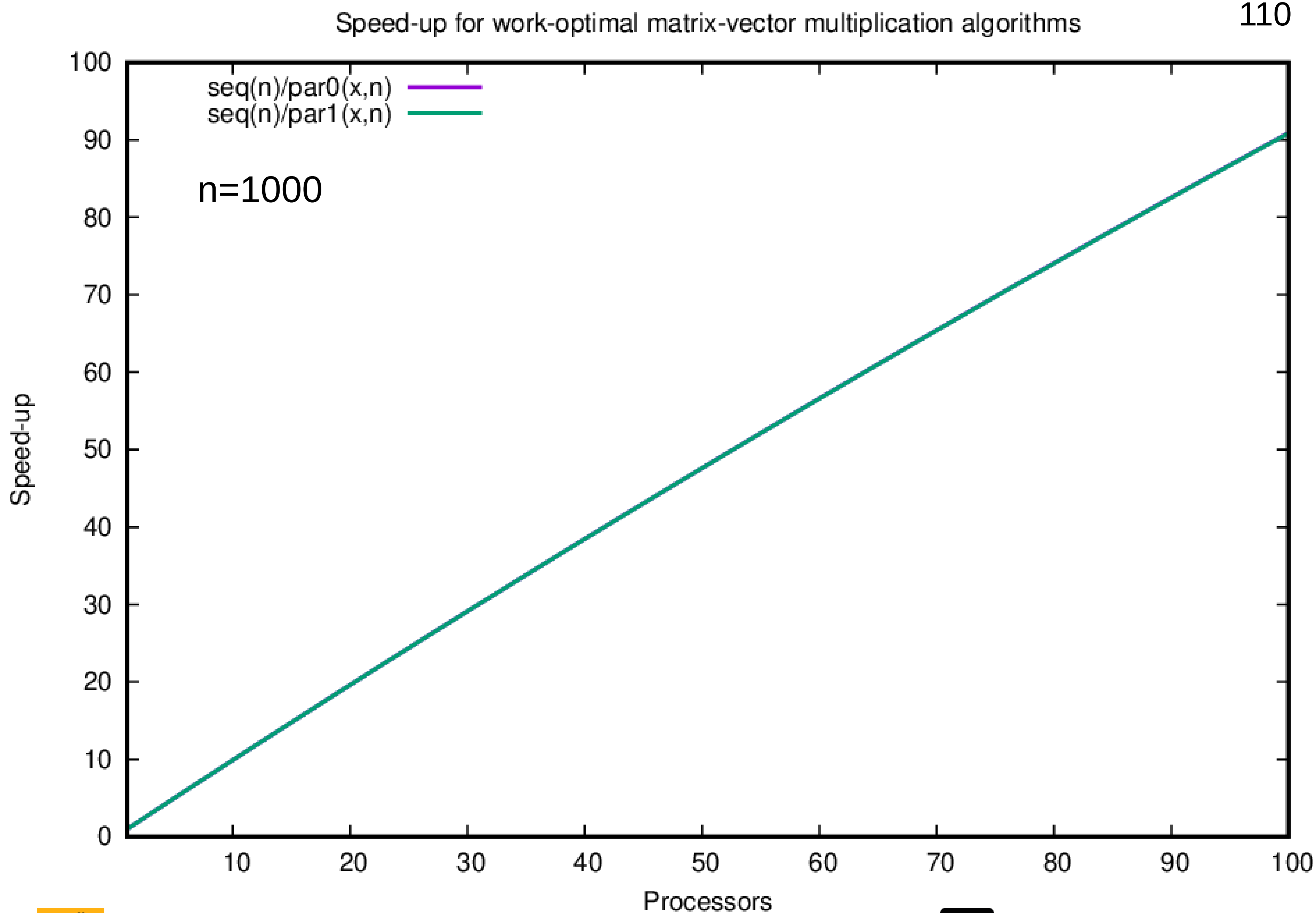

Iso-efficiency functions for certain work-optimal algorithms
f0(x)
f1(x)
f2(x)
f3(x)
f4(x)
Maintained efficiency = 0.9 (90%)
160000
140000
120000
100000
80000
60000
40000
20000
0
Input size
10
20
30
40
50
60
70
80
90
100
110
120
Processors

Matrix-vector multiplication parallelizations:

$Tseq(n) = n^2$

$Tpar0(p,n) = n^2/p + n$
$Tpar1(p,n) = n^2/p + n + \log p$


Speed-up for work-optimal matrix-vector multiplication algorithms
seq(n)/par0(x,n)
seq(n)/par1(x,n)
n=100
Speed-up
Processors
0
5
10
15
20
25
30
35
40
45
50
10
20
30
40
50
60
70
80
90
100


Speed-up for work-optimal matrix-vector multiplication algorithms
seq(n)/par0(x,n)
seq(n)/par1(x,n)
n=1000
Speed-up
Processors
0
10
20
30
40
50
60
70
80
90
100

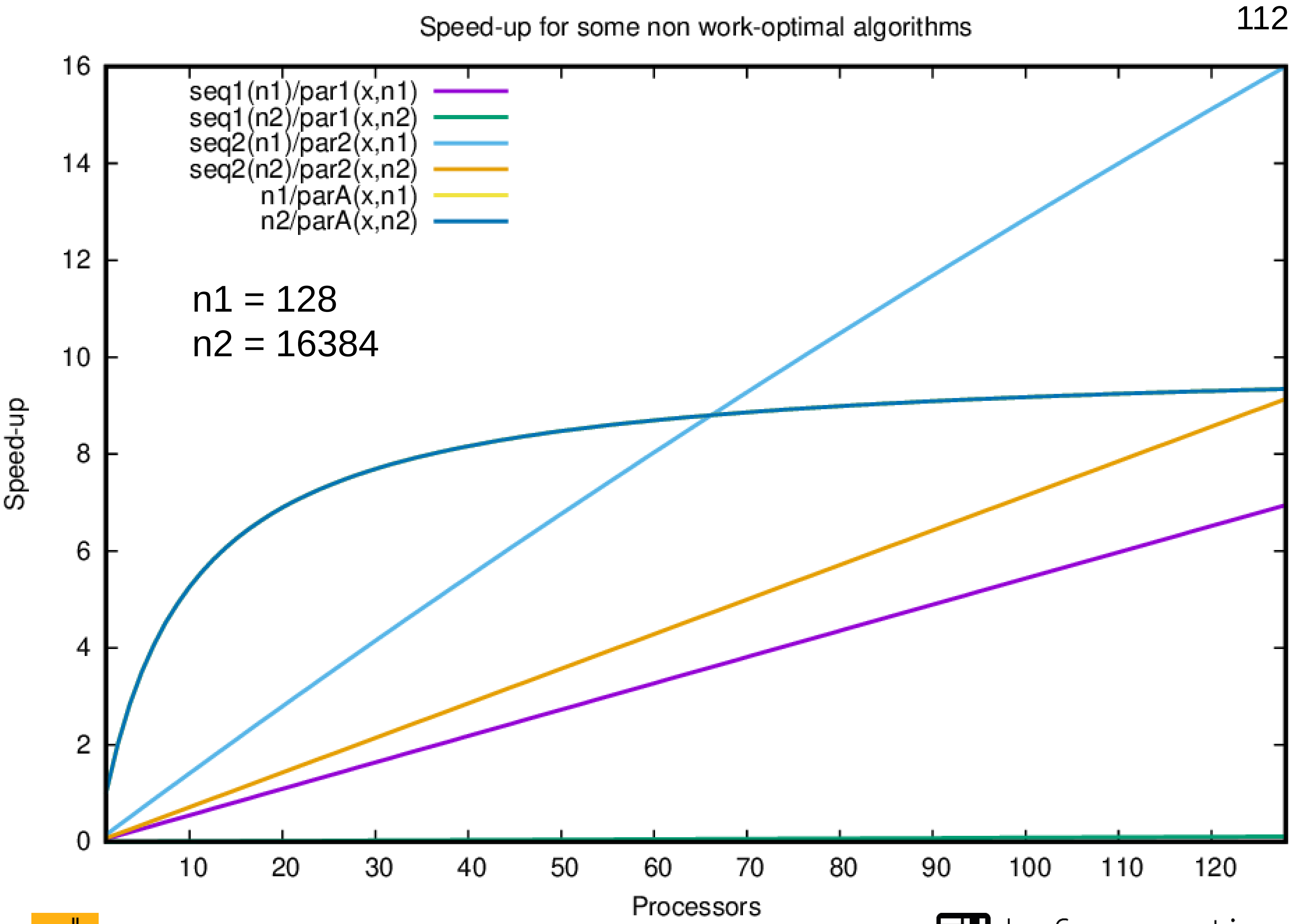
Some non work-optimal parallel algorithms

1. $Tseq1(n) = n \log n$ $\quad$ $Tpar1(p,n) = n^2/p+1$

2. $Tseq2(n) = n$ $\quad$ $Tpar2(p,n) = (n \log n)/p+1$

Amdahl case, linear sequential running time, 10% sequential fraction:

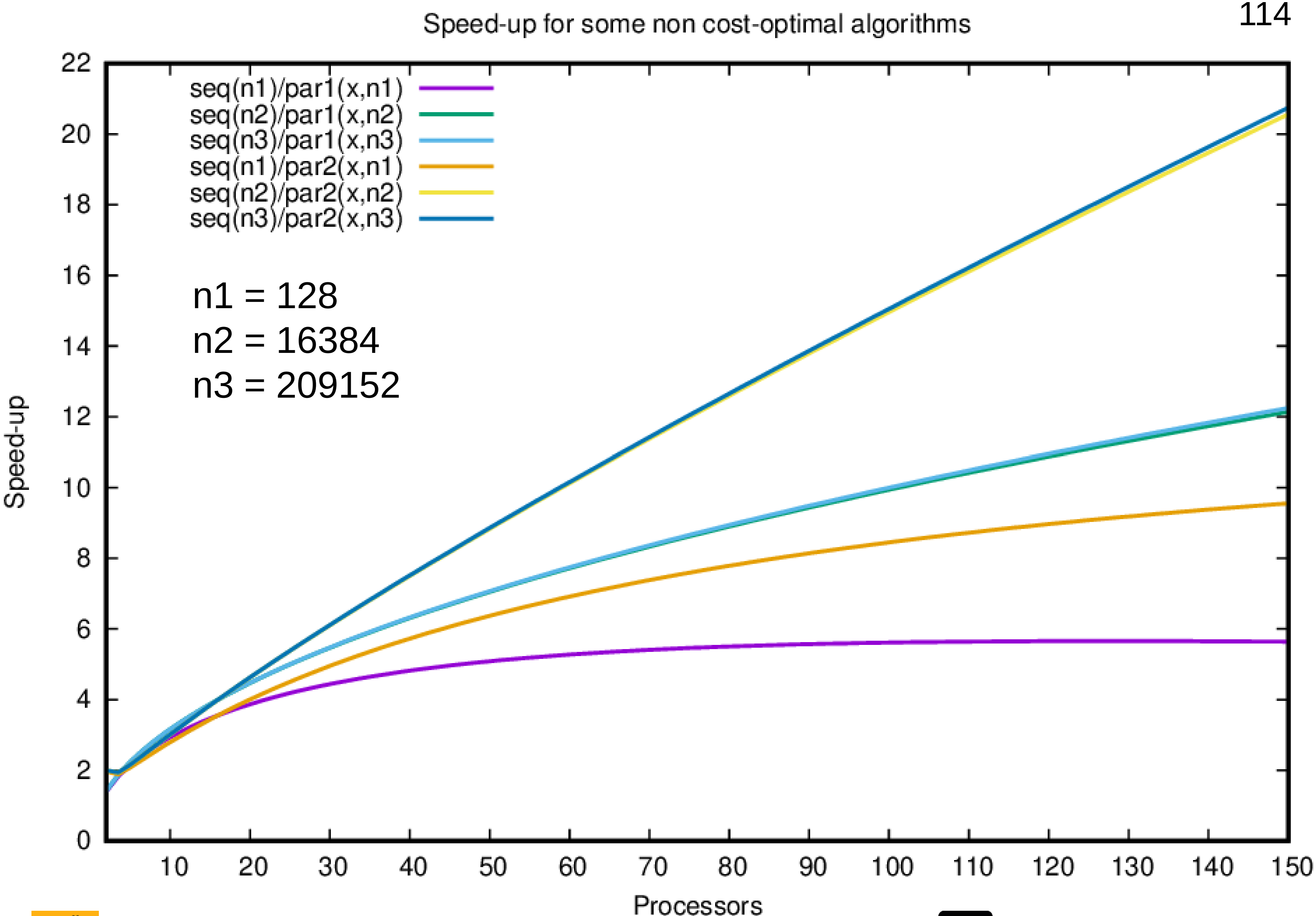
$TparA(p,n) = 0.9n/p+0.1n$


Speed-up for some non work-optimal algorithms
seq1(n1)/par1(x,n1)
seq1(n2)/par1(x,n2)
seq2(n1)/par2(x,n1)
seq2(n2)/par2(x,n2)
n1/parA(x,n1)
n2/parA(x,n2)
n1 = 128
n2 = 16384
Speed-up
Processors
0
2
4
6
8
10
12
14
16
10
20
30
40
50
60
70
80
90
100
110
120

Some non cost-optimal parallel algorithms

1. Tseq(n) = n   Tpar1(p,n) = n/√p + √p
2. Tseq(n) = n   Tpar2(p,n) = n/(p/log p) + log p

- S_p(n) = n/(n/√p + √p) = √pn/(n+p) = √p/(1+p/n). The fastest running time (equate the two terms in Tpar1(p,n)) and highest speedup is for p=n, with S_p = √p/2
- S_p(n) = n/(n/(p/log p) + log p) = pn/((log p)(n+p)) = (p/log p) (n/(n+p)) < p/log p. Again, the fastest running time and highest speed-up is for p = n with S_p = p/log p 1/2


Speed-up for some non cost-optimal algorithms
seq(n1)/par1(x,n1)
seq(n2)/par1(x,n2)
seq(n3)/par1(x,n3)
seq(n1)/par2(x,n1)
seq(n2)/par2(x,n2)
seq(n3)/par2(x,n3)
n1 = 128
n2 = 16384
n3 = 209152
Speed-up
Processors
0
2
4
6
8
10
12
14
16
18
20
22
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150

Lecture summary, checklist

- Sequential baseline
- Sequential and parallel time, Tseq, Tpar

- Speed-up (in theory and practice)
- Work and cost optimality

- Amdahl's law
- Efficiency, iso-efficiency function
- Scaled speed-up, strong and weak scaling