Introduction to Parallel Computing

Jesper Larsson Träff
Technical University of Vienna
Parallel Computing

Parallel computing:
„how to accomplish something as a coordinated team (CS: of computers carrying out an algorithm)"

Why study parallel computing?

- It's interesting, highly non-trivial
- Key discipline of computer science (von Neumann, golden theory decade: 1980-90)
- It's ubiquituous (gates, architecture: pipelines, ILP, TLP, systems: operating systems, software), not always opaque
- It's useful: large, extremely computationally intensive problems, Scientific Computing, HPC
- It's inevitable: multi-core revolution, GPGPU paradigm, …
- …

Parallel computing:
The discipline of efficiently utilizing dedicated parallel resources (processors, memories, …) to solve a single, given computation problem.

Specifically:
Parallel resources with significant inter-communication capabilities, for problems with non-trivial communication and computational demands

Buzz words: tightly coupled, dedicated parallel system; multi-core processor, GPGPU, High-Performance Computing (HPC), …

Distributed computing:
The discipline of making independent, non-dedicated resources coorperate toward solving a specified problem complex.

Typical concerns: correctness, availability, progress, security, integrity, privacy, robustness, fault tolerance, …

Buzz words: internet, grid, cloud, agents, autonomous computing, …

Concurrent computing:
The discipline of managing and reasoning about interacting processes that may (or may) not take place simultaneously

Typical concerns: correctness (often formal), e.g. deadlock-freedom, starvation-freedom, mutual exclusion, fairness

Buzz words: operating systems concepts, autonomous computing, process calculi, CSP, CCS

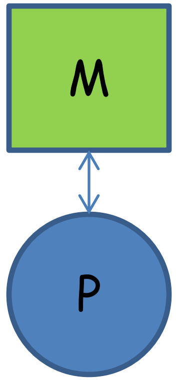
Parallel computing as a theoretical CS discipline

(Traditional) concern/objective: how to solve a given computational problem faster

- How fast can a given problem be solved? How many resources can be productively exploited?
- What is a reasonable conception („model") for parallel computing?
- Are there problems that cannot be solved in parallel? Fast? At all?
- …

<u>Architecture model</u>:
Abstraction of the important modules of a computational system (processor) , their interconnection and interaction.

Used as basis for the specification of a <u>computational model</u>: (formal) framework for the specification of algorithms for the computational system, including cost model.



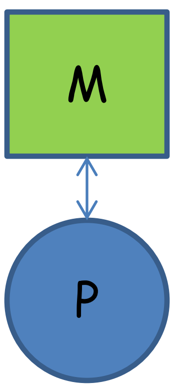
Example: RAM (Random-Access Machine)

Processor (ALU, PC, registers) capable of executing instructions stored in memory on data in memory

Execution of instruction, access to memory: unit cost

Architecture model:
Abstraction of the important modules of a computational system (processor) , their interconnection and interaction.

Used as basis for the specification of a computational model: (formal) framework for the specification of algorithms for the computational system, including cost model.



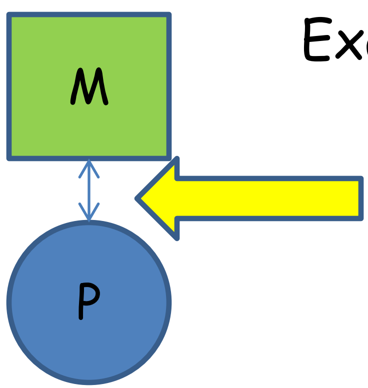
Example: RAM (Random-Access Machine)

Aka von Neumann architecture, stored program computer (contrast: finite state automaton)

[John von Neumann (1903-57), Report on EDVAC, 1945], also Eckert&Mauchly, ENIAC

<u>Architecture model</u>:
Abstraction of the important modules of a computational system (processor) , their interconnection and interaction.

Used as basis for the specification of a <u>computational model</u>: (formal) framework for the specification of algorithms for the computational system, including cost model.



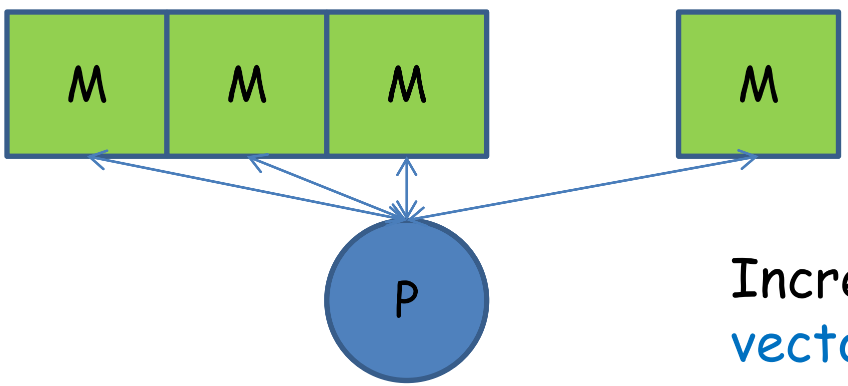
Example: RAM (Random-Access Machine)

„von Neumann bottleneck“: program and data separate from CPU, processing rate limited by memory rate.

[John Backus, Turing Award Lecture, 1977]

<u>Architecture model</u>:
Abstraction of the important modules of a computational system (processor) , their interconnection and interaction.

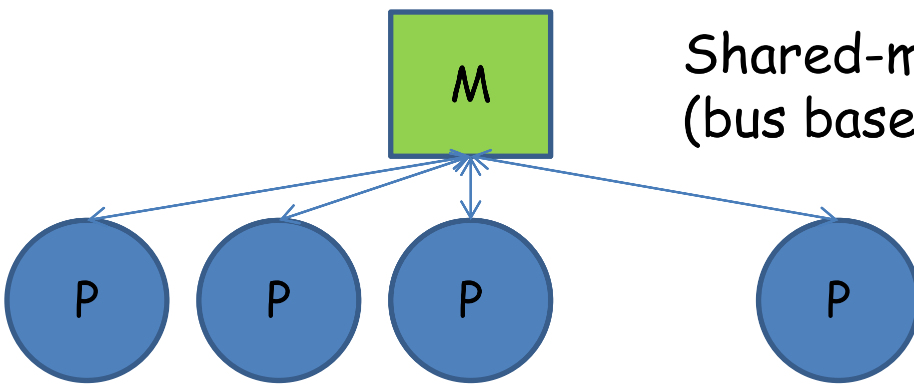
Used as basis for the specification of a <u>computational model</u>: (formal) framework for the specification of algorithms for the computational system, including cost model.



Increased memory rate, vector computer, ALU operates on vectors instead of scalars

Architecture model:
Abstraction of the important modules of a computational system (processor) , their interconnection and interaction.

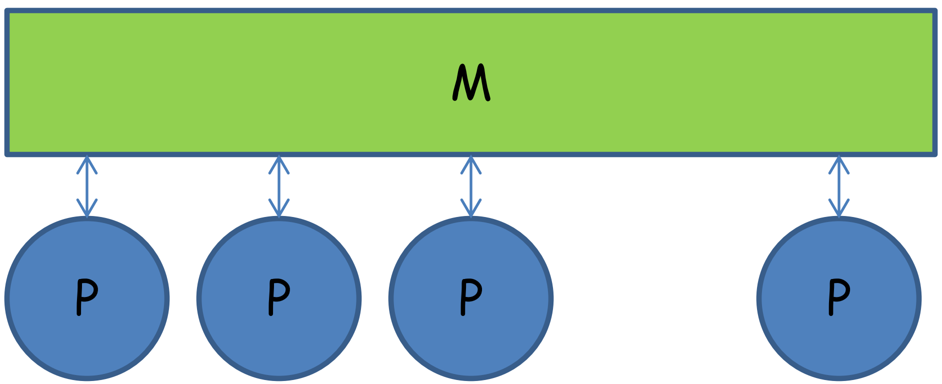
Used as basis for the specification of a computational model: (formal) framework for the specification of algorithms for the computational system, including cost model.



Shared-memory model
(bus based?)

Architecture model:
Abstraction of the important modules of a computational system (processor) , their interconnection and interaction.

Used as basis for the specification of a computational model: (formal) framework for the specification of algorithms for the computational system, including cost model.

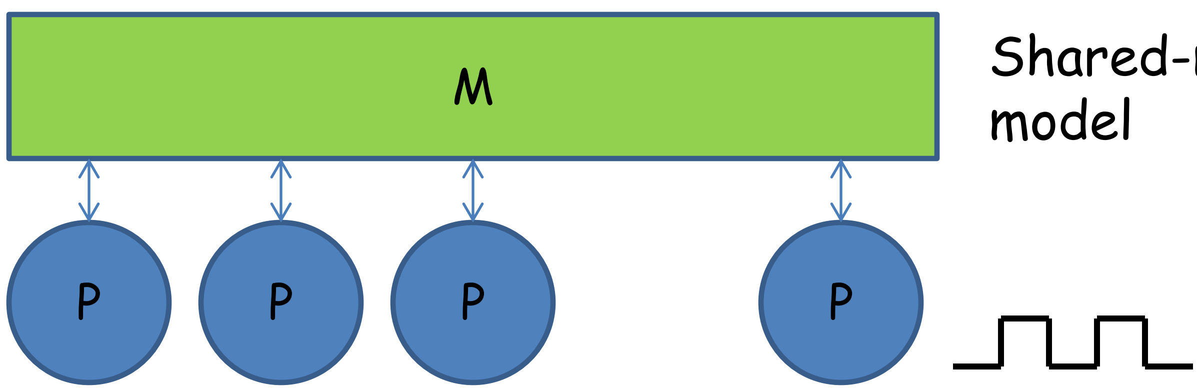


Shared-memory model

Parallel Computing

Architecture model:
Abstraction of the important modules of a computational system (processor) , their interconnection and interaction.

Used as basis for the specification of a computational model:
(formal) framework for the specification of algorithms for the computational system, including cost model.



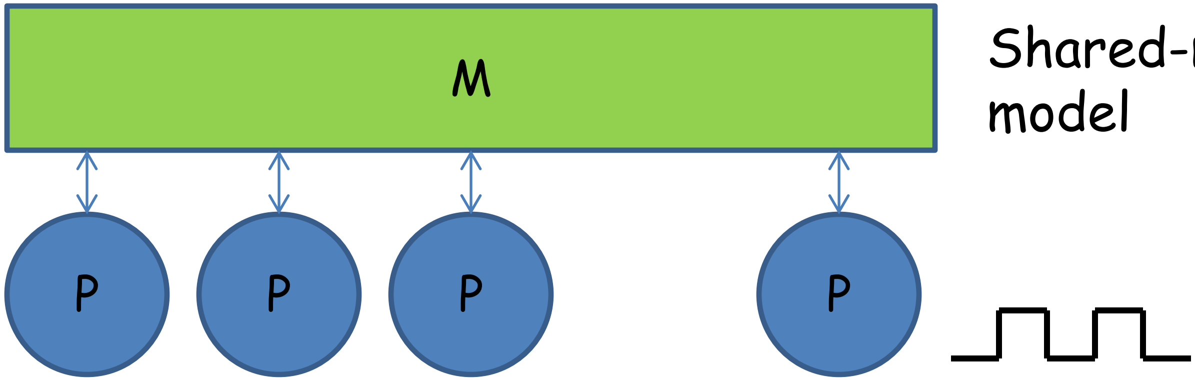
Shared-memory model

Processors operate in lock-step, uniform memory access time = instruction time: Parallel RAM (PRAM)

TU WIEN

Architecture model:
Abstraction of the important modules of a computational system (processor), their interconnection and interaction.

Used as basis for the specification of a computational model: (formal) framework for the specification of algorithms for the computational system, including cost model.

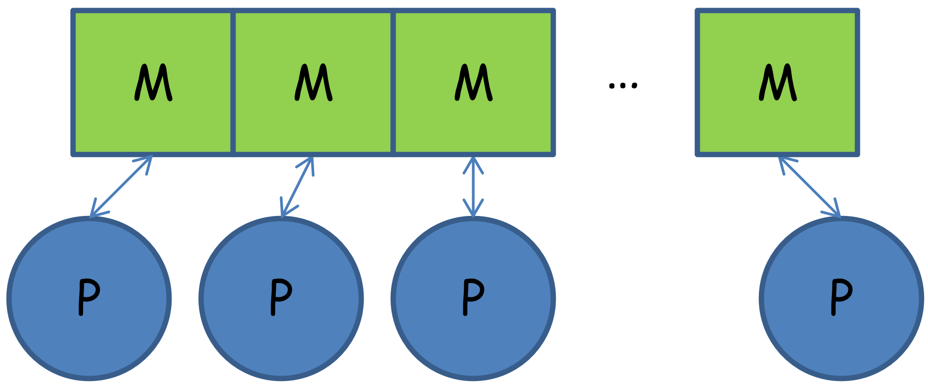


Shared-memory model

PRAM main theoretical model, introduced mid-70ties, throughout 80ties, lost interest ca. 1993

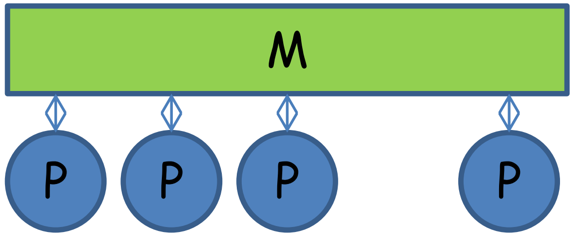
Architecture model:
Abstraction of the important modules of a computational system (processor) , their interconnection and interaction.

Used as basis for the specification of a computational model:
(formal) framework for the specification of algorithms for the computational system, including cost model.

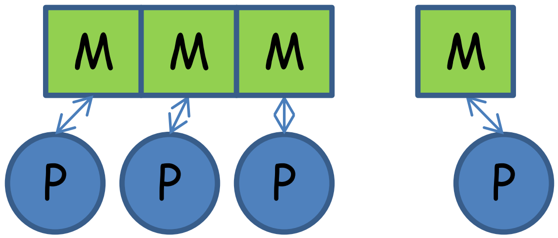


Shared-memory model

UMA (Uniform Memory Access): access time to memory location is independent of location and accessing processor, e.g. O(1), O(log M), …



NUMA (Non-Uniform Memory Access): access time dependent on processor and location. Locality: some locations can be accessed faster by a processor than others („are closer“)



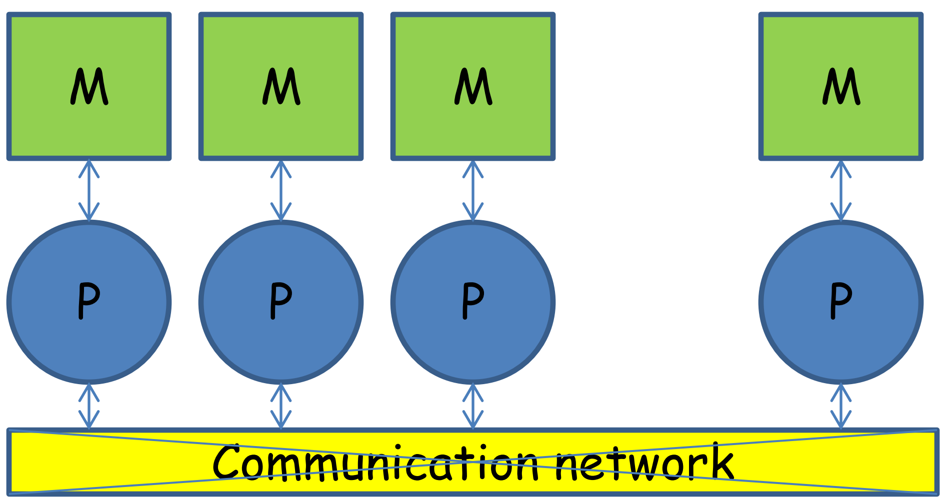
<u>Architectural model</u> defines „parallel resources“, specifies
•Power/composition of processor (ALU, FPU, registers, w-bit words vs. unlimited, Vector Unit (MMX, SSE))
•Types of instructions
•Memory system, caches
•…

<u>Execution model/cost model</u> specifies
•How instructions are executed
•(relative) Cost of instructions, memory accesses
•…

Level of detail/formality dependent on purpose: what is to be studied (complexity theory, algorithms design, …)

Architecture model:
Abstraction of the important modules of a computational system (processor) , their interconnection and interaction.

Used as basis for the specification of a computational model: (formal) framework for the specification of algorithms for the computational system, including cost model.

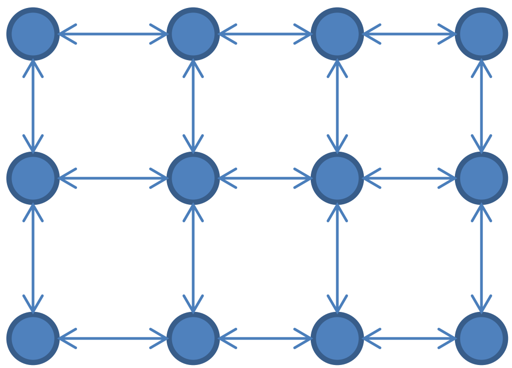


Distributed memory model

Parallel <u>architectural model</u> specifies
- Synchronization between processors
- Synchronization operations
- Atomic operations, shared resources (memory, registers)
- Communication mechanisms: network topology, properties
- …

<u>Cost model</u> specifies
- Cost of synchronination, atomic operations
- Cost of communication (latency, bandwidth, …)

Architectural model: cellular automaton, systolic array, … - simple processors without memory (finite state automata, FSA), operate in lock step on (potentially infinite) grid, local communication only



[John on Neumann, Arthur W. Burks: Theory of self-reproducing automata, 1966]
[H. T. Kung: Why systolic architectures? IEEE Computer 15(1): 37-46, 1982]. Goes back to early 70ties

Flynn's taxonomy: orthognal classification of (parallel) architectures.

Intruction stream

Data stream		
	SISD **Single Instruction Single Data**	**MISD** **Multiple Instruction Single Data**
	SIMD Single Instruction Multiple Data	MIMD Multiple Instruction Multiple Data

[M. J. Flynn: Some computer organizations and their effectiveness. IEEE Trans. Comp. C-21(9):948-960, 1972]

SISD: single processor, single stream of instructions, operates on single stream of data. Sequential architecture (e.g. RAM)

SIMD: Single processor, single stream of operations, operates on multiple data per instruction. Example: traditional vector computer

MISD: Multiple instructions operate on single data stream. Example: pipelined architectures, streaming architectures(?), systolic arrays (70ties architetural idea).

Some say:MISD class empty

MIMD: multiple instruction streams, multiple data streams

Programming model:
Abstraction close to programming language level defining parallel resources, management of parallel resources, parallelization paradigms, memory layout, synchronization and communication features, and their semantics

Parallel programming language, or library („interface“) is the concrete implementation of one (or more: multi-modal, hybrid) parallel programming models

Cost of operations: rather at level of architecture/computational model

Execution model: when and how parallelism in programming model is effected

Parallel programming model specifies, e.g.

•Parallel resources, entities, units: processes, threads, tasks, …
•Expression of parallelism: explicit or implicit
•Level and granularity of parallelism

•Memory model: shared, distributed, hybrid
•Memory semantics
•Data structures, data distribution

•Methods of synchronization (implicit/explicit)
•Methods and modes of communication

Examples:

•Threads, shared memory, block distributed arrays, fork-join parallelism
•Distributed memory, explicit message passing, collective communication, one-sided communication („RDMA“)
•Data parallel SIMD, SPMD
•...

Concrete libraries/languages: pthreads, OpenMP, MPI, UPC, TBB, ...

SPMD: Single Program, Multiple Data

[F.Darema at al.: A single-program-multiple-data computational model for EPEX/FORTRAN, 1988]

OpenMP MPI

Programming language/library/interface/paradigm

Programming model

Algorithms support

Architecture model

„Real" Hardware

Different architectures models can realize given programming model

Closer fit: more efficient use of architecture

Challenge: programming model that is useful and close to „realistic" architecture models

Challenge: language that conveniently realizes programming model

Examples:

OpenMP programming interface/language for shared-memory model, intended for shared memory systems.

Can be implemented with DSM (Distributed Shared Memory) on distributed memory architectures – but performance has usually not been good. Requires DSM implementation/algorithms

MPI interface/library for distributed memory model, can be used on shared-memory architectures, too. Often done, and makes sense...

Speeding up computations by parallel processing

p <u>dedicated, tightly coupled processors</u> collaborate to solve given problem of input size n:

Tseq(n): time for 1 processor to solve problem of size n

Tpar(p,n): time for p processors to solve problem of size n

$$\boxed{\text{Speedup}(p,n) = \text{Tseq}(n)/\text{Tpar}(p,n)}$$

Speedup measures the gain in moving from sequential to parallel computation

Speeding up computations by parallel processing

p dedicated, tightly coupled processors collaborate to solve given problem of input size n:

Tseq(n): time for 1 processor to solve problem of size n

Tpar(p,n): time for p processors to solve problem of size n

Sometimes S, SU, …

$$\boxed{\text{Speedup}(p) = \text{Tseq}(n)/\text{Tpar}(p,n)}$$

If n is fixed , or „disappears“

Speedup measures the gain in moving from sequential to parallel computation

Tseq(n), Tpar(p,n) ambiguous

- -Time for some algorithm 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, …?
- -Ignoring constants, e.g. O(f(p,n)) or 25n/p+3ln (4 (p/n))… ?

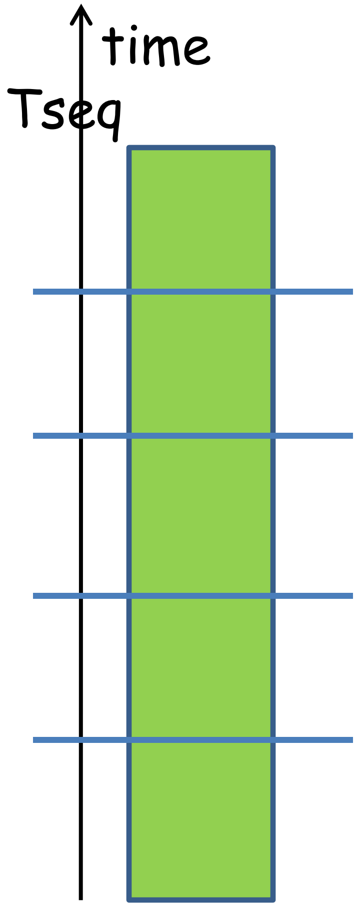
Typically: fix some (good) some algorithm, assume constants in Tseq(n) and Tpar(p,n) comparable, emphasis on orders of magnitude

Ideally: Tseq(n) time for best possible algorithm

As always in computer science, distinguish

•Problem G to be solved (mathematically specified)
•Algorithm A to solve G
•Best possible (lower bound) algorithm A* for G, best known algorithm A+ for G

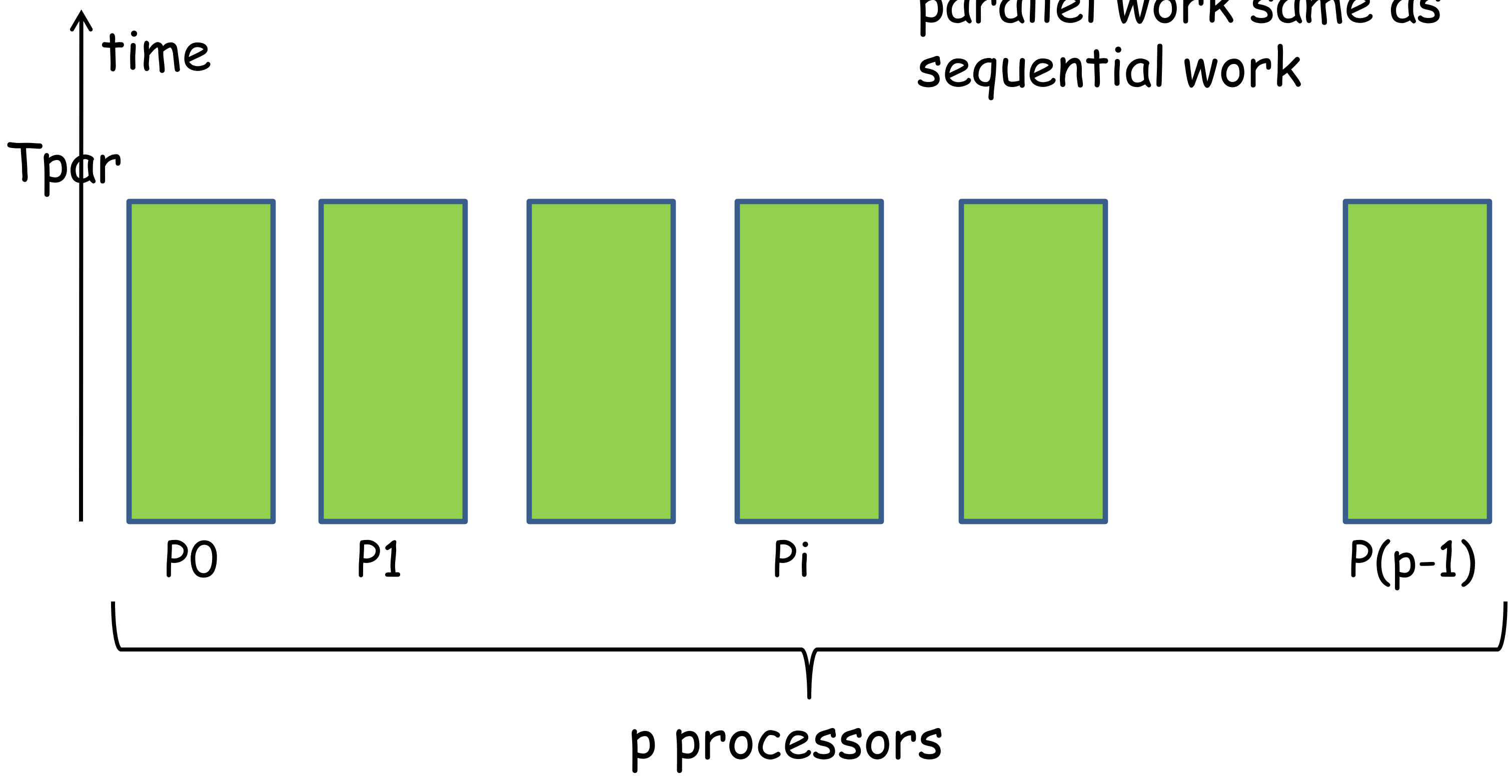
•Implementation of A on some architecture M



Parallelize: divide work into p independent pieces, assign to p processors…

Sequential time is (sequential) work

General: work is total number of instructions executed



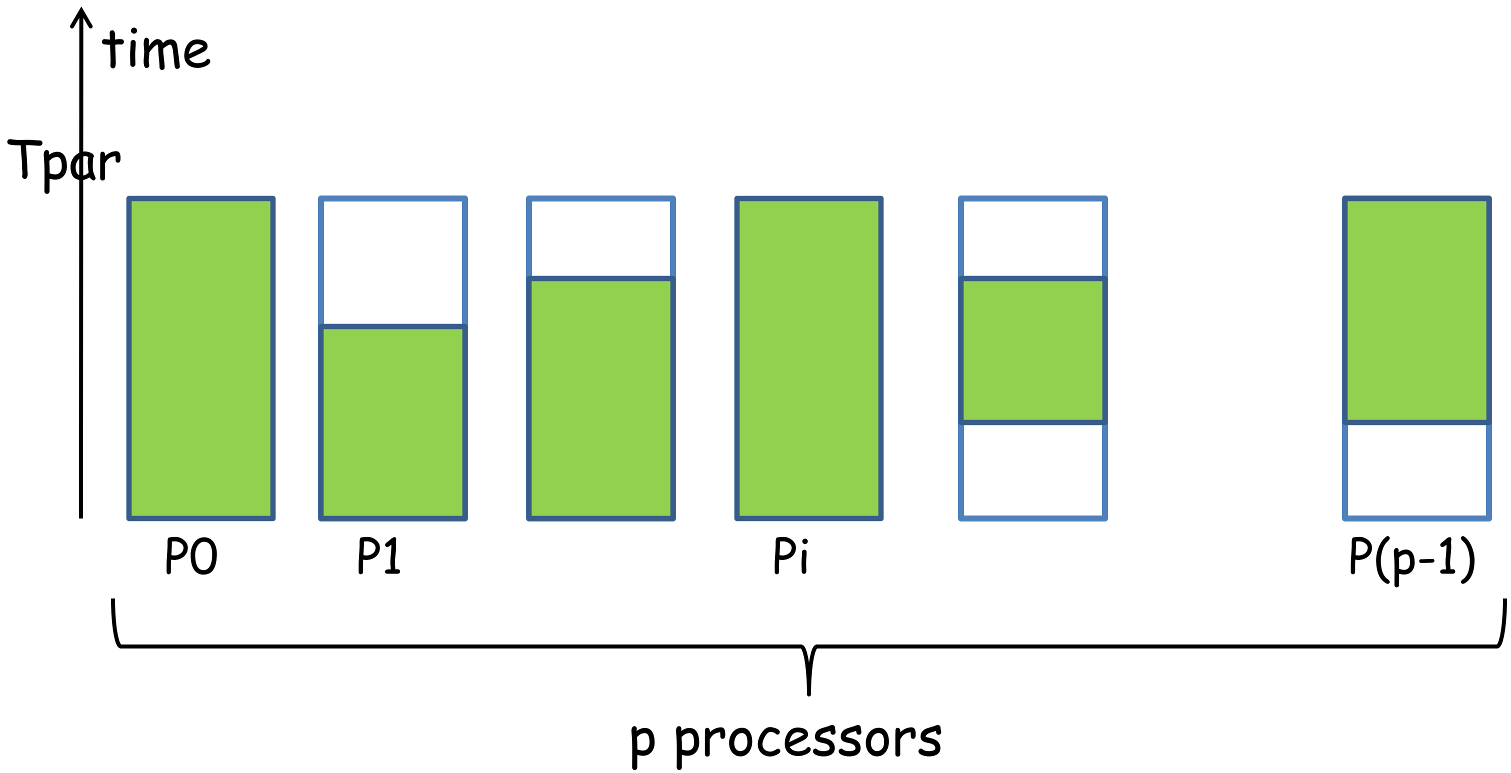
Here:
parallel work same as sequential work

Idealized, best case

"embarrassingly parallel"
"pleasingly parallel"
"perfect speedup"

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

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

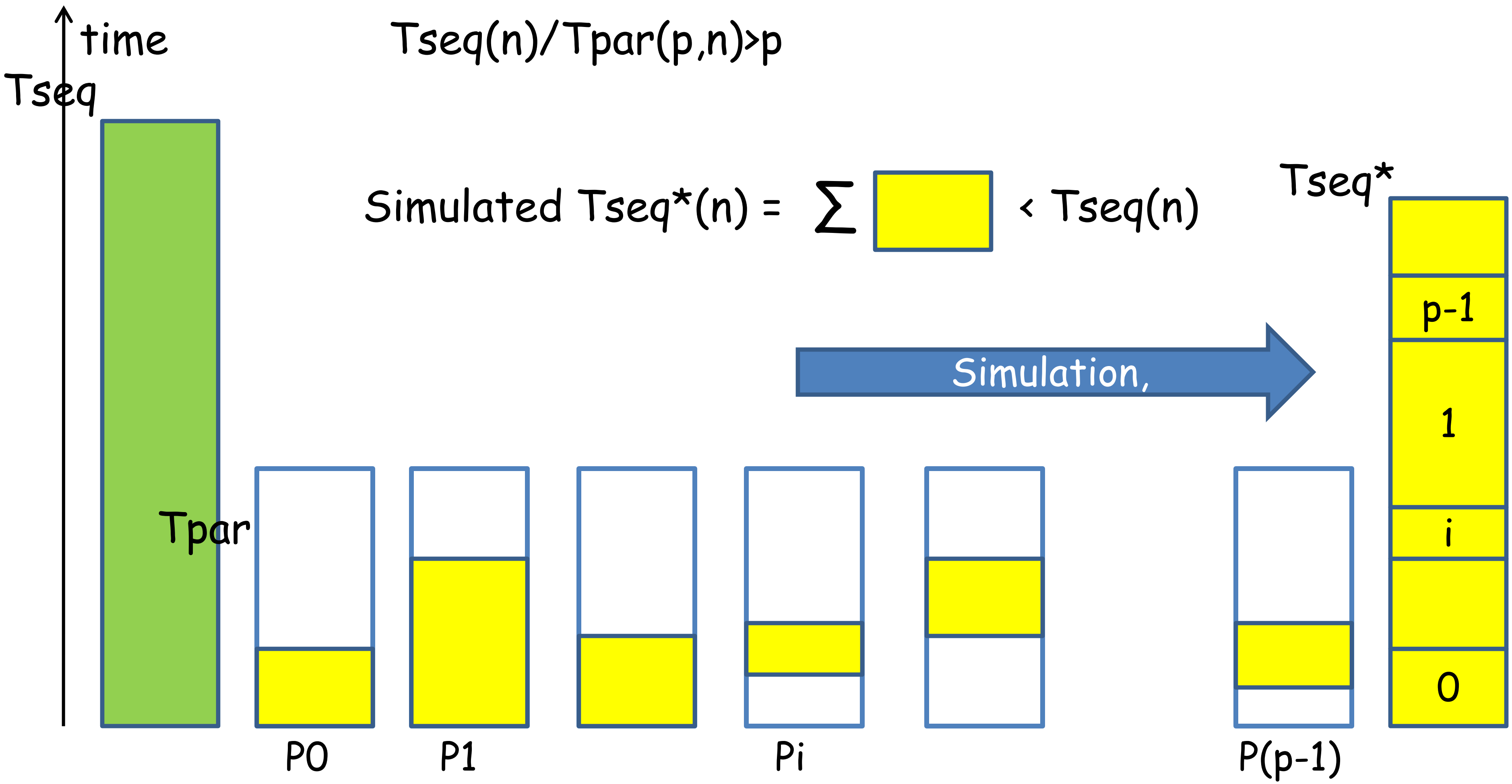


p processors assumed to start at the same time, Tpar is the time for the slowest/last processor to finish

"Theorem:"
Perfect Speedup(p,n) = p is best possible and cannot be exceeded

"Proof":
Tseq(n)/Tpar(p,n) > p implies Tseq(n) > p*Tpar(p,n), so a better sequential algorithm could be constructed by simulating the parallel algorithm on a single processor. The instructions of the p processors are carried out in some, correct order, one after another on the sequential processor.

Reminder:
Speedup is calculated (measured) relative to "best" sequential implementation/algorithm



Contradicts that Tseq(n) was best possible

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

Crucial assumptions: sequential simulation possible (enough memory to hold problem and state of parallel processors), sequential memory behaves as parallel memory, … NOT TRUE for real systems

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

[Lawrence Snyder: Type architecture, shared memory and the corollary of modest potential. Annual Review of Computer Science, 1986]

<u>Example</u>, Dumb sort, Tseq(n) = O(n^2)

that can be perfectly parallelized, Tpar(p,n) = O(n^2/p)

Well-known Tseq*(n) = O(n log n)

$\boxed{\text{Speedup(p,n) = n log n/n^2/p = (p/n) log n}}$

Linear (but low) speedup for fixed n

Break-even, when is parallel algorithm faster than sequential?

Tpar(p,n) < Tseq(n) ⇔ n^2/p < n log n ⇔ n/p < log n ⇔ p > n/log n

More processors than elements to be sorted!? Very MISD??

Lesson: Usually does not make sense to parallelize an inferior algorithm – although sometimes (much) easier

Best known/best possible parallel algorithm often difficult to parallelize
- no redundant work (that could have been done in parallel)
- tight dependencies (that forces things to be done one after another)

Lesson from PRAM theory: parallel solution of a given problem often requires a new algorithmic idea!!

But: given algorithms often have a lot of potential for easy parallelization (loops, independent functions, …), so why not?

Example: Data parallel loop of independent operations

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

Parallelize: break into p independent iteration blocks

f(i) depends only on i, no side effects, no global variables

Processor j, $0\leq j<p$

```
for (i=n[j]; i<n[j+1]; i++) {
   a[i] = f(i);
}
```

Parallelism explicit:

$n[j] = j*(n/p)$

assuming p divides n

Data Parallelism (SIMD programming model):
"p processors do same work on different data"

Example: Data parallel loop of independent operations

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

Parallelize: break into p independent iteration blocks

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

Parallelism implicit/less explicit:

Found in many models/interfaces: compiler divides iteration space, run-time schedules blocks of iterations to processors, by language construct compiler can make necessary independence assumptions

Example: Data parallel loop of independent operations

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

Parallelize: break into p independent iteration blocks

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

Parallelism implicit/transparent

Automatic parallelization: compiler detects that iterations are independent, automatically divides iteration space, interacts with run-time

Example: Data parallel loop of independent operations

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

Parallelize: break into p independent iteration blocks

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

Parallelism implicit/transparent

Automatic parallelization: can work in cases where dependency analysis is sufficient/possible, fails generally

Example: loop of dependent operations: a[i] <- a[i-1]+a[i]+a[i+1]

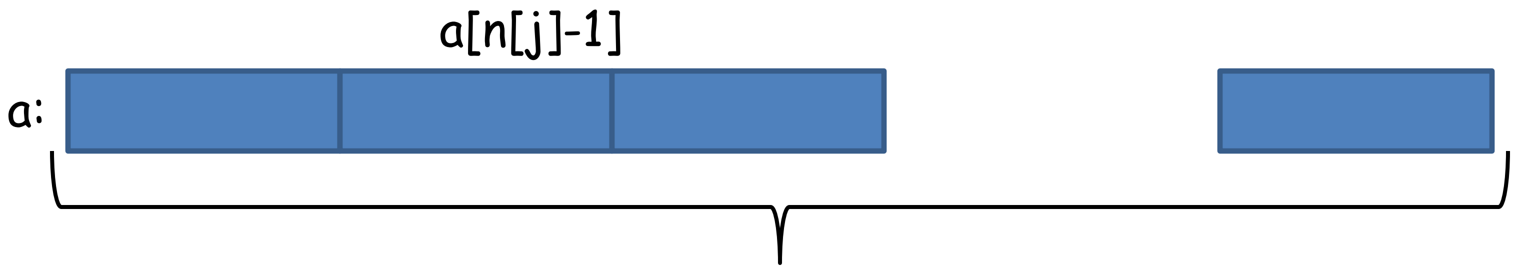
```
for (i=0; i<n; i++) {
  b[i] = a[i-1]+a[i]+a[i+1];
}
for (i=0, i<n; i++) {
  a[i] = b[i];
}
```

Processor j, 0≤j<p

```
for (i=n[j]; i<n[j+1]; i++) {
  b[i]  =a[i-1]+a[i]+a[i+1];
}
for (i=0, i<n; i++) {
  a[i] = b[i];
}
```

What about a[n[j]-1]?

Communication or synchronization needed

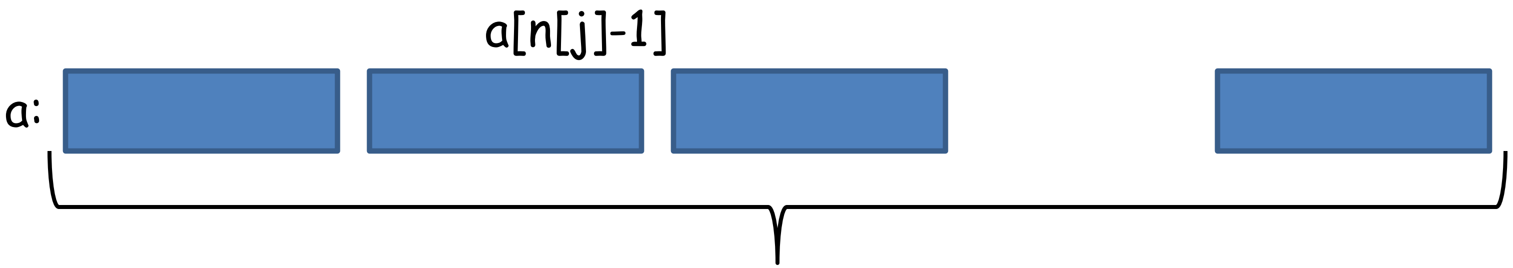


Array logically divided into p disjoint blocks

Shared memory programming model: all data can be accessed by all processors

- Memory model: when are data are data „visible“
- Memory cost model: same cost of access of all a[i]? NUMA, UMA?

- Synchronization



Array logically divided into p disjoint blocks

Distibuted memory programming model: data are local to processors

- Communication
- Cost of communication

Example:

```
for (i=0; i<n; i++) {
  switch (i%D) {
  case 0: task1(a[i]); break;
  case 1: task2(a[i]); break;
  ...
  case D-1: taskD(a[i]); break;
  default:
  }
}
```

Processor j, $0 \leq j < p$

```
for (i=0; i<n; i++) {
  if (i%D==j) taskj(a[i]);
}
```

Task/control parallelism:
„D different operations (tasks) on different data“

Example:

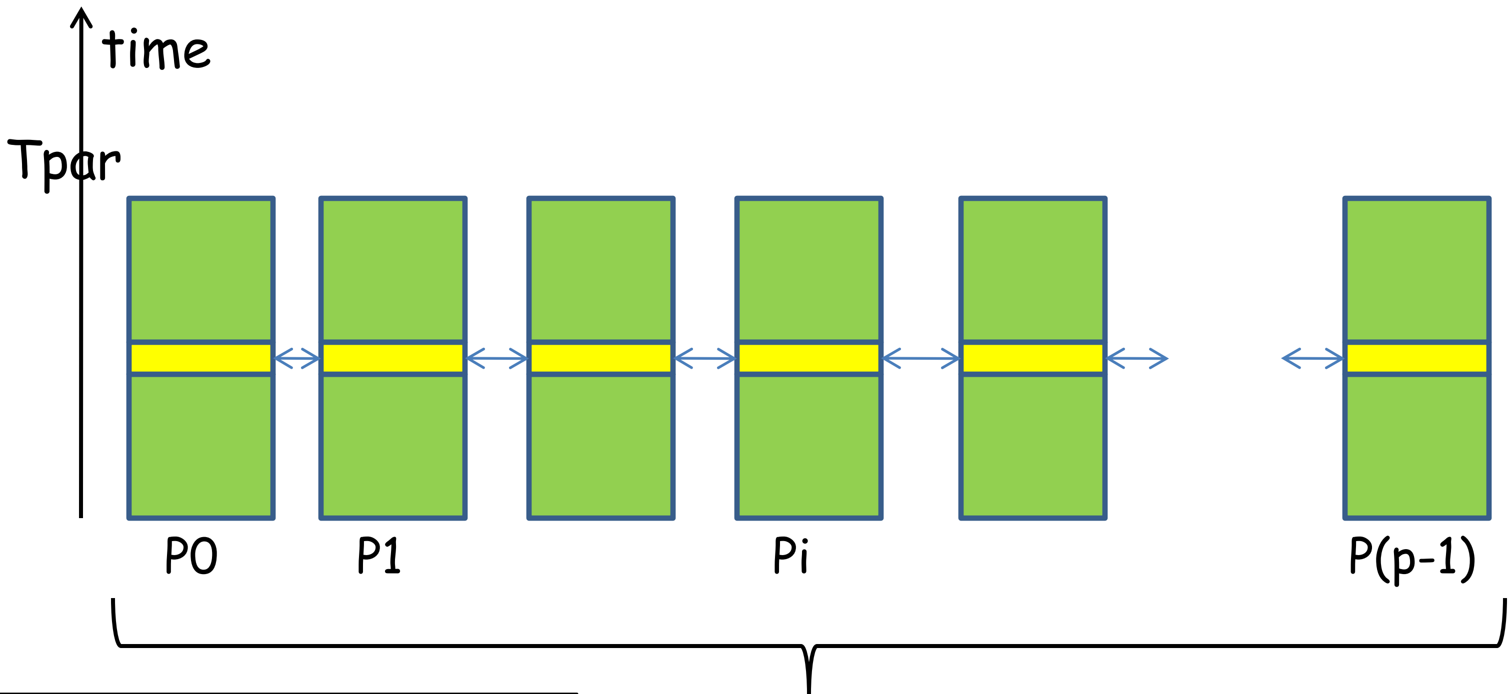
```
for (i=0;i<n;i++) {
   stage1(a[i]);
   stage2(a[i-1]);
   stage3(a[i-2]);
   ...
   stageS(a[i-S]);
}
```

Processor j, $0 \le j < p$

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

Synchronization needed: stage j on a[i] cannot start before stage j-1 on a[i] has completed

Pipeline parallelism:
„S different operations (stages) on same data“

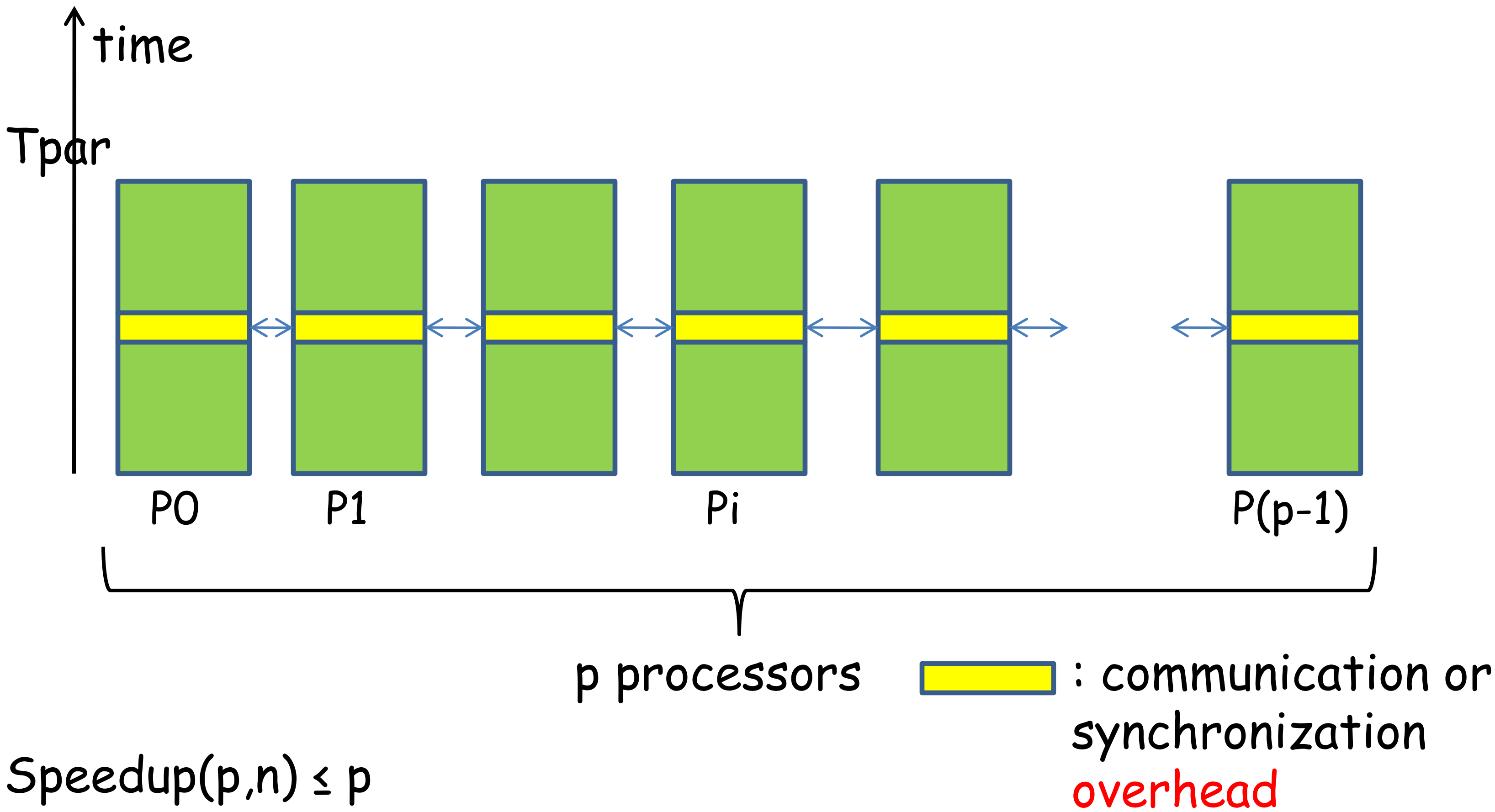



```
for (i=n[j]; i<n[j+1]; i++) {
   b[i]  =a[i-1]+a[i]+a[i+1];
} sync;
for (i=0, i<n; i++) {
   a[i] = b[i];
}
```

p processors

Processor j, 0≤j<p

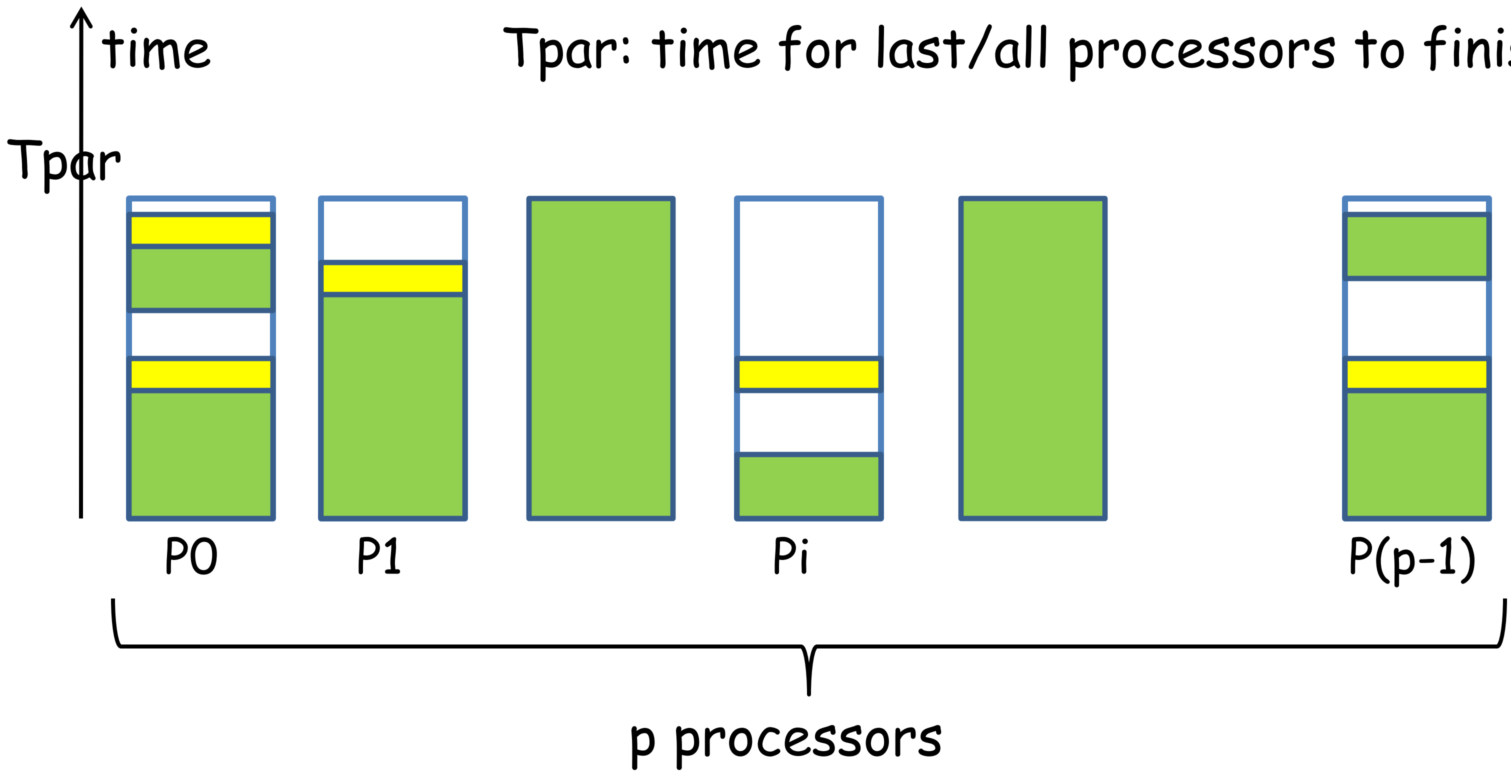
: communication or synchronization overhead



Speedup(p,n) ≤ p

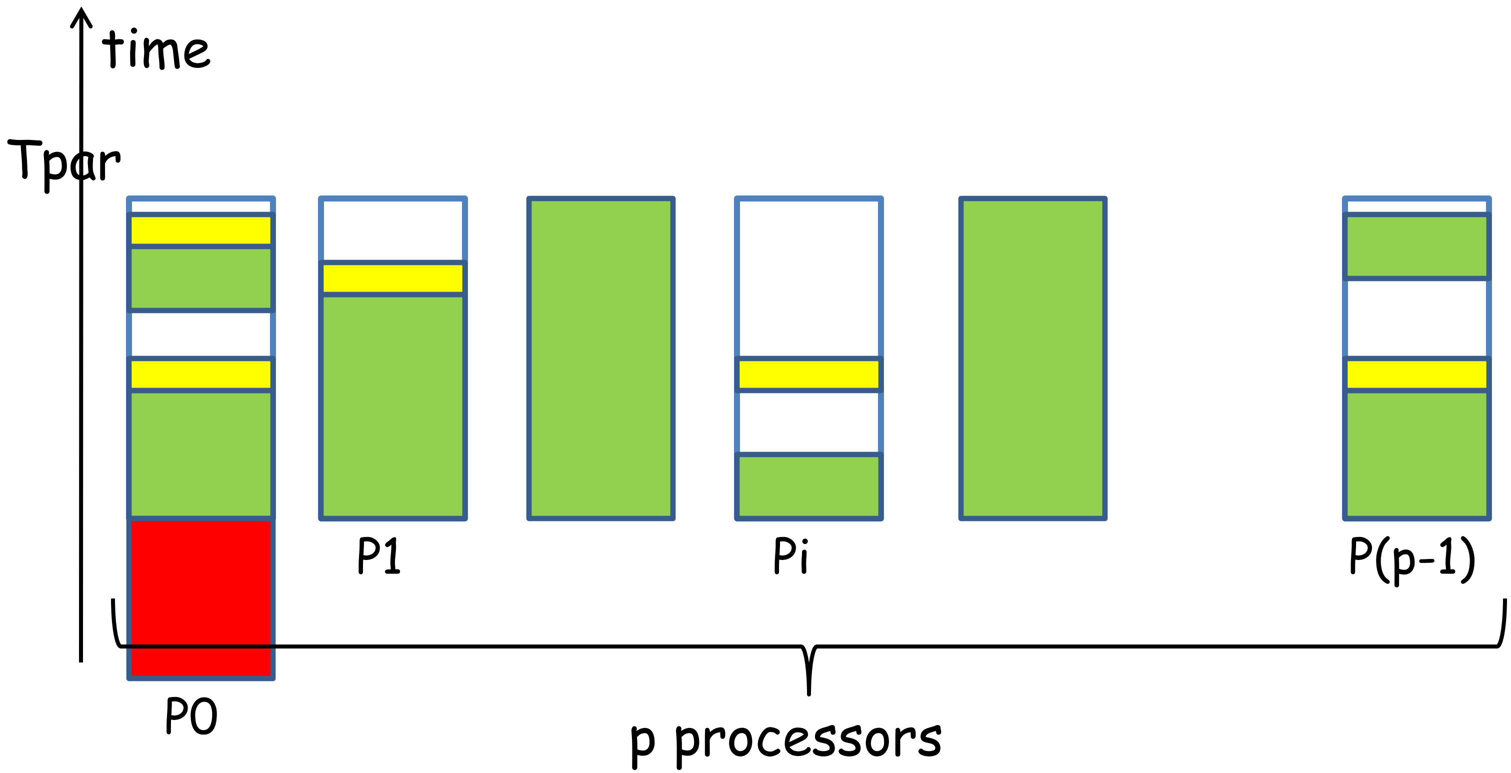
Linear speedup may still be possible, until overhead starts to dominate

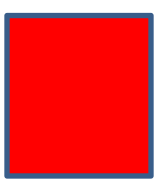
Tpar: time for last/all processors to finish



Tpar(p,n):

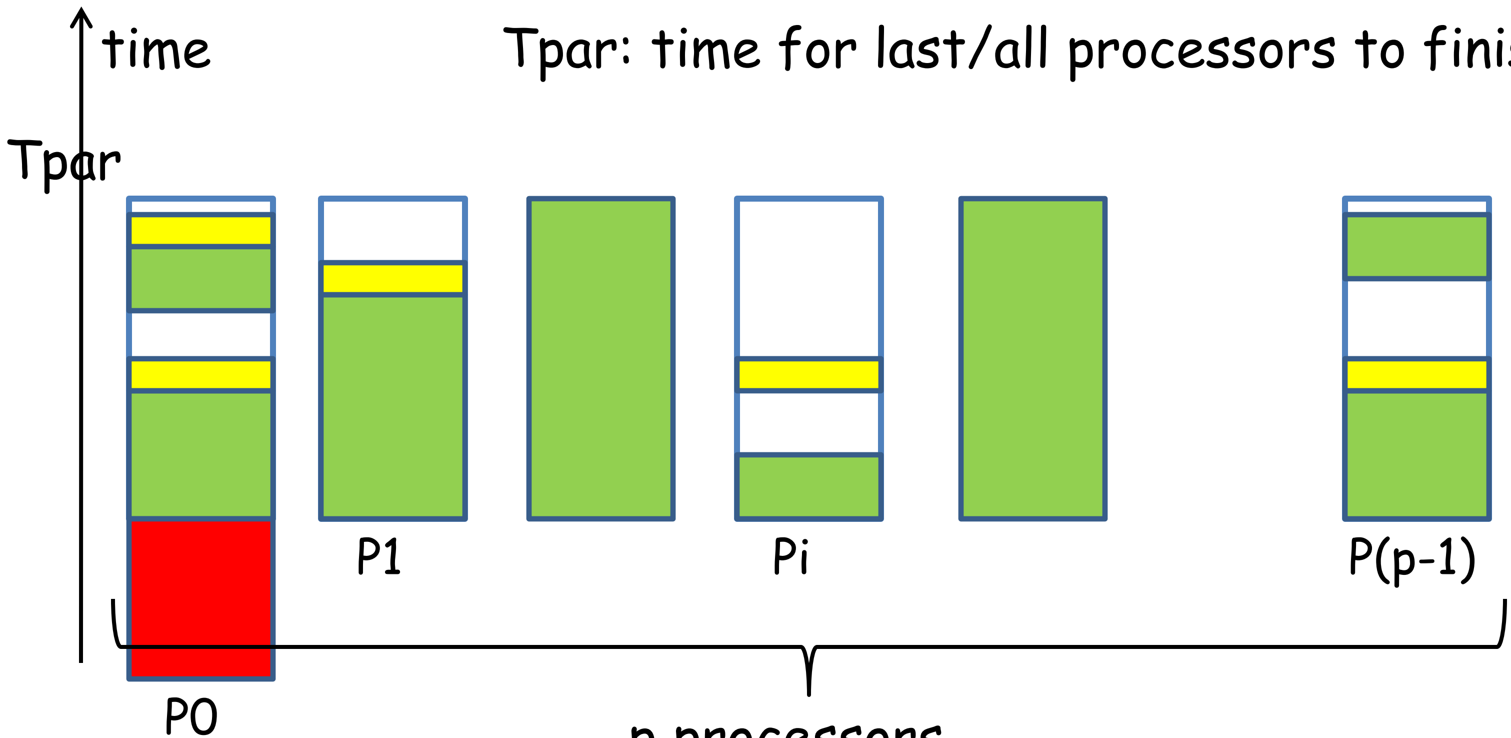
useful computational work + parallelization overhead + idle time





Algorithms/programs typically have a sequential part that cannot be parallelized: initialization of data structures, distribution of data, …

Tpar: time for last/all processors to finish



Tpar(p,n):
sequential work + useful computational work + parallelization overhead + idle time

Amdahls Law (parallel version):
Let a program A 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. For any fixed n, the maximum achievable speedup is 1/s

[G. Amdahl: Validity of the single processor approach to achieving large scale computing capabilities. AFIPS 1967]

Proof:

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

Tpar(p,n) = s*Tseq(n) + r*Tseq(n)/p

Speedup(p,n) = Tseq(n)/(s*Tseq(n)+r*Tseq(n)/p) = 1/(s+r/p) -> 1/s for p -> ∞

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…

Tseq(n) = n+K+Kn

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

Sequential fraction ≈ 1/(1+K)

Speedup(p,n) -> 1+K

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)
```

Speedup(p,n) -> 1+n

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

Tseq(n) = 1+K+Kn

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

Sequential fraction ≈ 1/(1+n)

Note:

If sequential part is constant (not fraction), Amdahl's law 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)
```

Speedup(p,n) -> 1+n

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

Tseq(n) = 1+K+Kn

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

Sequential fraction ≈ 1/(1+n)

Lesson: be careful with system functions (calloc, malloc)

Definition: parallel efficiency

$$E(p,n) = \text{Speedup}(p,n)/p = \text{Tseq}(n)/(p*\text{Tpar}(p,n))$$

Ratio of Speedup to best possible

- $E(p,n) \leq 1$
- $E(p,n) = c$: linear speedup

Scalability definitions:

A parallel algorithm/implementation is strongly scaling if Speedup(p,n) = $\Theta(p)$ (linear,independent of n)

A parallel algorithm/implementation is weakly scaling if there is a slow-growing o(1) function f(p), such that for n = $\Omega(f(p))$ E(p,n) is constant

„Efficiency maintained by increasing problem size as f(p) or more“

[J. Gustafson: Reevaluating Amdahls Law. CACM 1988]

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)
```

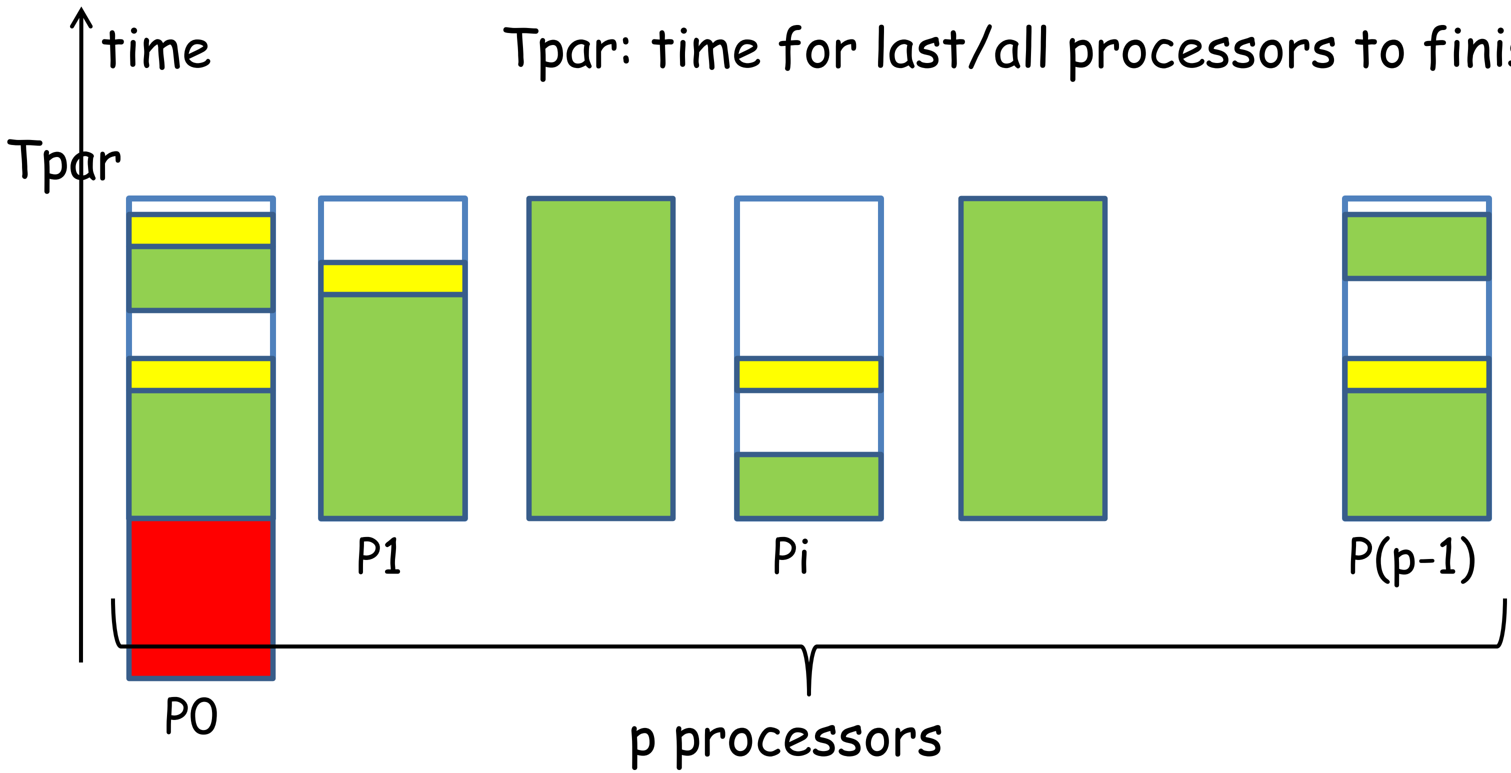
Assume convergence check takes O(log p) time

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

E(p,n) ≈ Kn/(Kn+pK log p)

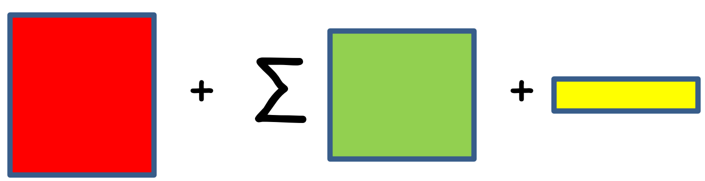
For n≥plog p, E(p,n) ≥ 1/2

Weakly scalable, n has to increase as O(p log p) to maintain constant efficiency – and as O(log p) per processor



Parallel work: sum of necessary, useful work of all processors

Wpar(p,n) =



Definition:
An algorithm/implementation is work-optimal if

$$Wpar(p,n) = O(Tseq(n))$$

Total parallel work (number of instructions over all processors) comparable to number of instructions of best sequential algorithm

Define

$$Tfast(n) = Tpar(\infty,n) = \min Tpar(p,n),\ p=1,2,\ldots$$

Fastest time that can be achieved assuming enough processors

If $Wpar(p,n)$ can be distributed evenly over the p processors, then

$$Tpar(p,n) = max(Wpar(p,n)/p, Tfast(n))$$

and

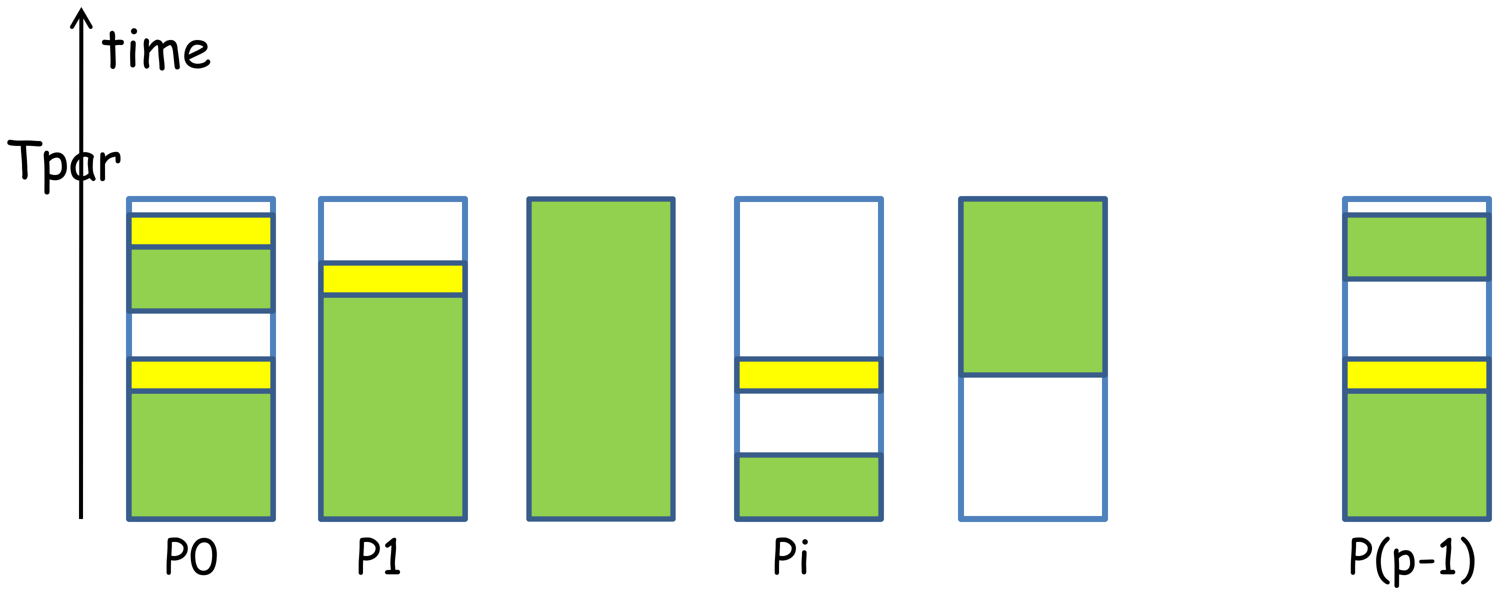
$$Speedup(p,n) = Tseq(n)/Wpar(p,n)/p = p/c$$

as long as $Wpar(p,n)/p \geq Tfast(n)$, for some constant c

Theorem:
Work-optimal implementations/algorithms can have linear speedup for $p \leq Wpar(p,n)/Tfast(n)$

- provided the work can be distributed evenly

Dividing the work Wpar(p,n) into even sized chunks is called load balancing. Often not trivial. Can sometimes be done statically, sometimes dynamically, then often called scheduling. Assigning the work to processors is called mapping. Also not trivial.

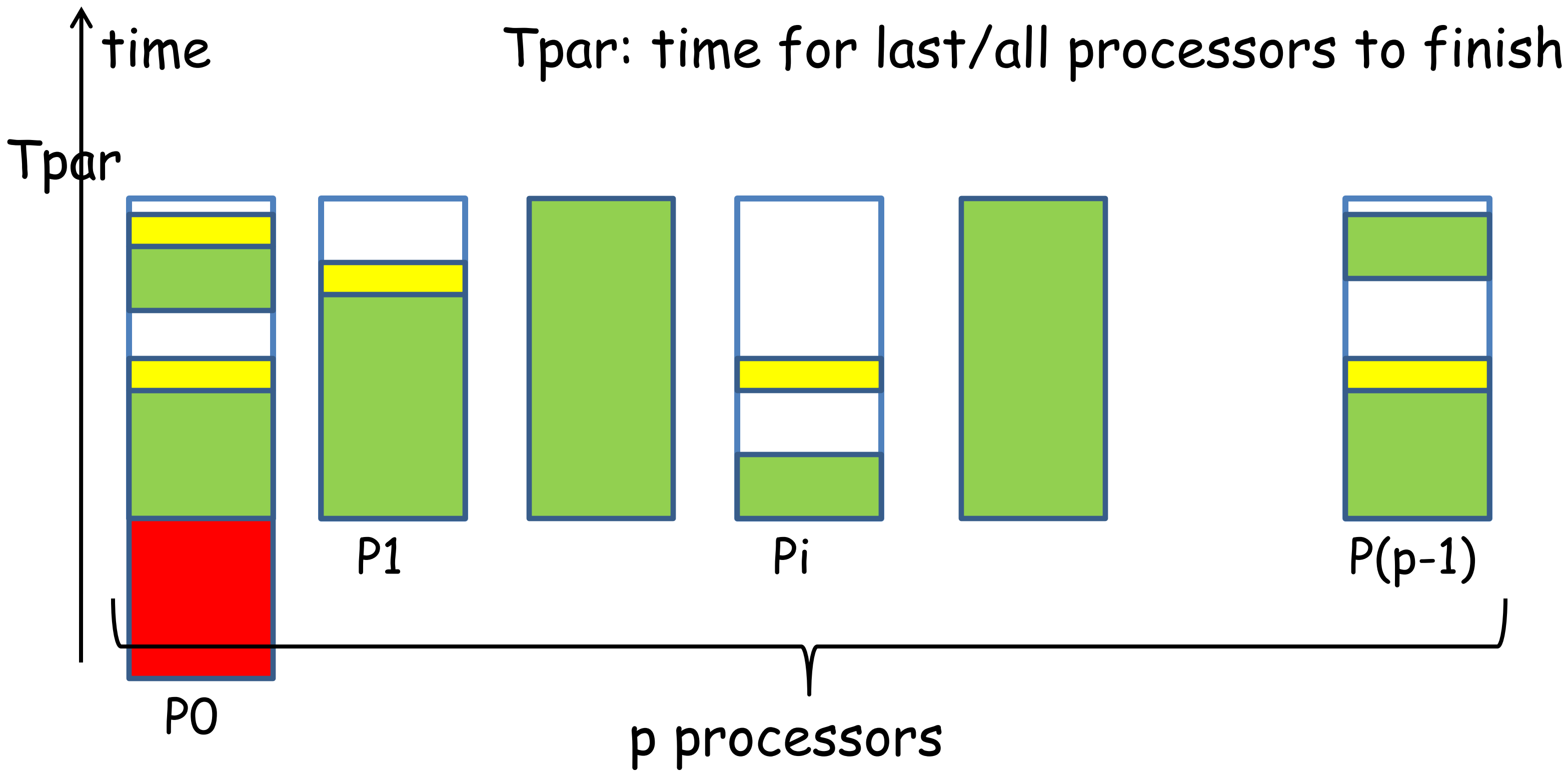


WT presentation framework (Work-Time, Work-Depth):

- Determine total work of parallel algorithm, $W(n)$
- Determine fastest time possible = longest chain of dependent operations = $Tfast(n)$ = „depth" d of parallel algorithm

- Assuming $W(n)$ can be distributed over the p processors, parallel performance is $O(W(n)/p+d)$

Introduced by Shiloach, Vishkin ca. 1982, often used, e.g. [JaJa: Introduction to Parallel Algorithms, 1992], [Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms, 3rd ed, 2009]



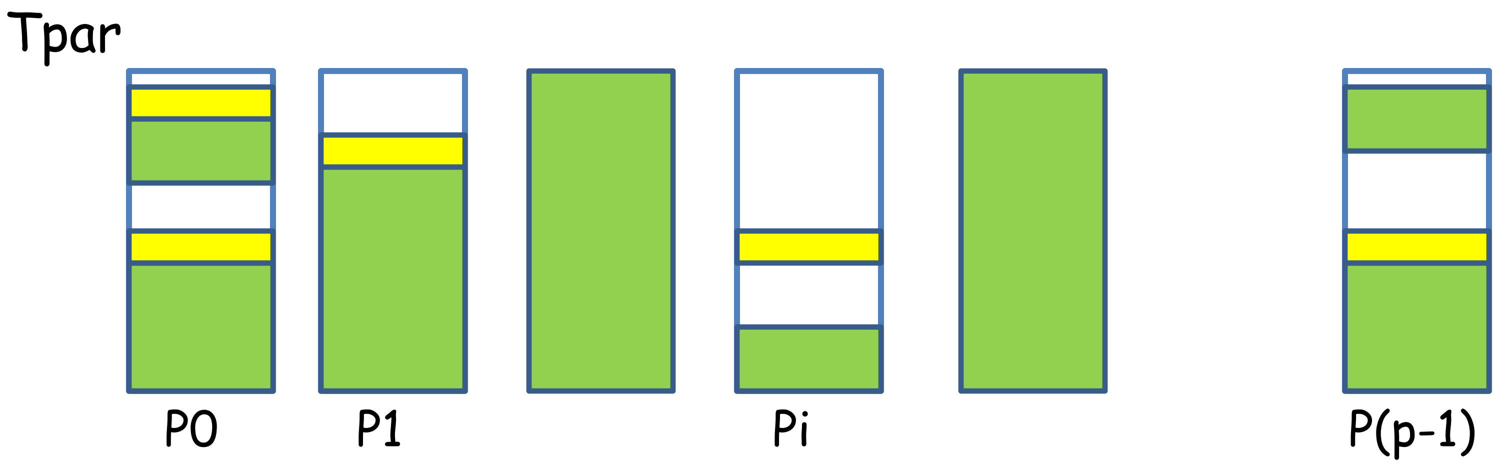
Cost: p*Tpar(p,n)

Dedicated parallel resources: p processors reserved for Tpar(p,n) time

Definition:
An algorithm/implementation is cost-optimal if

$$p*Tpar(p,n) = O(Tseq(n))$$

No idle time, work can actually be distributed over the p processors, optimally load balanced



Overhead is cost minus sequential work

$$\text{Overhead} = p*\text{Tpar}(p,n)-\text{Tseq}(n)$$

Overheads: extra work, synchronization, communication, idle time/load imbalance

Theorem:
Cost-optimal algorithms have constant efficiency and overhead $O(1)$

$$E(p,n) = Tseq(n)/p*Tpar(p,n) = Tseq(n)/c*Tseq(n) = 1/c$$

for some constant c hidden in $O(Tseq(n))$

Parallelization: a first example

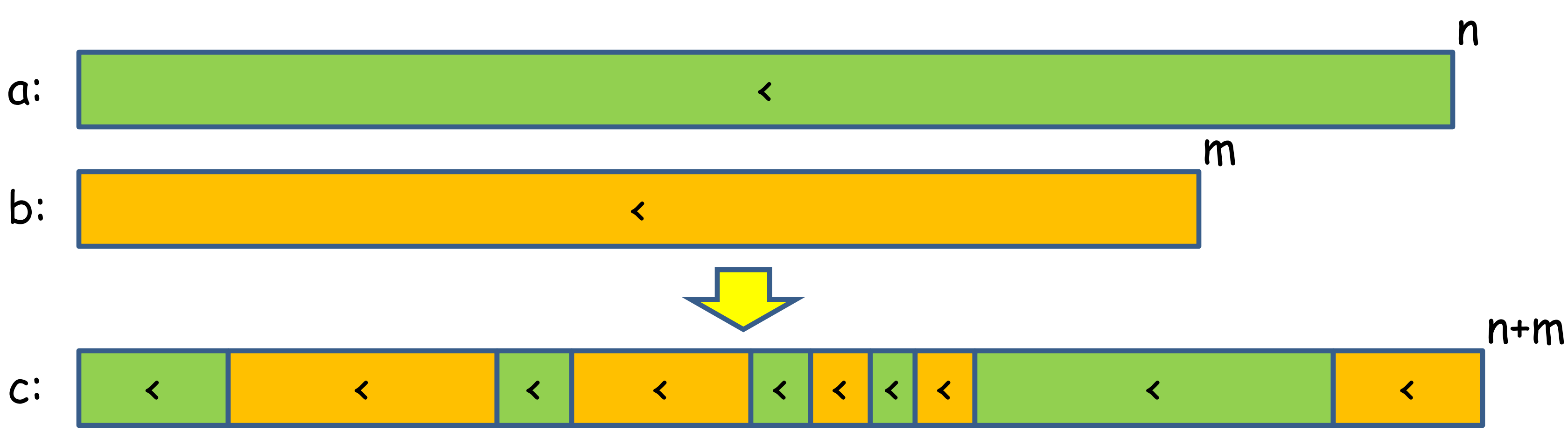
Problem:
given two ordered sequences (x_i), $i=0,\ldots,n-1$, and (y_i), $i=0,\ldots,m-1$ stored in arrays A and B, merge the two sequences into a single, ordered sequence (z_i), $i=0,\ldots,m+n-1$, stored in array C such that $z_i=x_k$ or $z_i=y_k$ for some k, and for each x_i and y_i there is a $z_k=x_i$ and $z_k=y_k$

(Tedious formulation of) Well-known, and useful problem.
For simplicity, assume that all x_i and y_i are distinct

Standard strictly sequential solution:

```
i = 0; j = 0; k = 0;
while (i<n&&j<m) {
  c[k++] = (a[i]<b[j]) ? a[i++] : b[j++];
}
while (i<n) c[k++] = a[i++];
while (j<m) c[k++] = b[j++];
```

Tseq(n+m) = (n+m)



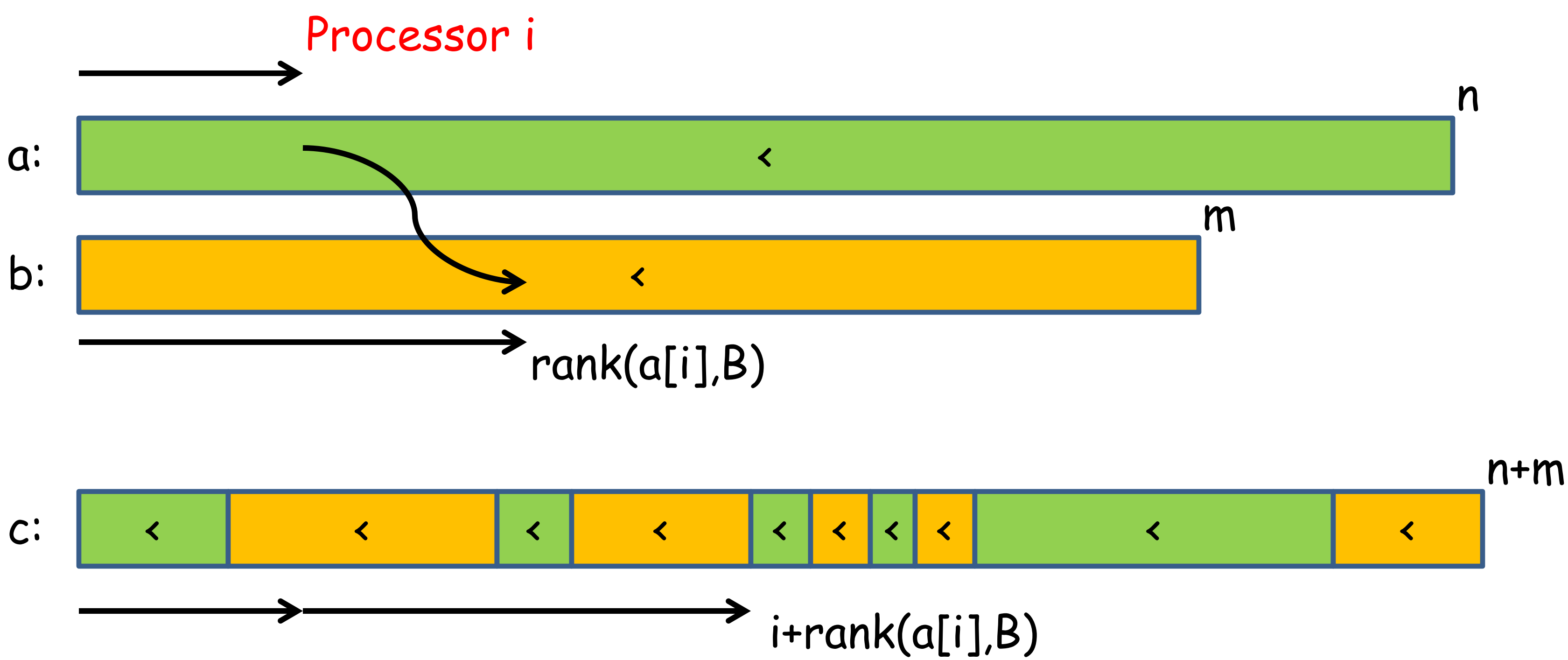
Parallel solution?

Assumption 1:
p independently working, „parallel" processors. All processors have access to the full input and random access to the output array: explicit, shared-memory programming model

Strategy:
Find a way to divide the merging steps evenly and independently between the p processors.

Solution 1:
Restricted to $p=n+m$ processors (as many processors as elements in the input array)

Definition: element x, set A not containing x, $rank(x,A)$ is the number of elements in A smaller than x




```
if (i<n) c[i+rank(a[i],B)] = a[i];
else if (i<n+m) {
  j = i-n;
  c[j+rank(b[j],A)] = b[j];
}
```

for processor i,
0≤i<n+m

Observation: for an ordered sequence stored in an array A, rank(x,A) can be computed by binary search!

Number of operations is O(log n) for an n-element array A

Tpar(n+m,n+m) = O(log (max(m,n))

Exponential improvement in time, with linear number of processors!!

Work = O((m+n)log(max(n,m)) ≤ O(2nlog n) = O(n logn)

The algorithm is not work efficient, Speedup(p) = p/log p

Problems:

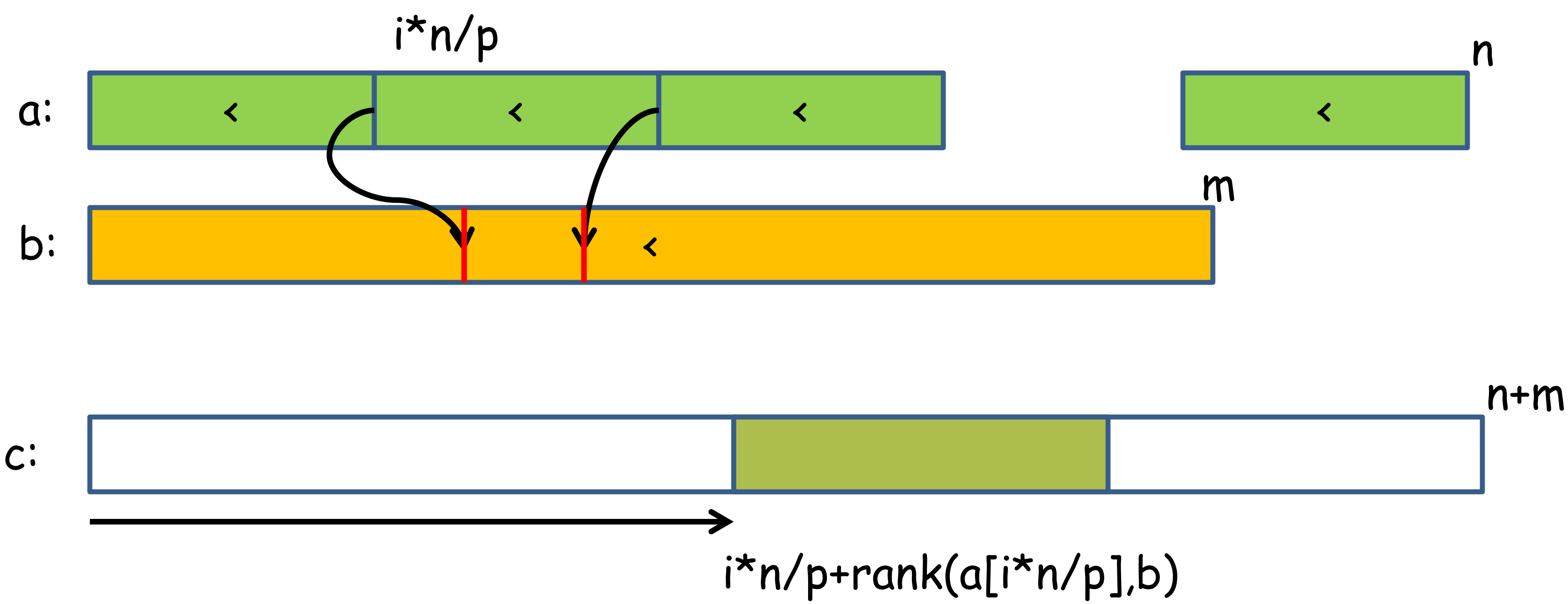
- Algorithm is not efficient
- Normally, n>>p

- When is the computation done (are processes synchronized?)?

```
if (i<n) c[i+rank(a[i],B)] = a[i]; else if
(i<n+m) {
  j = i-n;
  c[j+rank(b[j],A)] = b[j];
}
barrier; // synchronization construct
```

Done!

Solution 2:
Divide a into p blocks of size approx. n/p, rank only first element of each block, in parallel merge blocks of a with blocks of b sequentially



Processor i, 0≤i<n

```
merge(&a[i*(n/p)],n/p,
      &b[rank(a[i*(n/p)],b)],
      rank(a[(i+1)*(n/p)],b)]-rank(a[i*(n/p)],b),
      &c[i*(n/p)+rank(a[i*(n/p)],b)]);
barrier;
```

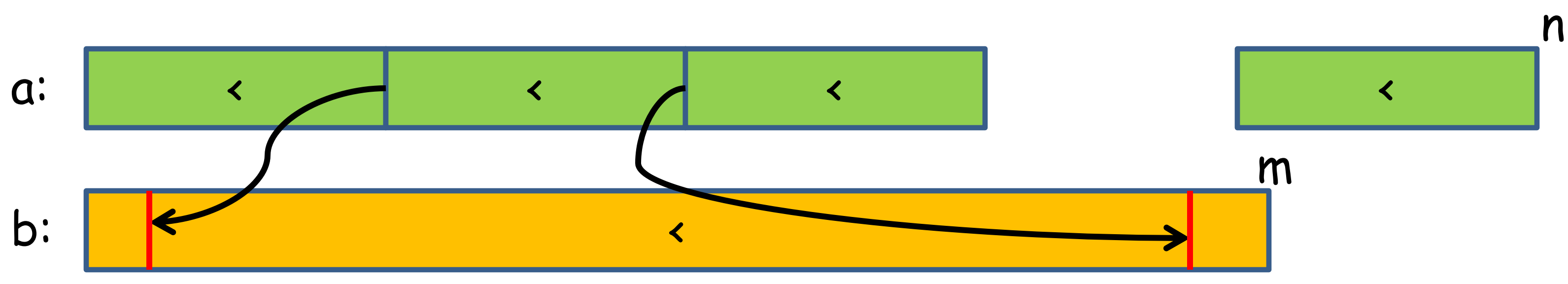
`merge(a,n,b,m,c)`: merges a of size n and b of size m into c

Structure:
•Parallel preprocessing – rank: binary search - to divide problem into p independent pieces
•Sequential algorithm to process subproblems in parallel

Work optimal: Work = p log m + p*(n/p)+m = p log m + (n+m) = O(n+m)

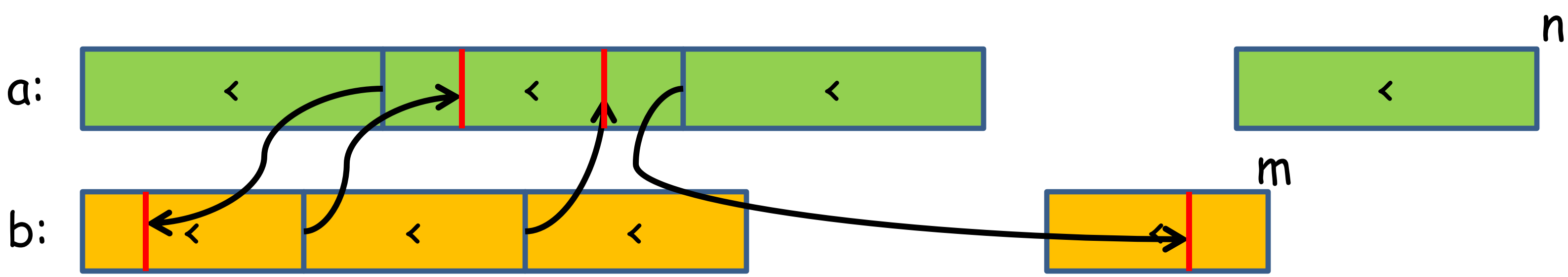
Problems:

•Assumed that p divides n
•Severe load imbalance in worst case



One processor does almost all work O(n/p+m), time is O(n/p+m+log n)

<u>Solution 3</u>:
Divide a into p blocks of size approx. n/p, rank only first element of each block, and divide b into p blocks of size approx. m/p; in parallel merge blocks of a with blocks of b sequentially



2p smaller merge problems, but all O(n/p+m/p). Shown by case analysis

Theorem:
On a shared-memory system, two ordered sequences of size n and m can be merged in time $O((n+m)/p+\log n)$

Exercise:
Implement, test and benchmark the merge algorithm in pthreads or OpenMP

Parallel Computing

Parallelization (of merge problem):

- Focus on the problem
- Parallel work comparable to sequential work
- Consider potential for parallelization of known sequential algorithm
- Look for good load balance
- Minimize synchronization points
- (Communication: not yet seen)
- Sequential algorithms as subalgorithms

Automatic parallelization???

Foster's methodology:

1. Partitioning: divide the computation into independent tasks
2. Communication: determine communication needed between tasks
3. Agglomeration/aggregation: combine tasks and communications together into larger (independent) chunks
4. Mapping: assign tasks and communications to processes, threads, …

Rule of thumb, not always applicable (architecture dependent: what is the best granularity of „tasks")

There is no recipe for parallelizing a problem or an algorithm!

[Ian Foster: Designing and building parallel programs. 1995]