

Introduction to Parallel Computing

Introduction, architectures, performance

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 ubiquitous (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 cooperate 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

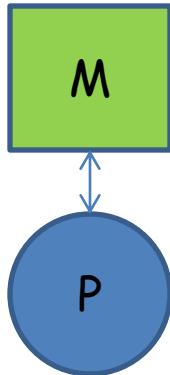
(Traditional) concern/objective: how to solve a given computational problem faster

- 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?
- ...

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)

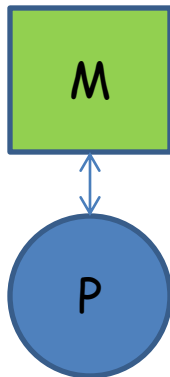
Processor (ALU, PC, registers) capable of executing instructions stored in memory on data in memory

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)

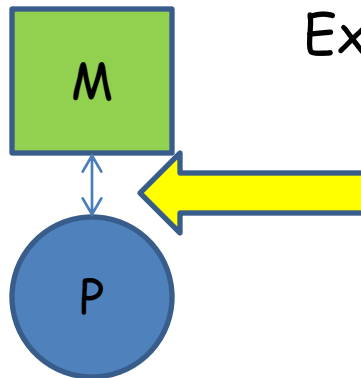
Aka **von Neumann** architecture, stored program computer (contrast: finite state automaton)

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

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)

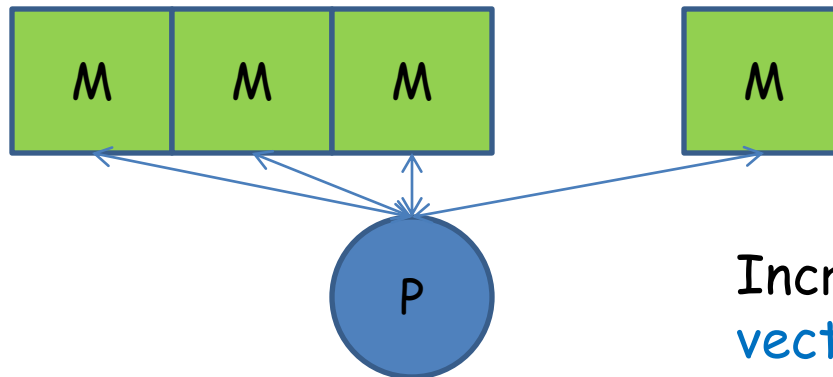
„**von Neumann bottleneck**“: program and data separate from CPU, processing rate limited by memory rate.

[John Backus, Turing Award Lecture, 1977]

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.

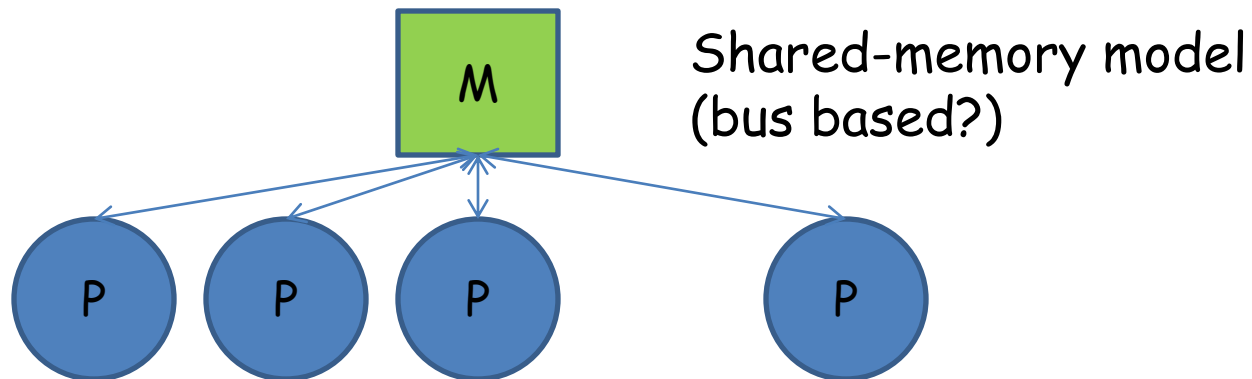


Increased memory rate,
vector computer, ALU
operates on vectors
instead of scalars

Architecture model:

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

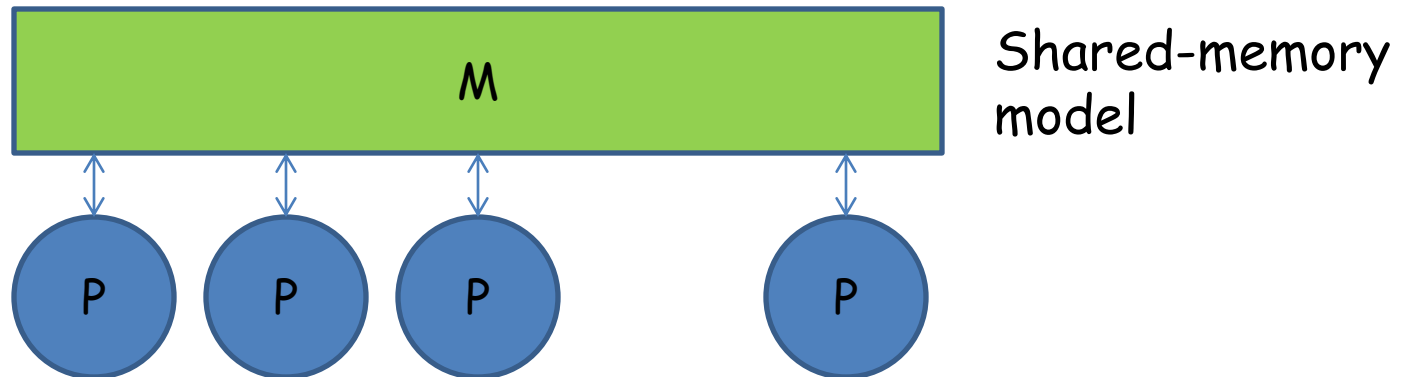
Used as basis for the specification of a computational model:
(formal) framework for the specification of algorithms for the computational system, including cost model.



Architecture model:

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

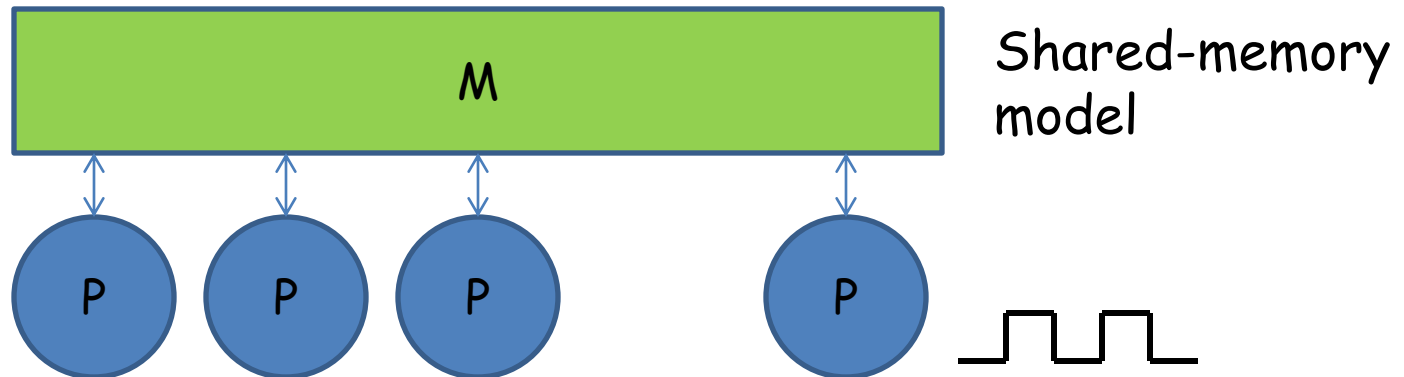
Used as basis for the specification of a computational model:
(formal) framework for the specification of algorithms for the computational system, including cost model.



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.

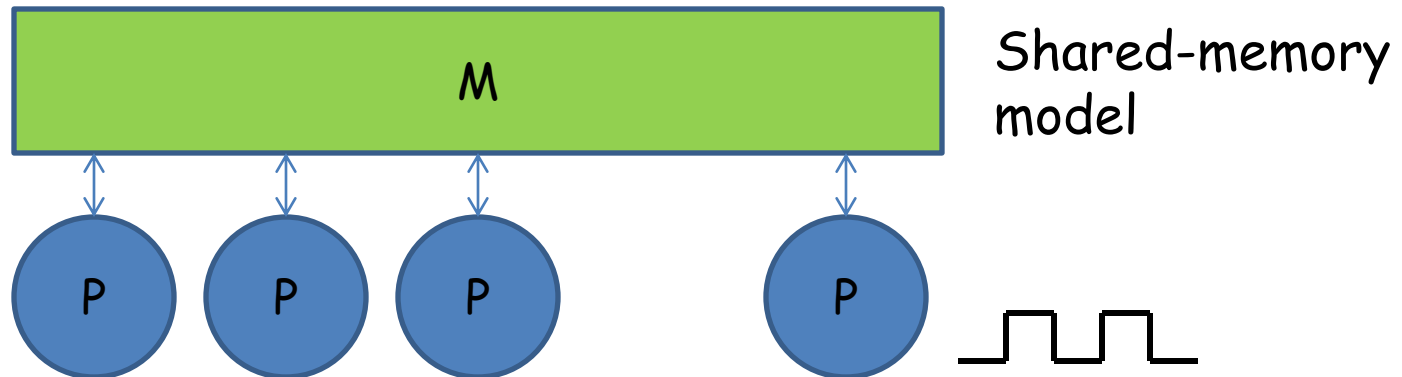


Processors operate in lock-step, uniform memory access time = instruction time: Parallel RAM (PRAM)

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.

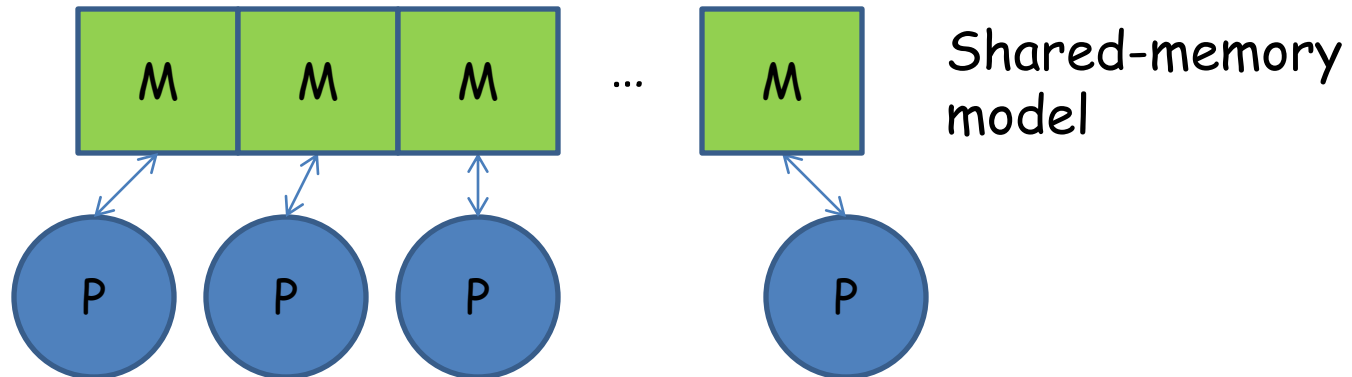


PRAM main theoretical model, introduced mid-70ties, throughout 80ties, lost interest ca. 1993

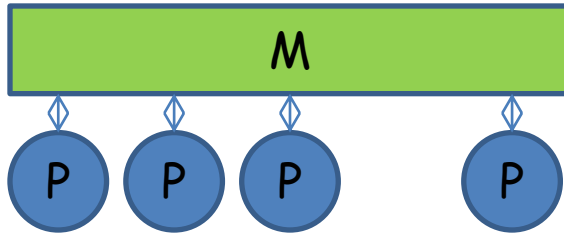
Architecture model:

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

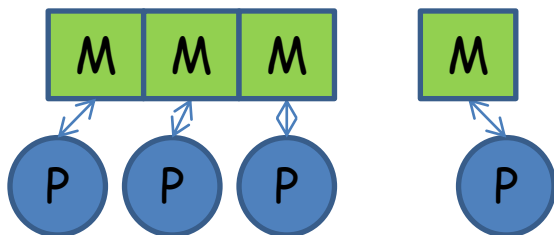
Used as basis for the specification of a computational model:
(formal) framework for the specification of algorithms for the computational system, including cost model.



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“)



Architectural model 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
- ...

Execution model/cost model specifies

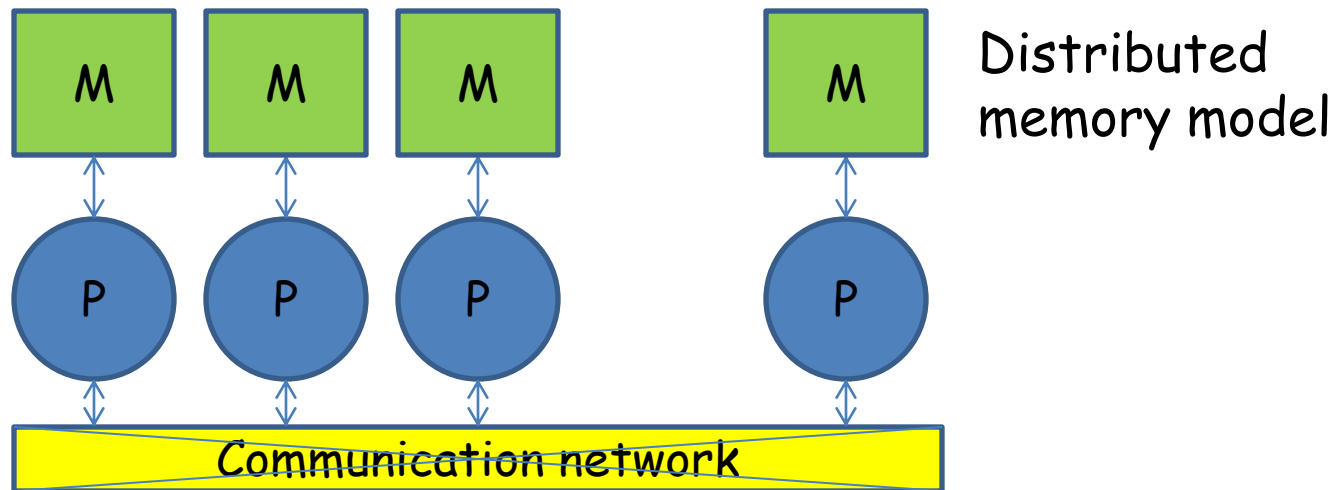
- How instructions are executed
- (relative) Cost of instructions, memory accesses
- ...

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.



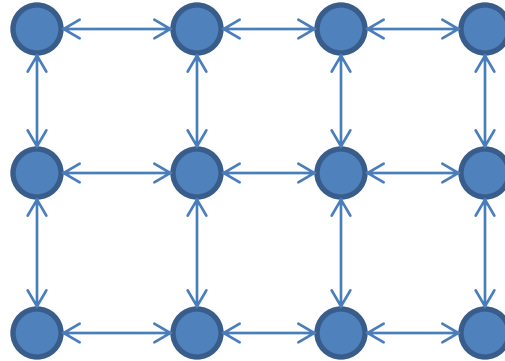
Parallel architectural model specifies

- Synchronization between processors
- Synchronization operations
- Atomic operations, shared resources (memory, registers)
- Communication mechanisms: network topology, properties
- ...

Cost model specifies

- Cost of synchronization, 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 von 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: orthogonal 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 architectural 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 structure, memory model, 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 et al.: A single-program-multiple-data computational model for EPEX/FORTRAN, 1988]

OpenMP

MPI

Programming language/library/interface/paradigm



Programming model

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

Algorithms
support



Architecture model



„Real“ Hardware

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

$T_{seq}(n)$: time for 1 processor to solve problem of size n

$T_{par}(p,n)$: time for p processors to solve problem of size n

$$\text{Speedup}(p,n) = T_{seq}(n)/T_{par}(p,n)$$

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:

$T_{seq}(n)$: time for 1 processor to solve problem of size n

$T_{par}(p,n)$: time for p processors to solve problem of size n

Sometimes
S, SU, ...

$$\text{Speedup}(p) = T_{seq}(n) / T_{par}(p,n)$$

If n is fixed, or
„disappears“

Speedup measures the **gain** in moving from sequential to parallel computation

$T_{seq}(n)$, $T_{par}(p,n)$ 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 $25n/p + 3 \ln(4(p/n)) \dots ?$

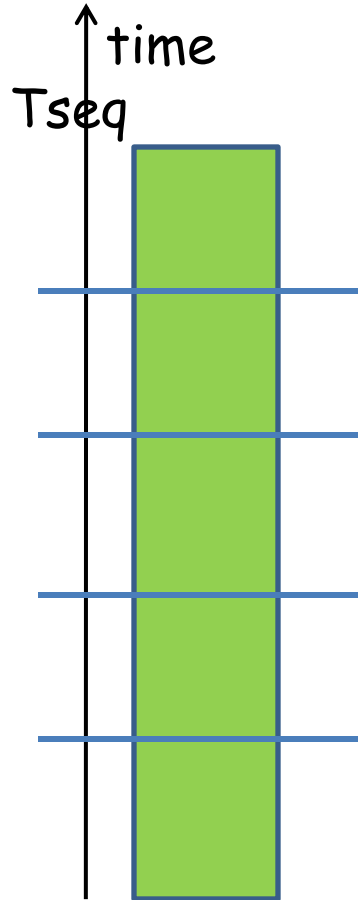
Typically: fix some (good) some algorithm, assume constants in $T_{seq}(n)$ and $T_{par}(p,n)$ comparable, emphasis on orders of magnitude

Ideally: $T_{seq}(n)$ 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

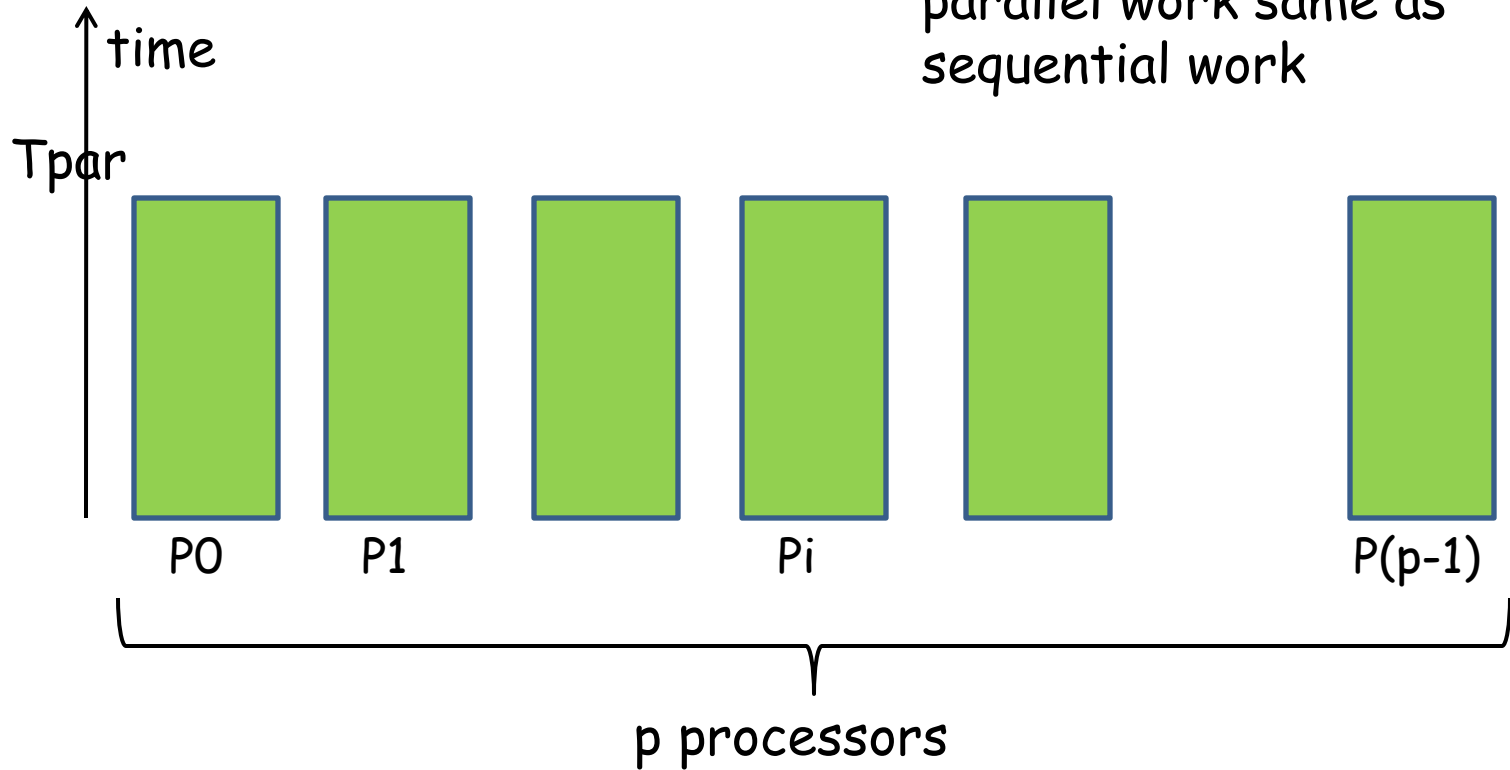


Parallelize: divide work into p independent pieces, assign to p processors...

Sequential time is (sequential) **work**

General: **work** is total number of instructions executed

Here:
parallel work same as
sequential work

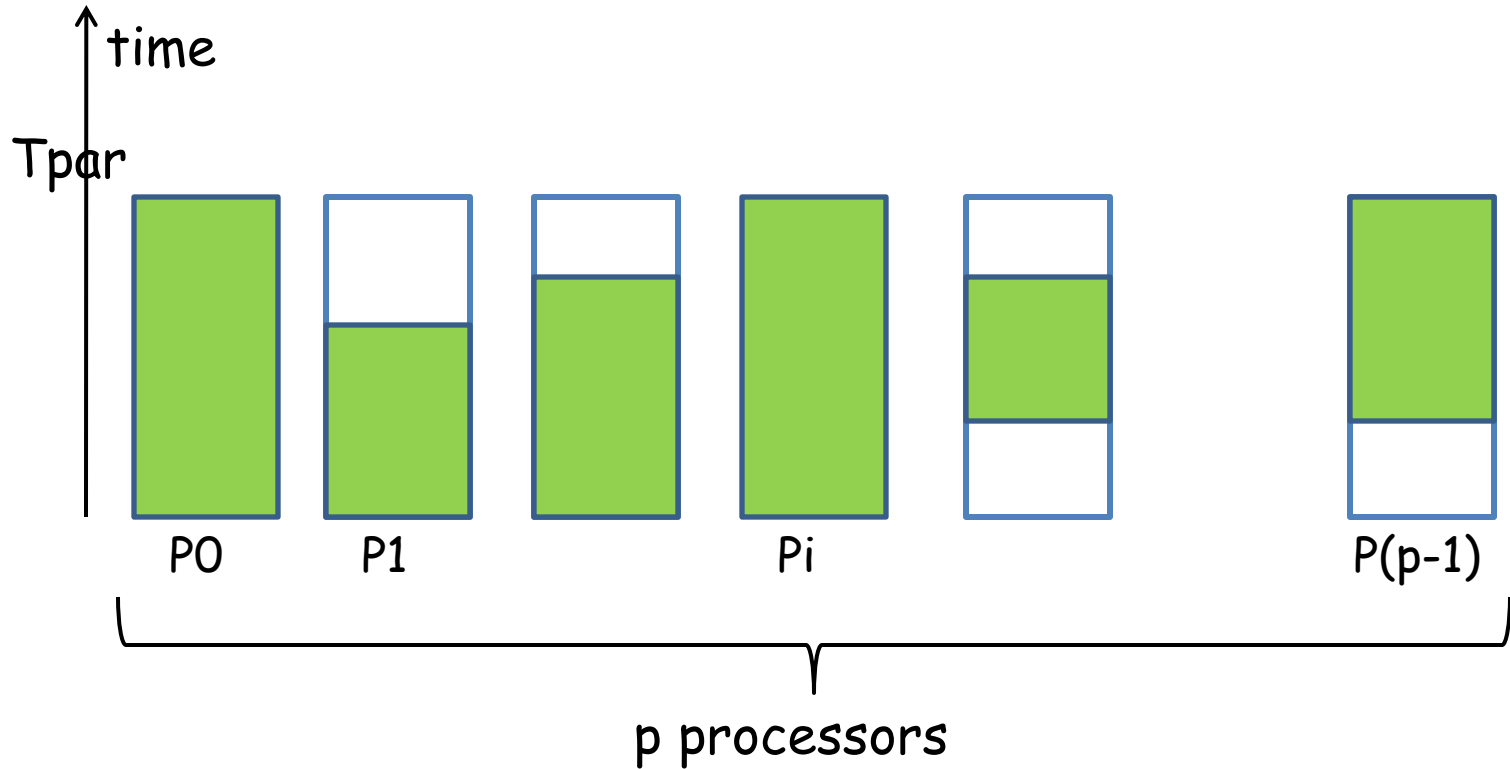


$$T_{par}(p,n) = T_{seq}(n)/p$$

$$\text{Speedup}(p,n) = T_{seq}(n)/T_{par}(p,n) = p$$

Idealized, best case

“embarrassingly parallel”
“pleasingly parallel”
“perfect speedup”



p processors assumed to start at the same time, T_{par} is the time for the slowest/last processor to finish

“Theorem:”

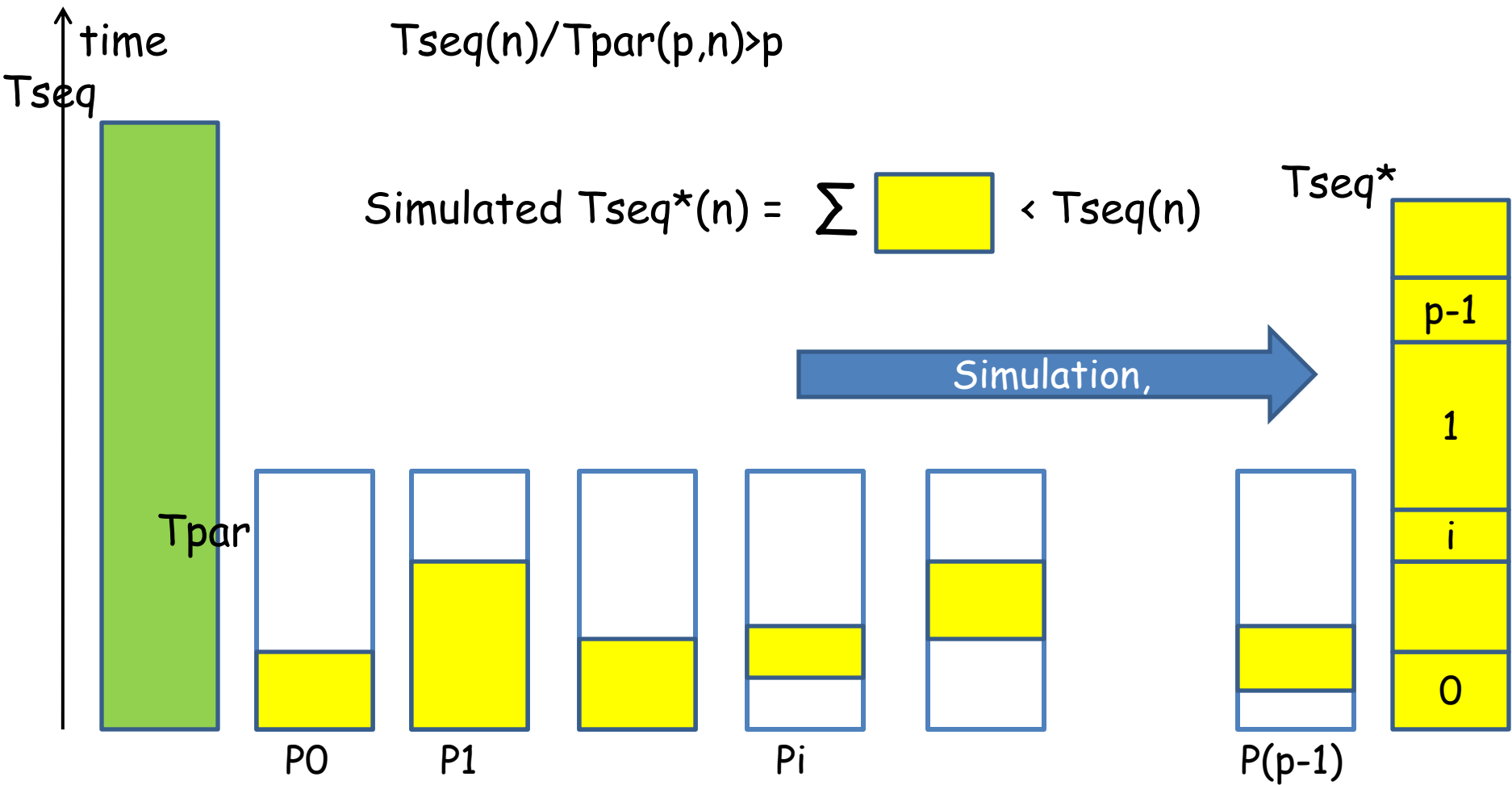
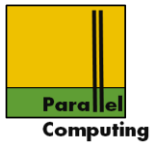
Perfect Speedup(p,n) = p is best possible and cannot be exceeded

“Proof”:

$T_{seq}(n)/T_{par}(p,n) > p$ implies $T_{seq}(n) > p * T_{par}(p,n)$, 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 $T_{seq}(n)$ was best possible

Construction shows that the **total parallel work** must be at least as large as **sequential work** T_{seq} , 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]

Example, Dumb sort, $T_{\text{seq}}(n) = O(n^2)$

that can be perfectly parallelized, $T_{\text{par}}(p,n) = O(n^2/p)$

Well-known $T_{\text{seq}}^*(n) = O(n \log n)$

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

Linear (but low) speedup for **fixed** n

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

$$T_{\text{par}}(p,n) < T_{\text{seq}}(n) \Leftrightarrow n^2/p < n \log n \Leftrightarrow n/p < \log n \Leftrightarrow p > n/\log n$$

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);  
}
```

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

assuming p divides n

**Parallelism
explicit:**

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: $a[i] \leftarrow a[i-1] + a[i] + a[i+1]$

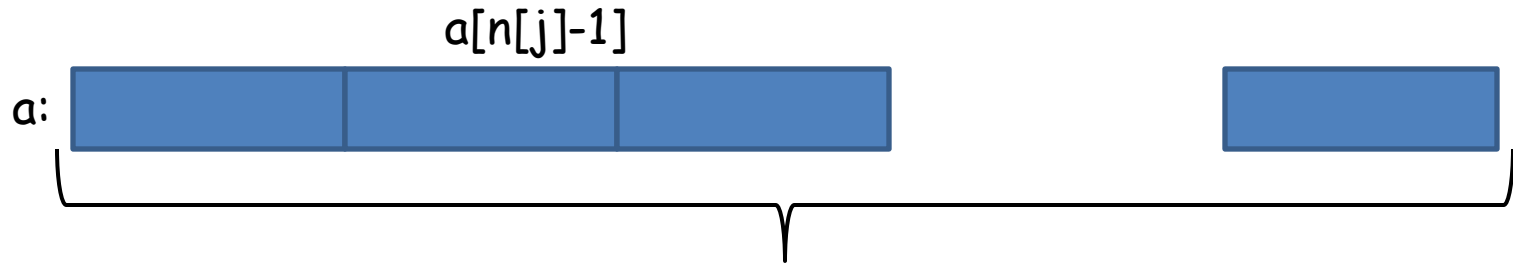
```
for (i=0; i<n; i++) {  
    b[i] = a[i-1]+a[i]+a[i+1];  
}  
for (i=0, i<n; i++) {  
    a[i] = b[i];  
}
```

Processor j , $0 \leq j < p$

```
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 $a[n[j]-1]$?

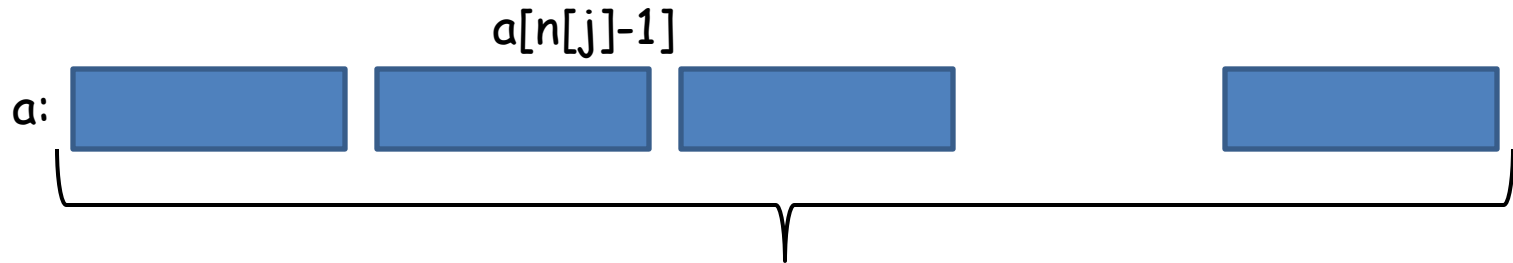
Communication or
synchronization needed



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

Distributed 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]);  
}
```

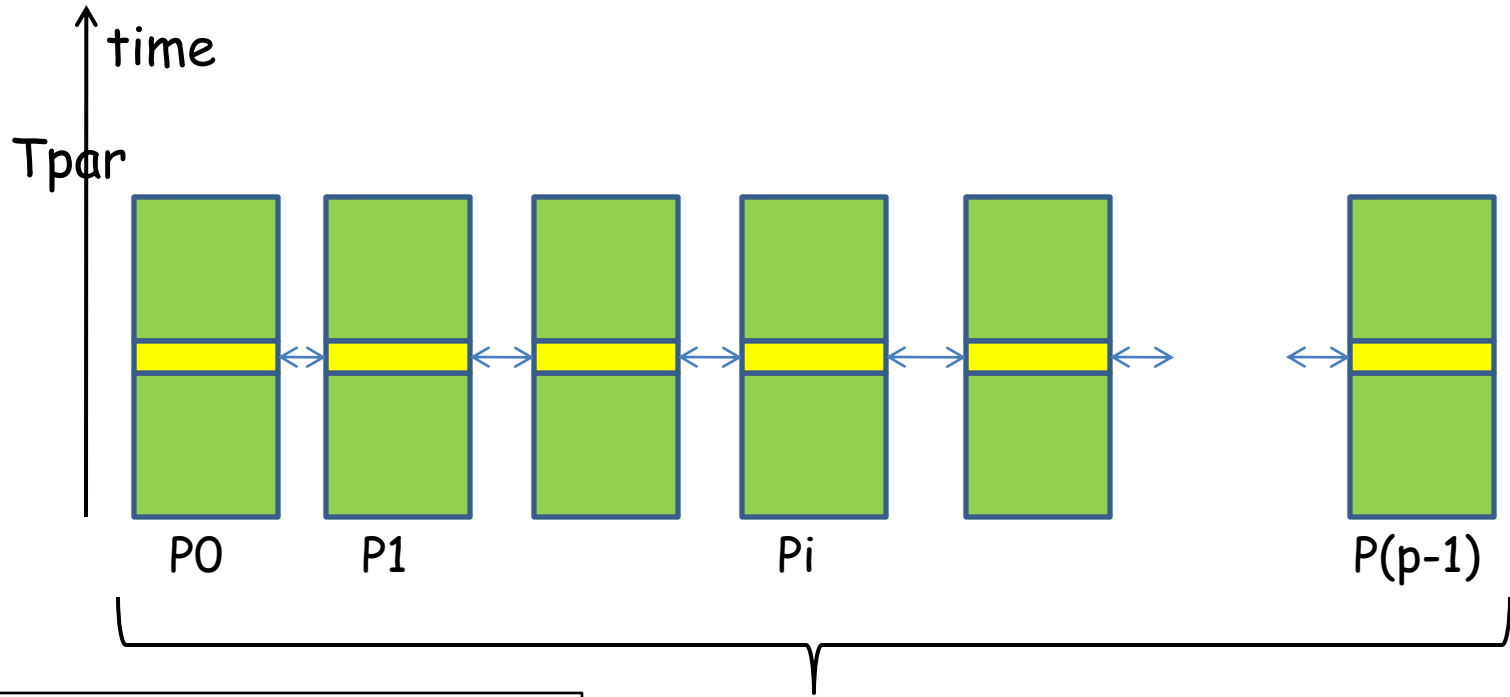
Processor j , $0 \leq j < p$

```
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“



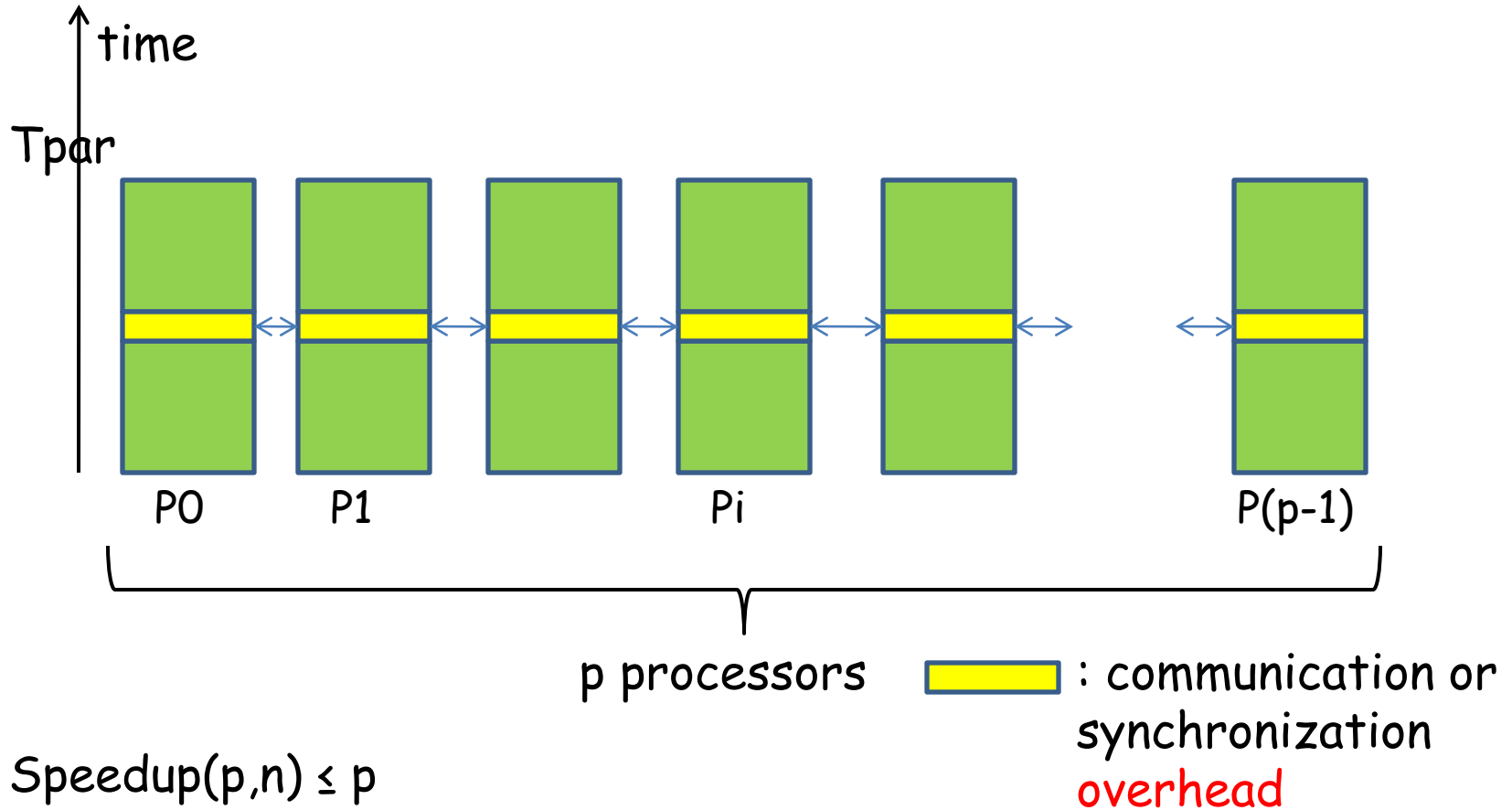
```

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];
}

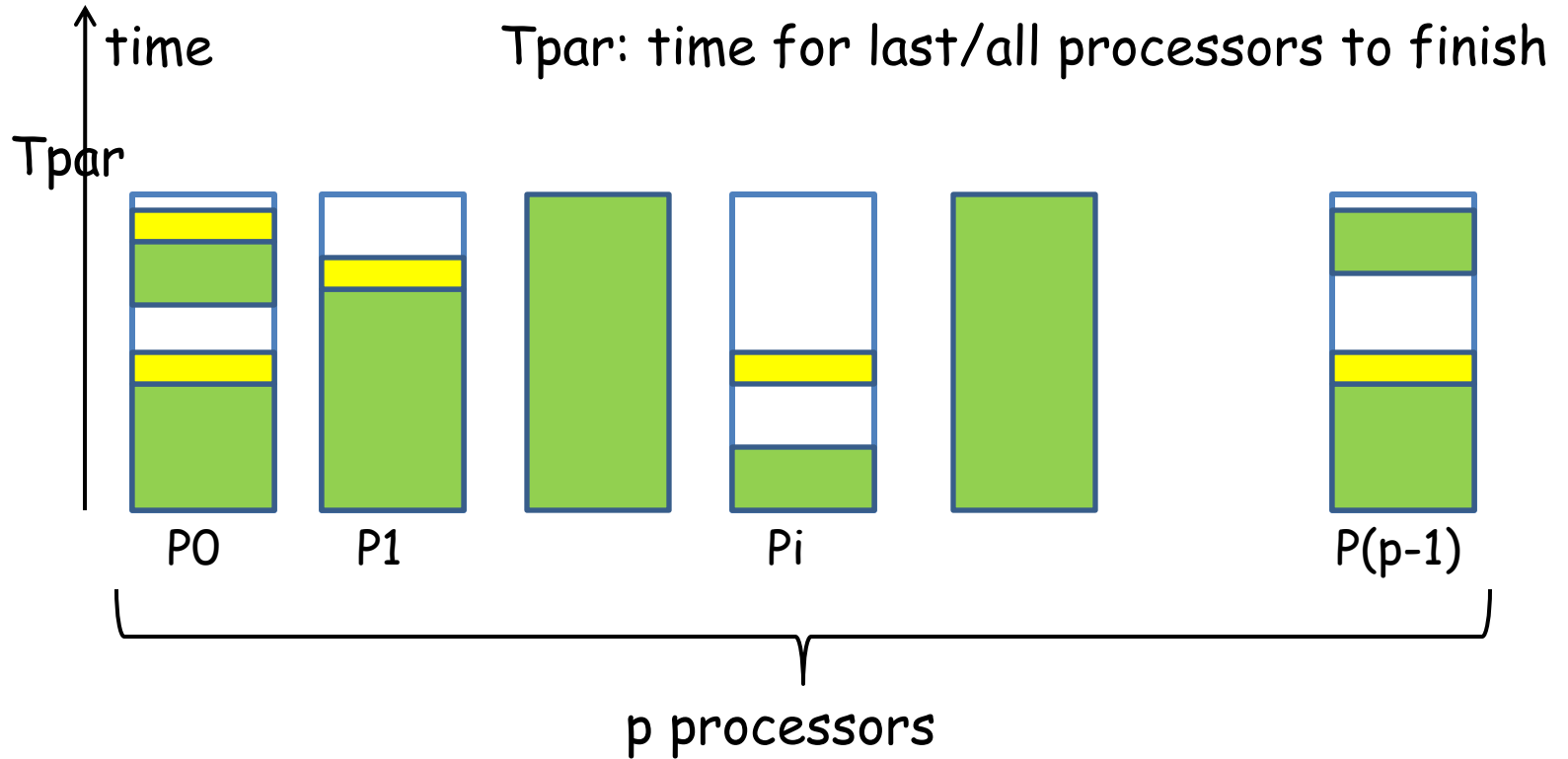
```

p processors  : communication or synchronization overhead

Processor j, $0 \leq j < p$



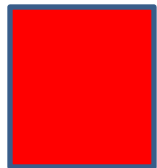
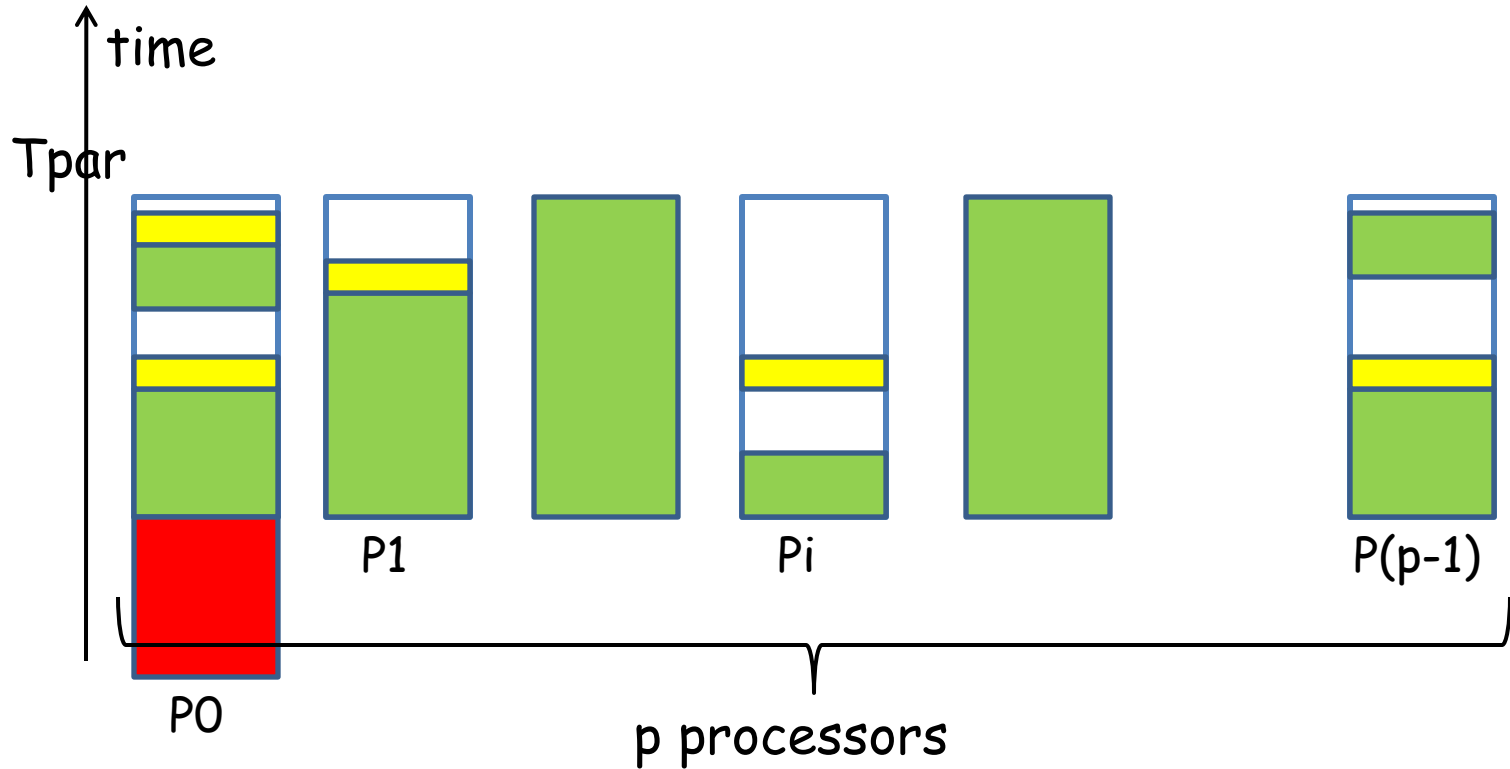
Linear speedup may still be possible, until **overhead** starts to dominate



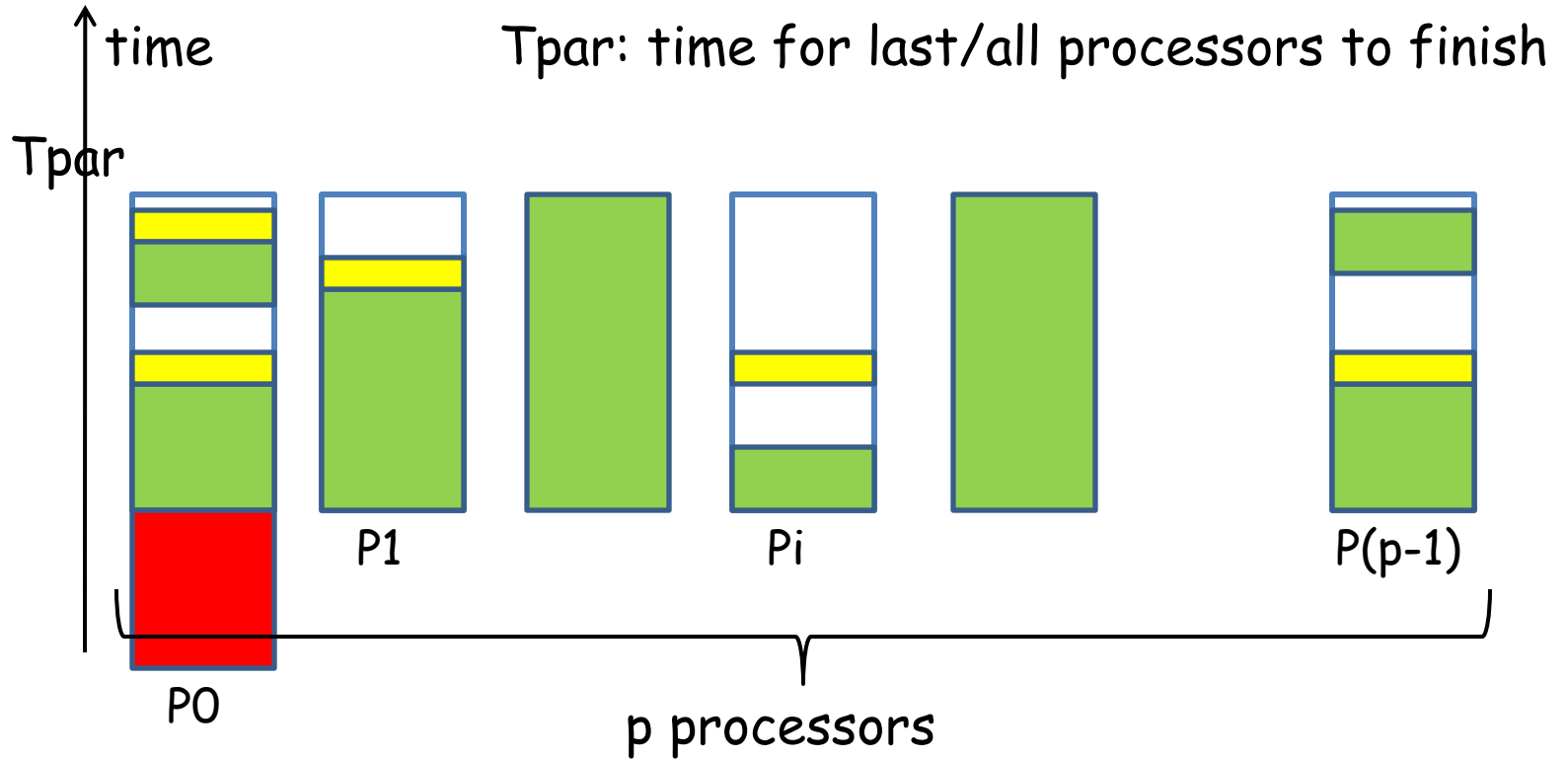
$T_{par}(p,n)$:

useful computational work + parallelization overhead + idle time

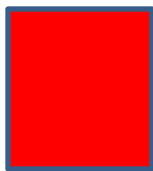




Algorithms/programs typically have a **sequential part** that cannot be parallelized: initialization of data structures, distribution of data, ...



$T_{par}(p,n)$:
 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:

$$T_{seq}(n) = (s+r) \cdot T_{seq}(n)$$

$$T_{par}(p,n) = s \cdot T_{seq}(n) + r \cdot T_{seq}(n)/p$$

$$\text{Speedup}(p,n) = T_{seq}(n) / (s \cdot T_{seq}(n) + r \cdot T_{seq}(n)/p) = 1 / (s + r/p) \rightarrow 1/s \text{ for } p \rightarrow \infty$$

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...

$$T_{\text{seq}}(n) = n + K + Kn$$

$$T_{\text{par}}(p, n) = n + K + Kn/p$$

Sequential fraction $\approx 1/(1+K)$

Speedup(p,n) $\rightarrow 1+K$

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...

$$T_{seq}(n) = 1+K+Kn$$

$$T_{par}(p,n) = 1+K+Kn/p$$

Sequential fraction $\approx 1/(1+n)$

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...

$$T_{seq}(n) = 1+K+Kn$$

$$T_{par}(p,n) = 1+K+Kn/p$$

Sequential fraction $\approx 1/(1+n)$

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

Definition: parallel efficiency

$$E(p,n) = \text{Speedup}(p,n)/p = T_{\text{seq}}(n)/(p \cdot T_{\text{par}}(p,n))$$

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
 $\text{Speedup}(p,n) = \Theta(p)$ (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)
```

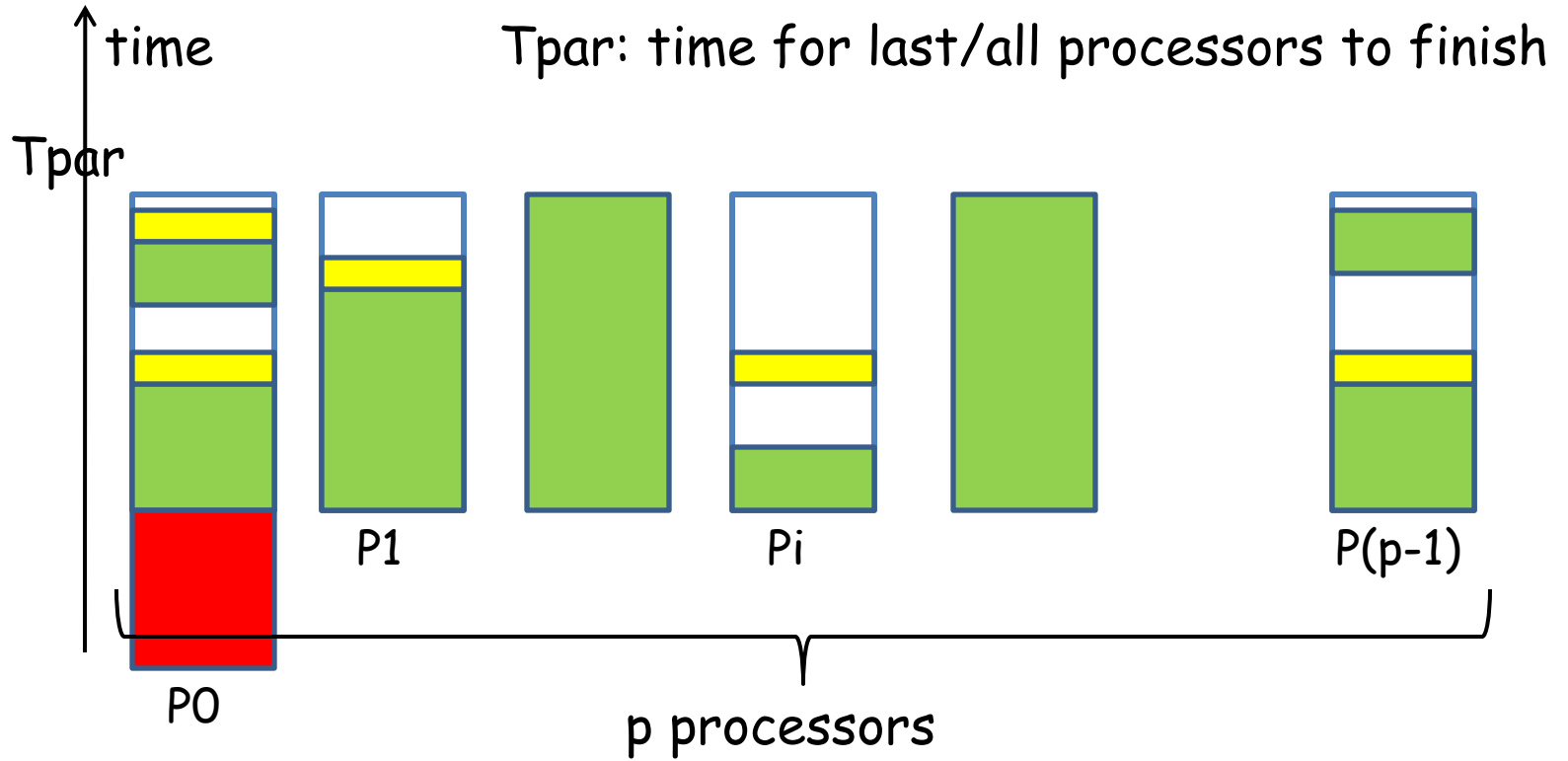
Assume convergence check
takes $O(\log p)$ time

$$T_{\text{par}}(p,n) = Kn/p + K \log p$$

$$E(p,n) \approx Kn / (Kn + pK \log p)$$

For $n \geq p \log p$, $E(p,n) \geq 1/2$

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

$$W_{par}(p,n) = \text{[Red Block]} + \sum \text{[Green Block]} + \text{[Yellow Block]}$$

Definition:

An algorithm/implementation is **work-optimal** if

$$W_{\text{par}}(p,n) = O(T_{\text{seq}}(n))$$

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

Define

$$T_{\text{fast}}(n) = T_{\text{par}}(\infty, n) = \min T_{\text{par}}(p, n), p=1,2,\dots$$

Fastest time that can be achieved assuming enough processors

If $W_{\text{par}}(p,n)$ can be distributed evenly over the p processors, then

$$T_{\text{par}}(p,n) = \max(W_{\text{par}}(p,n)/p, T_{\text{fast}}(n))$$

and

$$\text{Speedup}(p,n) = T_{\text{seq}}(n)/W_{\text{par}}(p,n)/p = p/c$$

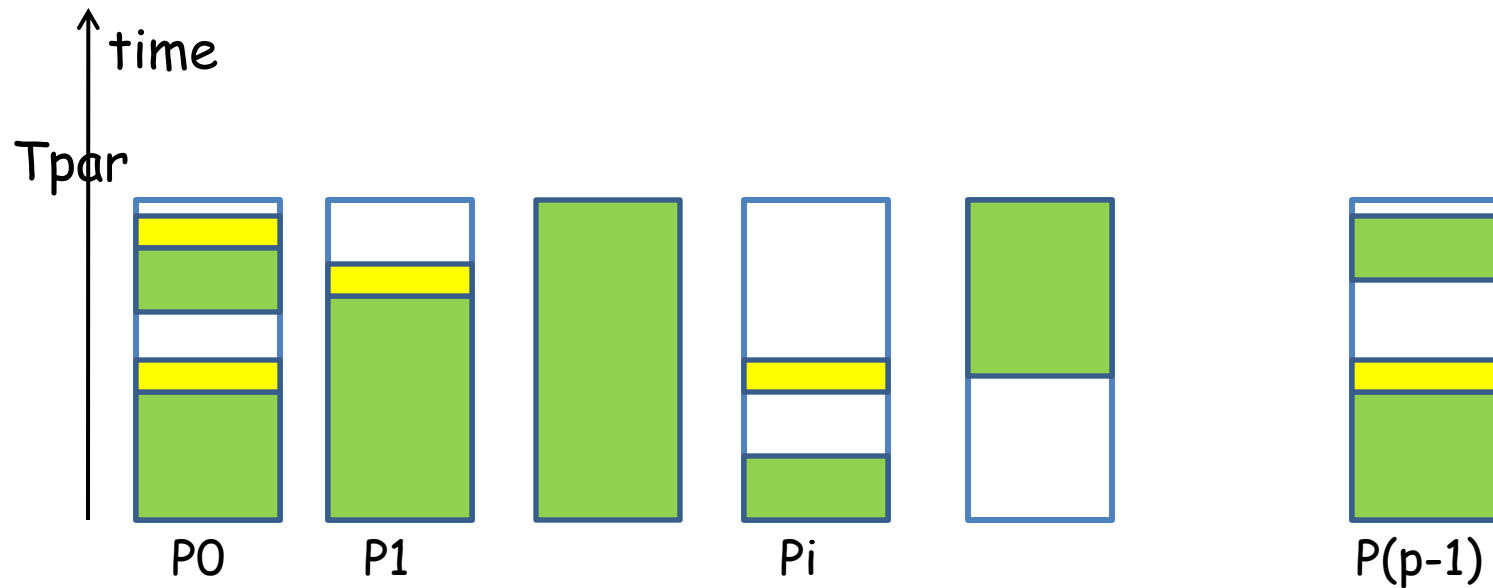
as long as $W_{\text{par}}(p,n)/p \geq T_{\text{fast}}(n)$, for some constant c

Theorem:

Work-optimal implementations/algorithms can have linear speedup for $p \leq W_{\text{par}}(p,n)/T_{\text{fast}}(n)$

- provided the work can be distributed evenly

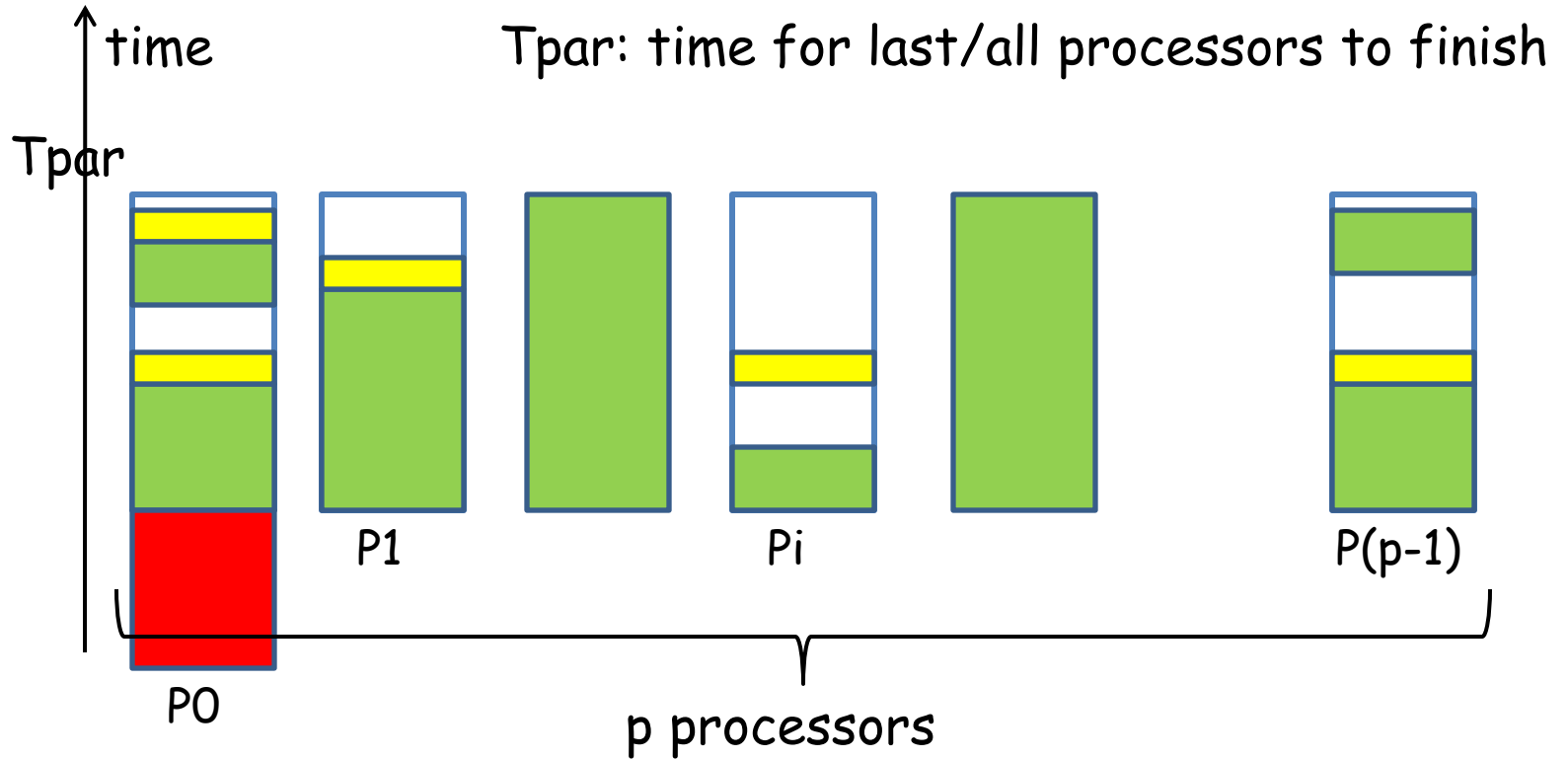
Dividing the work $W_{\text{par}}(p,n)$ into even sized chunks is called **load balancing**. Often **not** trivial. Can sometimes be done **statically**, sometimes **dynamically**, then often called **scheduling**. Assigning the work to processors is called **mapping**. Also **not** trivial.



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

- Determine total work of parallel algorithm, $W(n)$
- Determine fastest time possible = longest chain of dependent operations = $T_{\text{fast}}(n)$ = „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: $p * T_{par}(p, n)$

Dedicated parallel resources: p processors reserved for $T_{par}(p, n)$ time

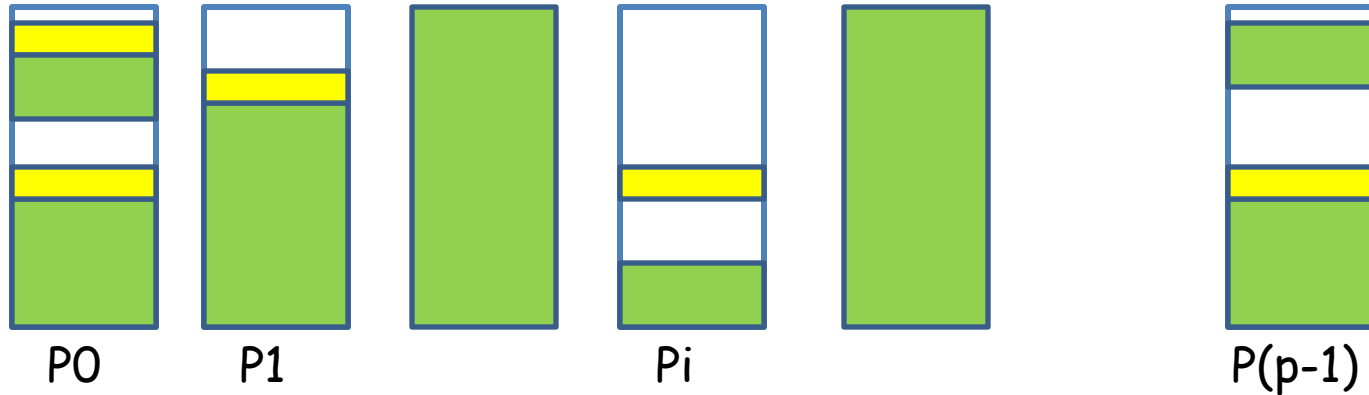
Definition:

An algorithm/implementation is **cost-optimal** if

$$p \cdot T_{\text{par}}(p, n) = O(T_{\text{seq}}(n))$$

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

T_{par}



Overhead is cost minus sequential work

$$\text{Overhead} = p \cdot T_{par}(p, n) - T_{seq}(n)$$

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

Theorem:

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

$$E(p,n) = T_{\text{seq}}(n)/p * T_{\text{par}}(p,n) = T_{\text{seq}}(n)/c * T_{\text{seq}}(n) = 1/c$$

for some constant c hidden in $O(T_{\text{seq}}(n))$

Parallelization: a first example

Problem:

given two ordered sequences (x_i) , $i=0, \dots, n-1$, and (y_i) , $i=0, \dots, m-1$ stored in arrays A and B , merge the two sequences into a single, ordered sequence (z_i) , $i=0, \dots, m+n-1$, 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 $z_k = y_i$

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

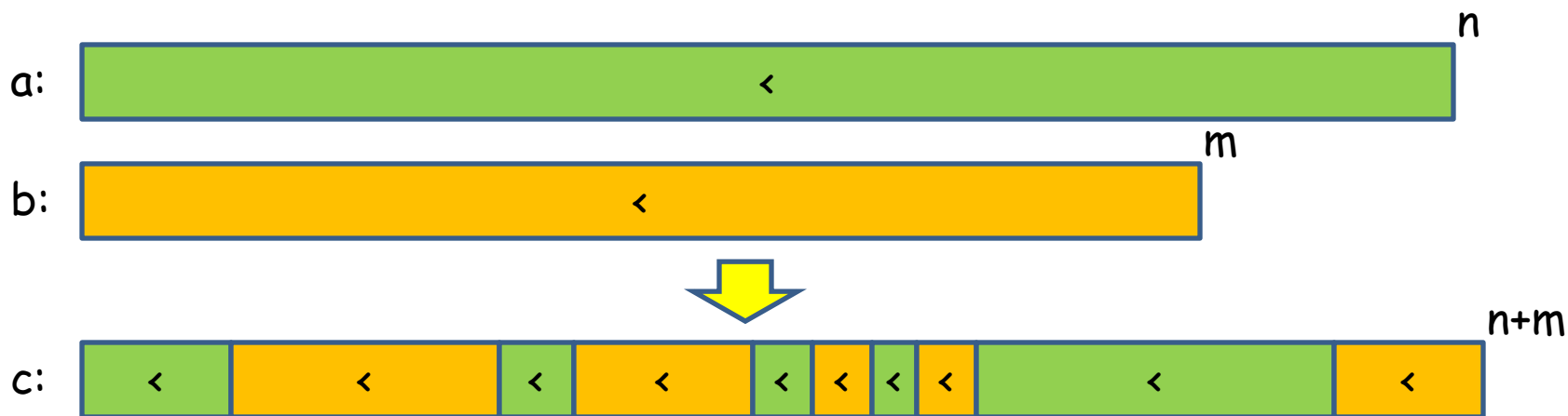
Standard **strictly** sequential solution:

```

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++];

```

$$T_{seq}(n+m) = (n+m)$$



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*

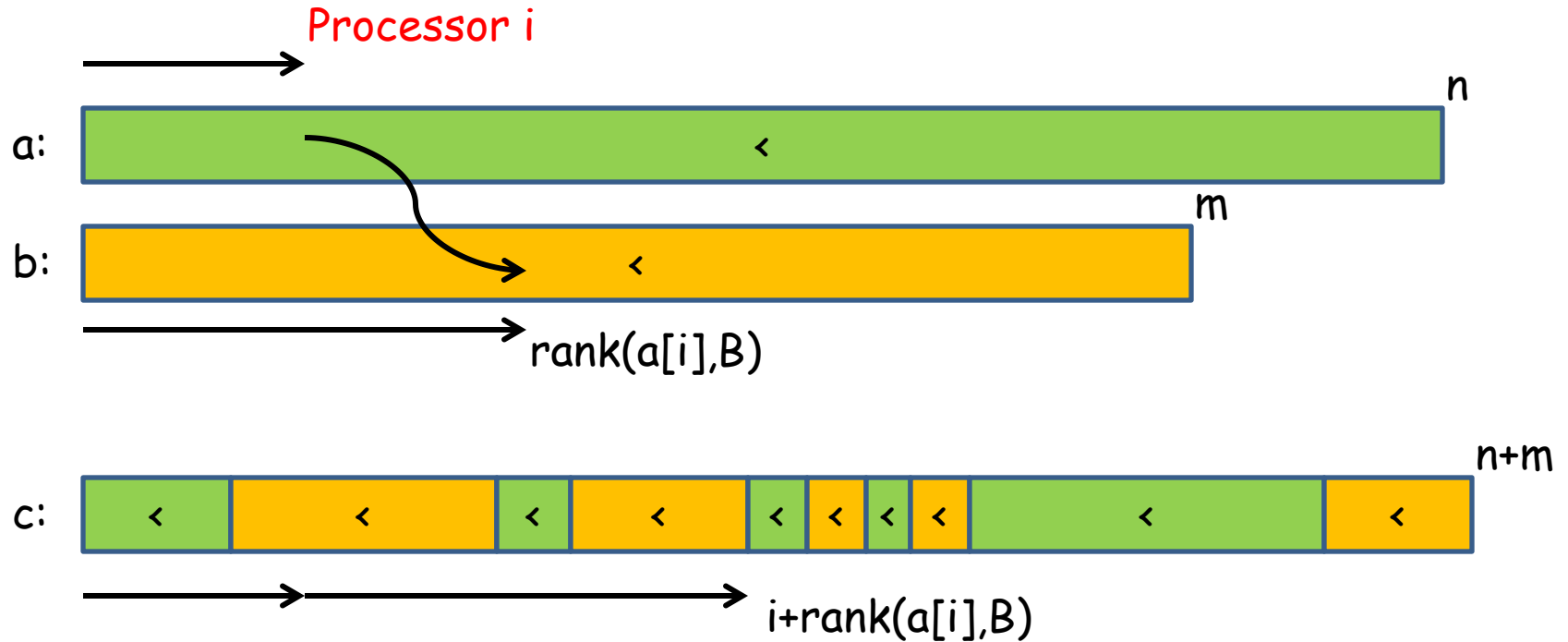
Strategy:

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 , $\text{rank}(x,A)$ 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 ,
 $0 \leq i < n + m$

Observation: for an ordered sequence stored in an array A , $\text{rank}(x, A)$ can be computed by **binary search!**

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

$$T_{\text{par}}(n+m, n+m) = O(\log(\max(m, n)))$$

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

$$\text{Work} = O((m+n)\log(\max(n, m))) \leq O(2n \log n) = O(n \log n)$$

The algorithm is not work efficient, $\text{Speedup}(p) = p/\log p$

Problems:

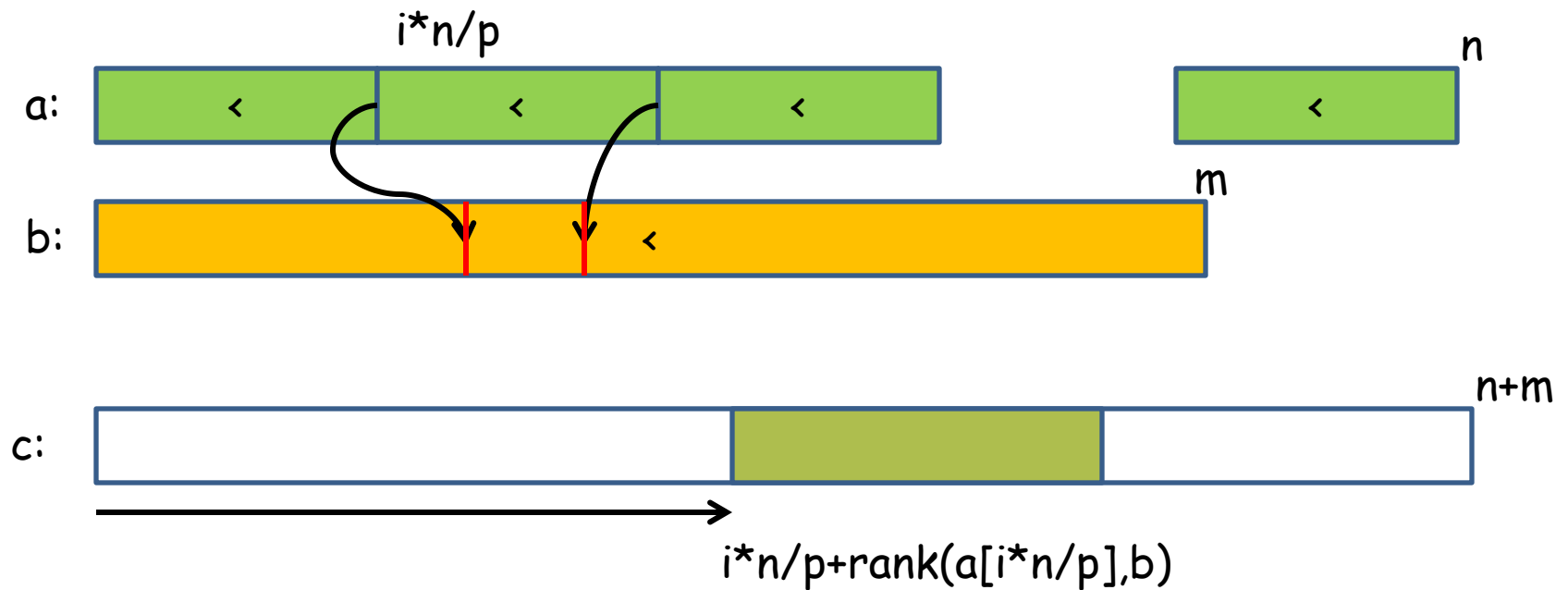
- Algorithm is not efficient
- Normally, $n \gg p$
- When is the computation done (are processes synchronized?)?

```
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 , $0 \leq i < n$

```
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`

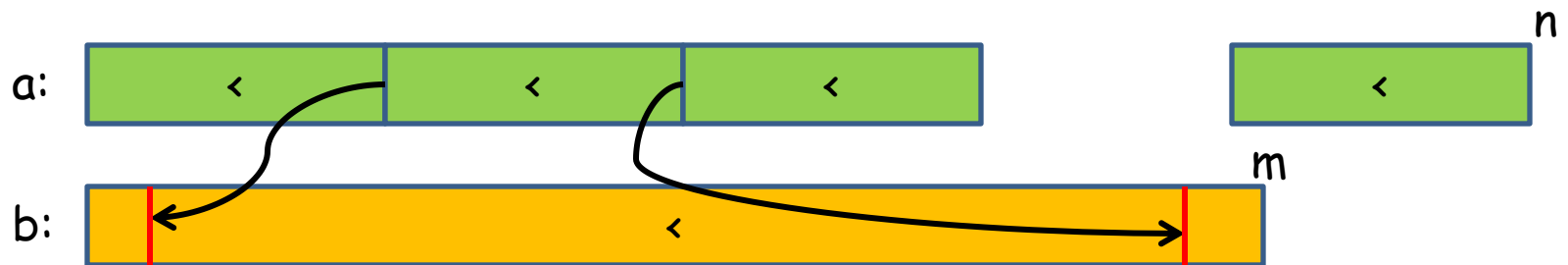
Structure:

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

Work optimal: $Work = p \log m + p \cdot (n/p) + m = p \log m + (n+m) = O(n+m)$

Problems:

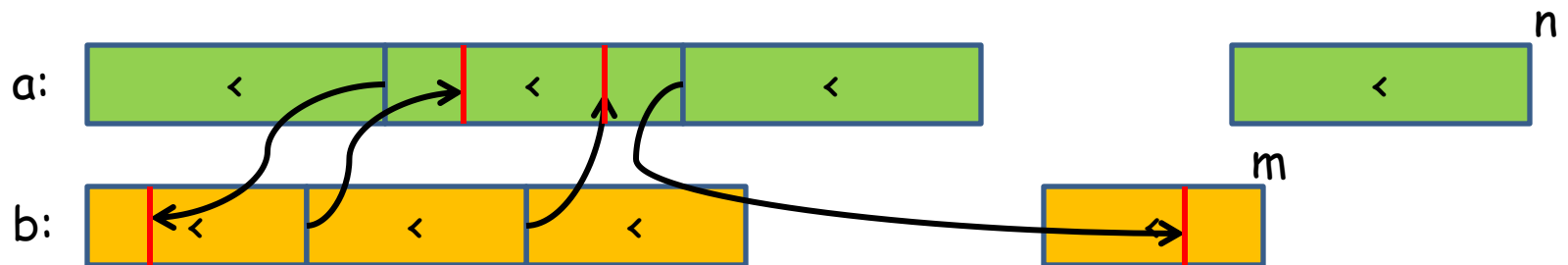
- Assumed that p divides n
- Severe load imbalance in worst case



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

Solution 3:

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

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]

Parallel computing as a practical discipline

Solving given, computational problems in parallel on real parallel machines!

Concerns:

- Solving problems faster
- Solving larger problems
- Solving problems cheaper, **less energy**

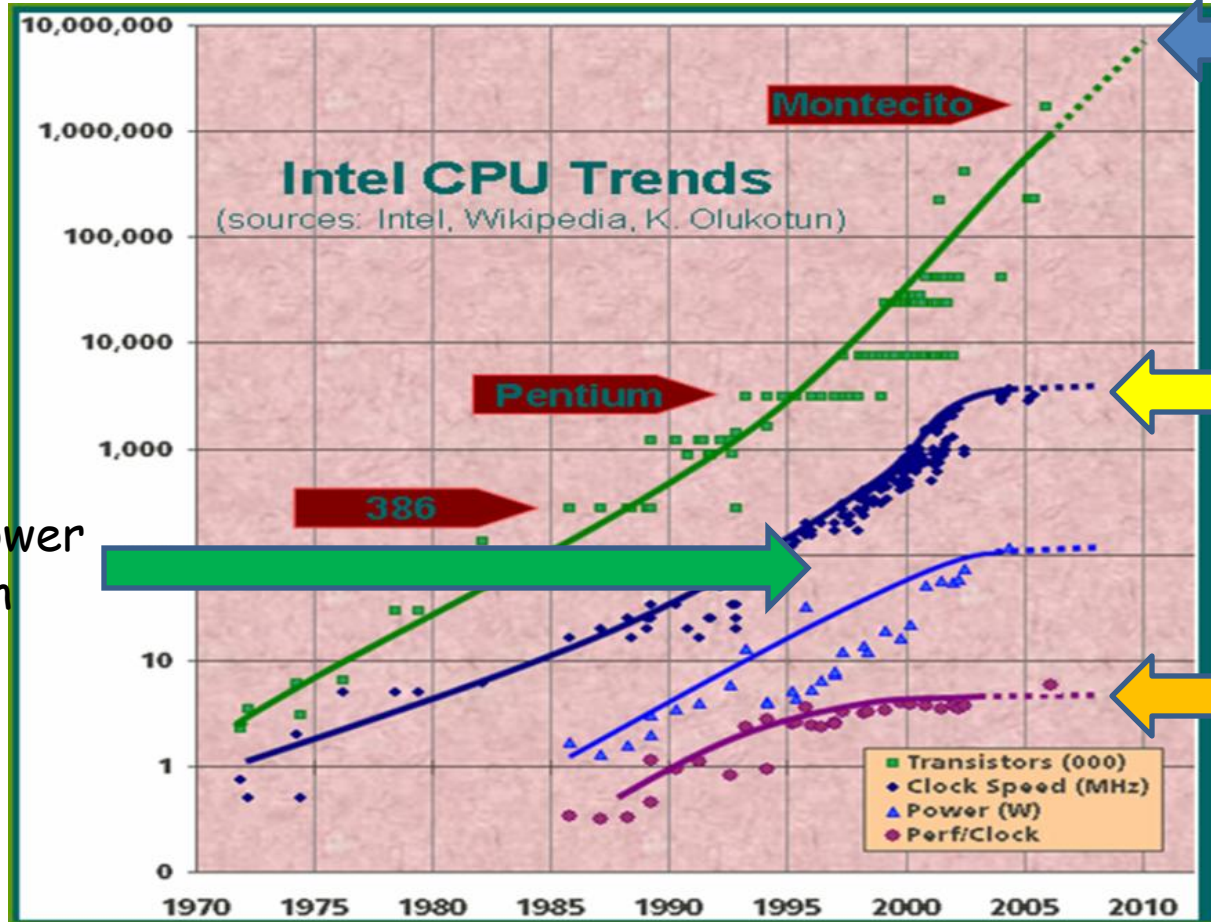
Free lunch is over: sequential processors hardly becoming faster

- How do „real parallel machines“ look?
- Different parallel programming models/paradigms
- Concrete programming languages/interfaces

[Herb Sutter: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, Dr. Dobbs' Journal, 30(3), 2005]

What happened (around 2003)

BUT: number of transistors can still grow



Limits to power consumption

Raw clock speed leveled out

Perf/clock leveled out

Exponential growth in performance often referred to as

Moore's „Law“ (popular version):
Sequential processor performance **doubles every 18 months**

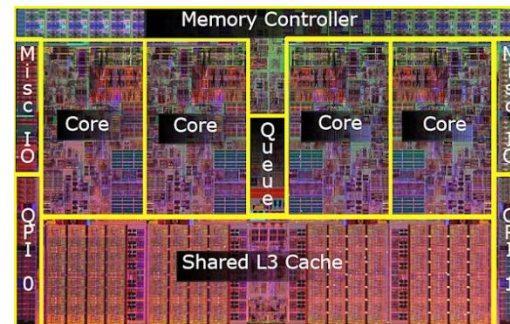
which in the 90ties effectively **killed parallel computing**: to increase performance by an order of magnitude, wait 3 processor generations

The „free lunch“

[Gordon Moore: Cramming more components onto integrated circuits. Electronics, 38(8), 114-117, 1965]

Performance increase due to

1. Increased clock frequency (4MHz ca. 1975 to 4GHz ca. 2005; factor 1000, 3 orders of magnitude)
2. Increased processor complexity: deep pipelining requiring branch prediction and speculative execution; processor ILP extraction (**transistors!**)
3. Multi-level caches (**transistors!**)



Intel i7

Buzz words:

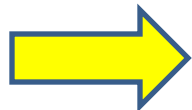
2. **ILP wall**: extracting Instruction Level Parallelism (ILP) grows quadratically or worse with lookahead
3. **Memory wall**: large complex caches to hide memory latency

Buzz word:

3. **Power wall**: limits to how much heat can be cooled from a chip, heat related to power, power related to clock frequency approx. as $P \approx C \cdot V \cdot V \cdot f$, frequency related approx. linearly to voltage, so $P \approx f \cdot f \cdot f$ (perhaps only $P \approx f \cdot f \cdot \sqrt{f}$); f : frequency, V : voltage

Clock frequency increase stopped around 2003, forecasted processors in 10GHz range never materialized

Free lunch is over: single processor cores will not become faster, may even **become slower** - but there will be more of them



Need for efficient parallel processing everywhere!

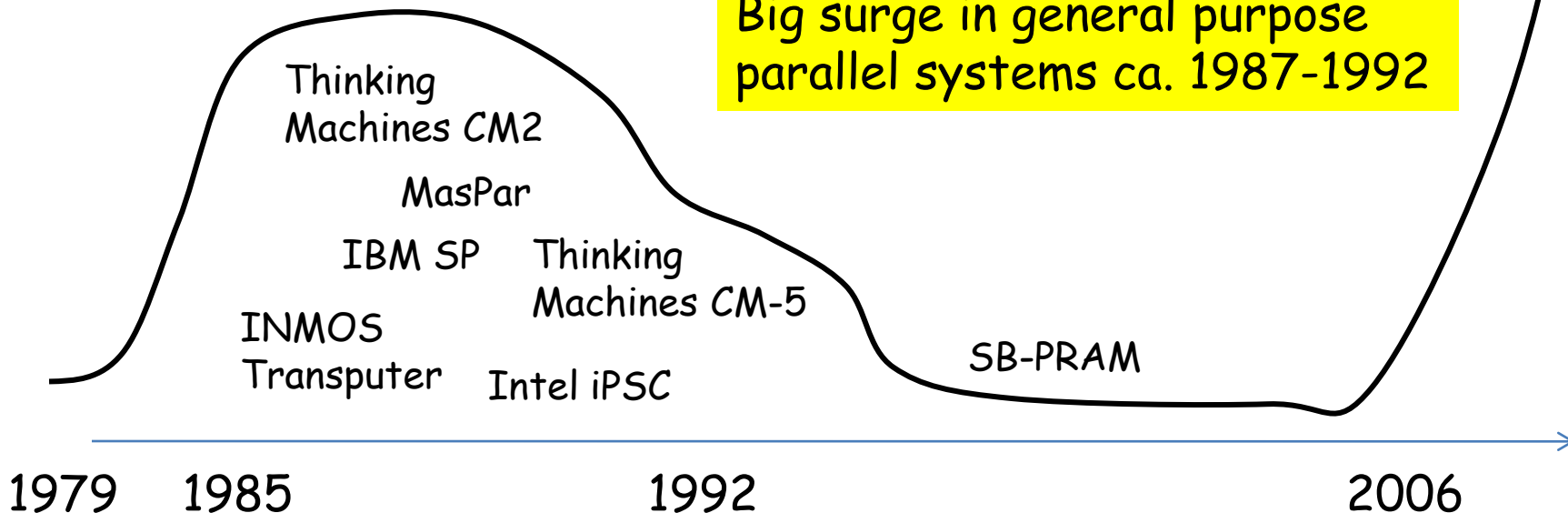
Buzz word:

Green computing: $\frac{1}{2} f \Rightarrow 1/8$ power, same performance by doubling number of cores (≤ 2 times number of transistors), and solving problem in parallel... at $\frac{1}{4}$ power

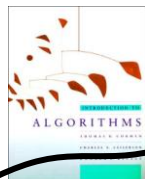
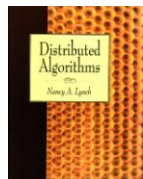
Parallel processing can be more energy efficient; but so can better software...

A final bit of history

Big surge in general purpose parallel systems ca. 1987-1992



PRAM: Parallel RAM

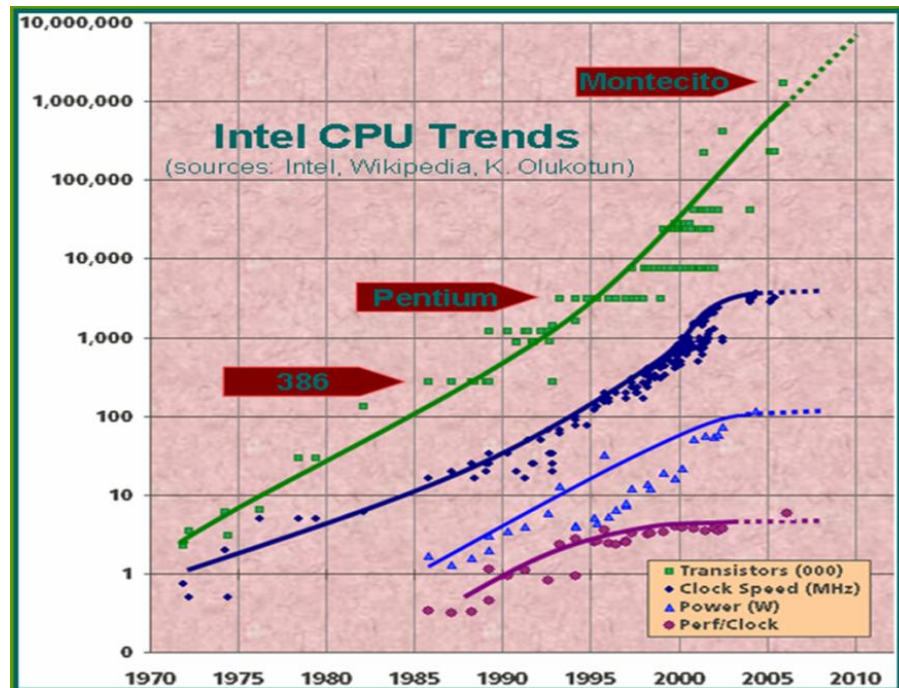


Big surge in theory:
parallel algorithmics
ca. 1980-1993

Surge in parallel computing late 80ties:

- Politically motivated; answer to Japan's 5th generation project (massively parallel computing based on logic programming)
- "Grand challenge" problems (CFD, Quantum, Weather/climate, Symbolic computation)
- **Abundant** (**military**) funding (Reagan, Star Wars, ...)
- Some belief that sequential computing was approaching its limits
- A good model for theory: PRAM
- ...

... but in the early 90ties it became clear that sequential (single-processor) computing was by no means near its limit



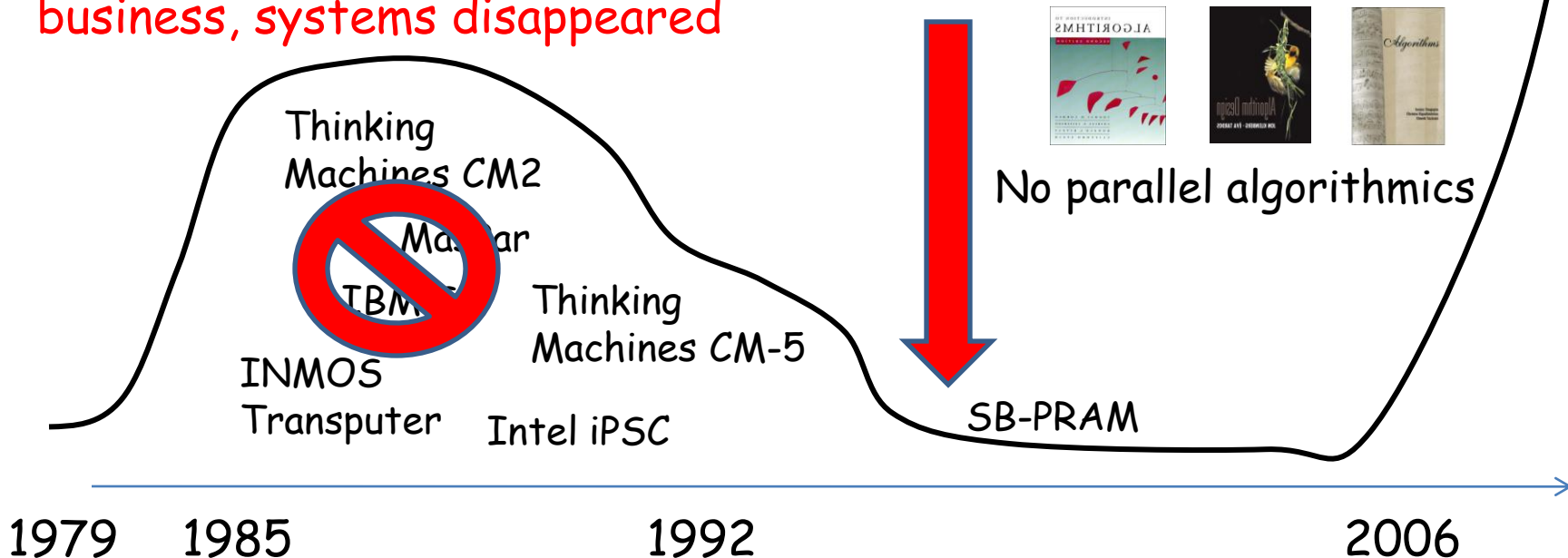
Exponential growth (Moore's law") in

- #transistors
- Clock speed
- ILP

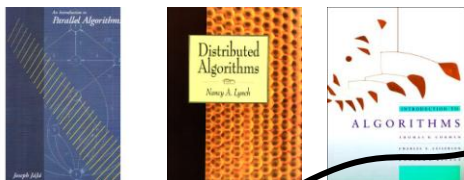
Physicists and computer architects (too) successful
Tricks: shrinking, clock increase, ILP, caches

Companies went out of
business, systems disappeared

Parallel computing/algorithmics
disappearing from mainstream
CS curricula



PRAM: Parallel RAM



Scientific, High Performance Computing (SC, HPC)

Methods and systems for solution of **extremely large, extremely computationally intensive problems**

- Grand challenge problems (still): climate, global warming
- Engineering: CAD
- Physics: cosmology, particle physics, string theory, ...
- Biology, chemistry: protein folding

- Drug design, screening

- Security, military??? Who knows?

- ...

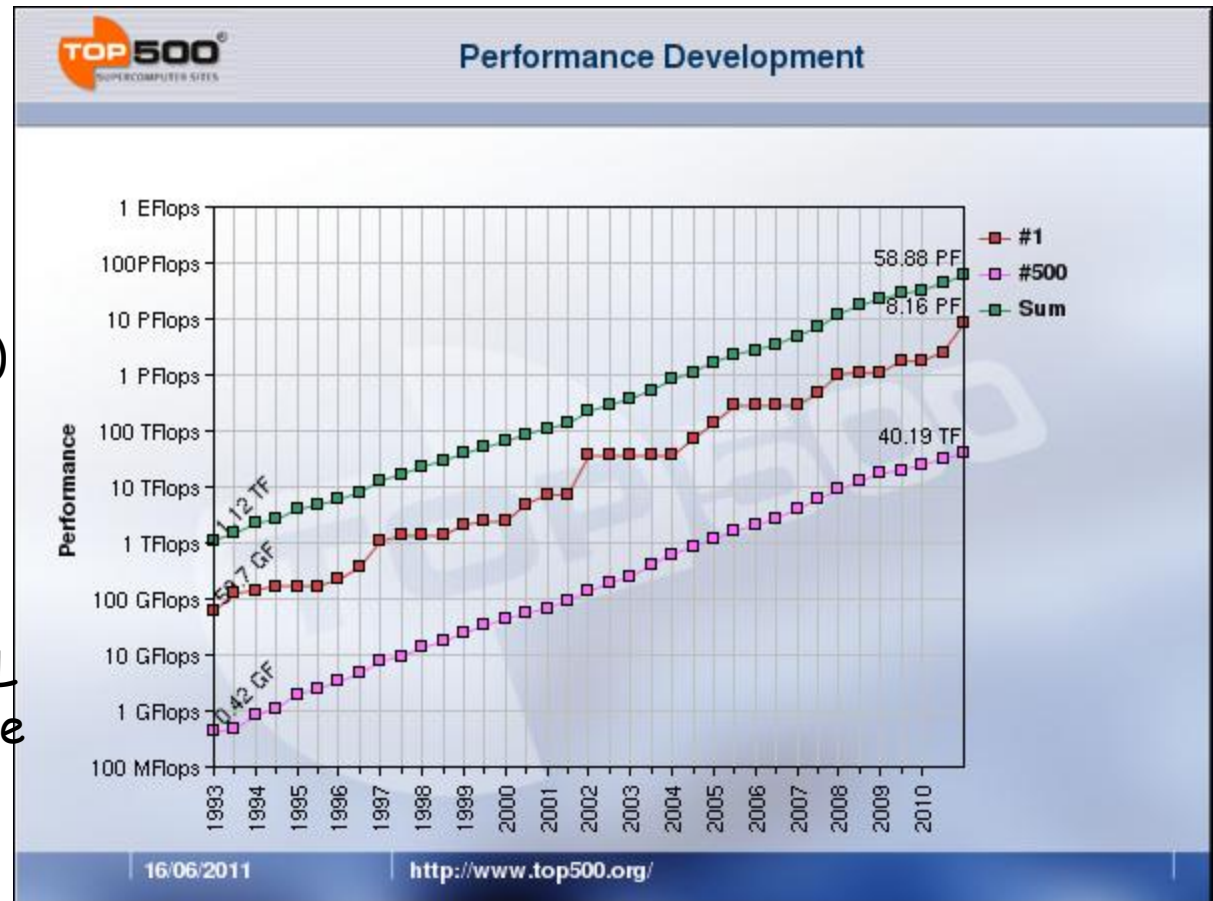
Since early 90ties a niche for parallel computing.

HPC System performance recorded in Top500 list

- 500 most „powerful“ systems in the world

- Measured based on performance (FLOPS) rate of **single benchmark**: Linpack

Reasonable? Does HPL performance translate into application performance?



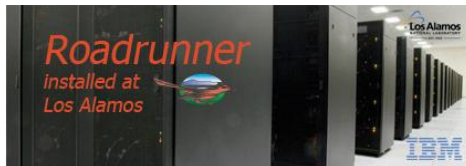
#	Organization	System	Manufac	Country	Cores	Max	Peak
1	RIKEN Advanced Institute for Computational Science	K computer, SPARC64 VIIIfx Tofu interconnect	Fujitsu	Japan	548352	8162000	8773630
2	National Supercomputing Center Tianjin	Intel X5670, NVIDIA GPU,	NUDT	China	186368	2566000	4701000
3	DOE/SC/Oak Ridge National Laboratory	Cray XT5-HE Opteron 6-core	Cray Inc.	USA	224162	1759000	2331000
4	National Supercomputing Centre Shenzhen	Intel X5650, NVidia Tesla GPU	Dawning	China	120640	1271000	2984300
5	GSIC Center, Tokyo Institute of Technology	Xeon X5670, Nvidia GPU	NEC/HP	Japan	73278	1192000	2287630
6	DOE/NNSA/LANL/SNL	Cray XE6	Cray Inc.	USA	142272	1110000	1365810
7	NASA/Ames Research Center/NAS	SGI Altix Xeon Infiniband	SGI	USA	111104	1088000	1315330
8	DOE/SC/LBNL/NERSC	Cray XE6	Cray Inc.	USA	153408	1054000	1288630
9	Commissariat a l'Energie Atomique	Bull	Bull SA	France	138368	1050000	1254550
10	DOE/NNSA/LANL	PowerX Cell 8i Opteron Infiniband	IBM	USA	122400	1042000	1375780

#	Organization	System	Manufac	Country	Cores	Max	Peak
...							
	TU Wien, Uni Wien, 56BOKU	Opteron, Infiniband	Megware	Austria	20700	135600	185010
...							

Max, Peak: GFLOPS



June 2011



NEC Earth
Simulator:
2002-2004

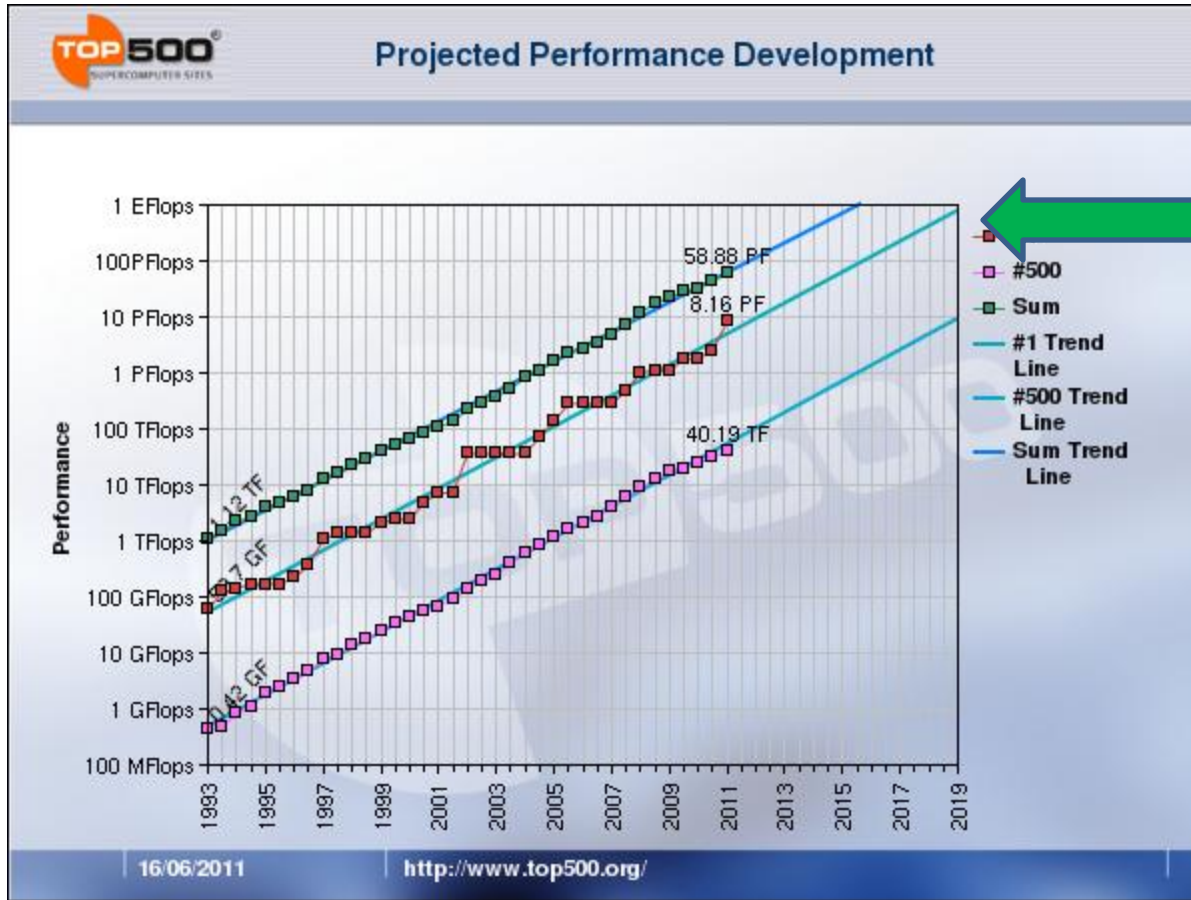


Top500 (source: www.top500.org - also www.green500.org) gives **valuable information** on current trends, developments, and the history of supercomputing systems:

- Since ca. 1995 no single-processor system on list(!)
- HPL (High Performance Linpack) performance over 8 PFLOPS
- Using well over 100.000 processor cores
- Many systems are hybrid/heterogeneous: accelerators (Nvidia GPU, Radeon GPU, Cell, ...)

NOT necessarily on „most powerful system“: can it be programmed? Good for other, real applications? Quality of software? MTF? ...

Supercomputer „performance“ evidence for Moore's law!?

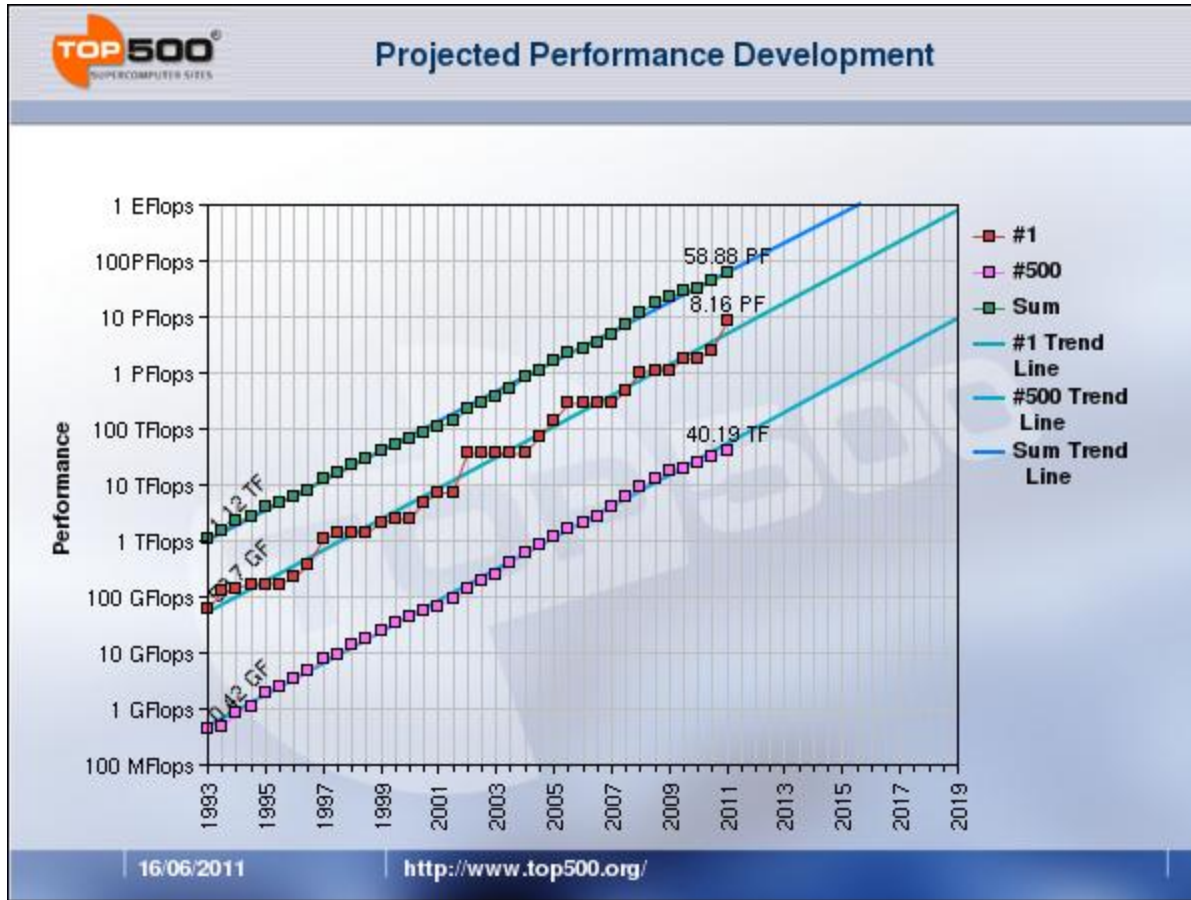


Exascale
(10^{18} FLOPS)

Moore's law:

- Not a law of nature
- Empirical observation
- Prediction
- **Self-fulfilling prophecy/dictate**

Supercomputer „performance“ evidence for Moore's law!?



Peter Hofstee,
IBM Cell co-
designer



„...a self-fulfilling prophecy... nobody can afford to put a processor or machine on the market that does not follow it“, HPPC 2009

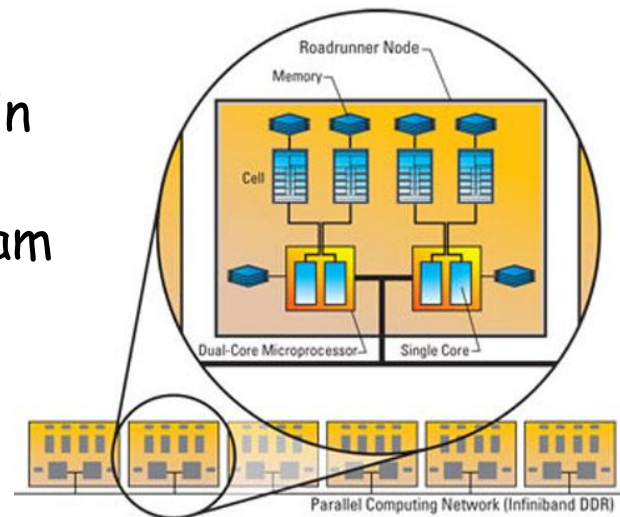
Supercomputer „performance“ evidence for Moore's law!?



Peter Hofstee,
IBM Cell co-
designer



...which may explain
some of the very
difficult to program
Top500 systems



„...a self-fulfilling
prophecy... nobody
can afford to put a
processor or machine
on the market that
does not follow it“,
HPPC 2009

Note: in HPC sometimes alternative, hardware-oriented definition of efficiency is used

System/HPC Efficiency:
Ratio of „theoretical peak-rate“ to FLOPS (FLOating Point operations per Second) achieved by application

Measures:

How well is hardware/architecture capabilities actually used?

Caution:

- Used in Top500 and elsewhere in HPC
- Undefined what „theoretical peak-rate“ is
- Just measuring FLOPS: Inferior algorithm may be more „efficient“ than otherwise preferable algorithm

Parallel computing as a practical discipline

1. Often: how can an **already given algorithm** (program) be parallelized?
2. Only if not possible, or not effective: look at **problem**, develop parallel algorithm, implement

Amdahl's law: the fraction that is not parallelized limits speedup

$$\text{Speedup}(p,n) = T_{\text{seq}}(n) / T_{\text{par}}(p,n)$$

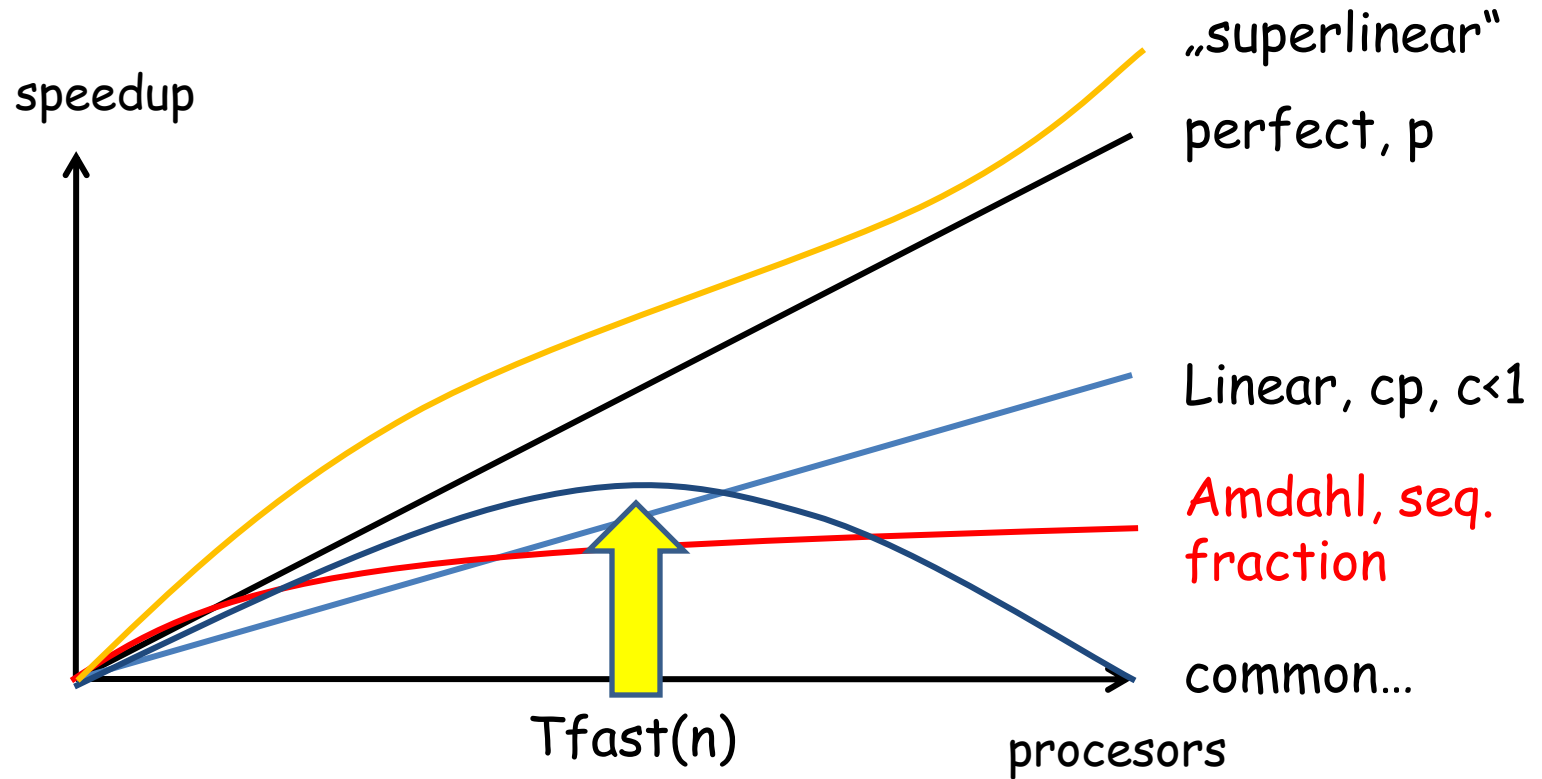
Empirical measure: T_{seq} , T_{par} **actual execution** times of (best) sequential and parallel implementations on given, concrete machines.

Caution:

Speedup is not relative to parallel time with 1 processor - although often/sometimes reported

$$T_{\text{par}}(1,n)/T_{\text{par}}(p,n)$$

Dumb sort $O(n^2/p)$ would mistakenly be judged an excellent parallel algorithm/implementation; might also have high hardware efficiency (FLOPS rate)



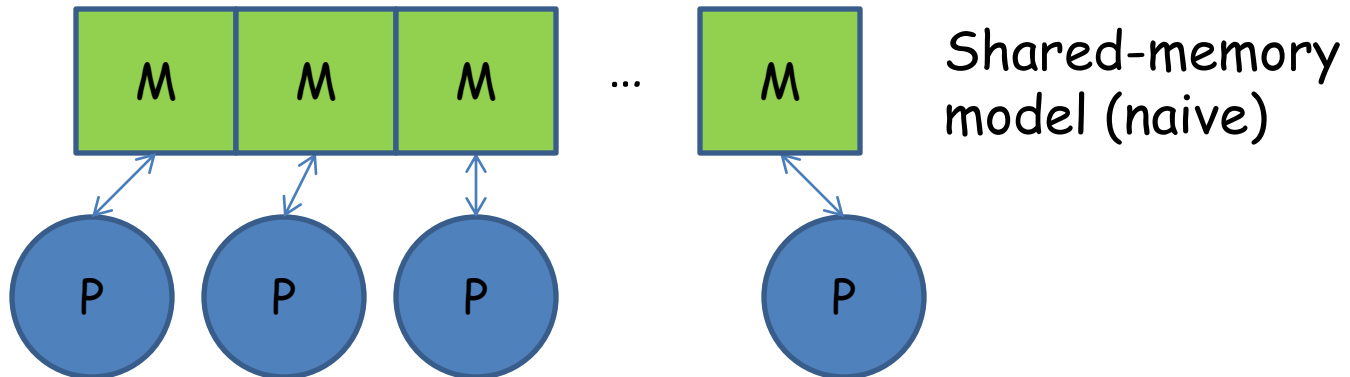
- Typical speedup, fixed n
- Empirical speedup declines after some p^* corresponding to $T_{fast}(n) = T_{par}(p^*)$ - problem too small, overhead dominates
- **Superlinear?**

Sources of superlinear speedup:

Differences between sequential and parallel hardware:
a) the memory hierarchy

Sources of superlinear speedup:

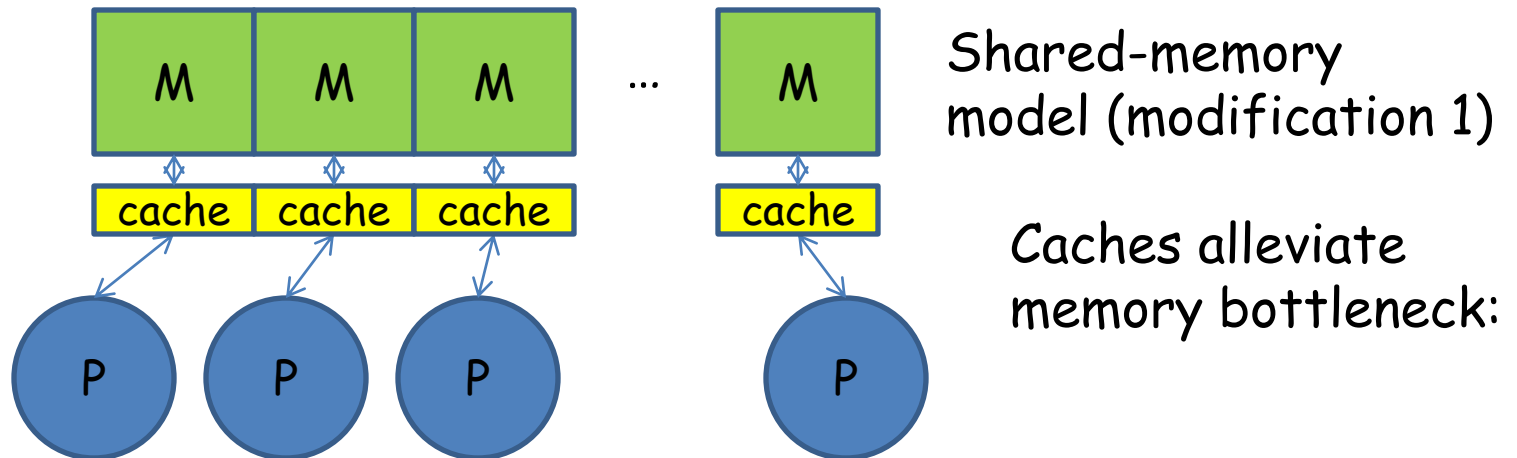
Differences between sequential and parallel hardware:
a) the memory hierarchy



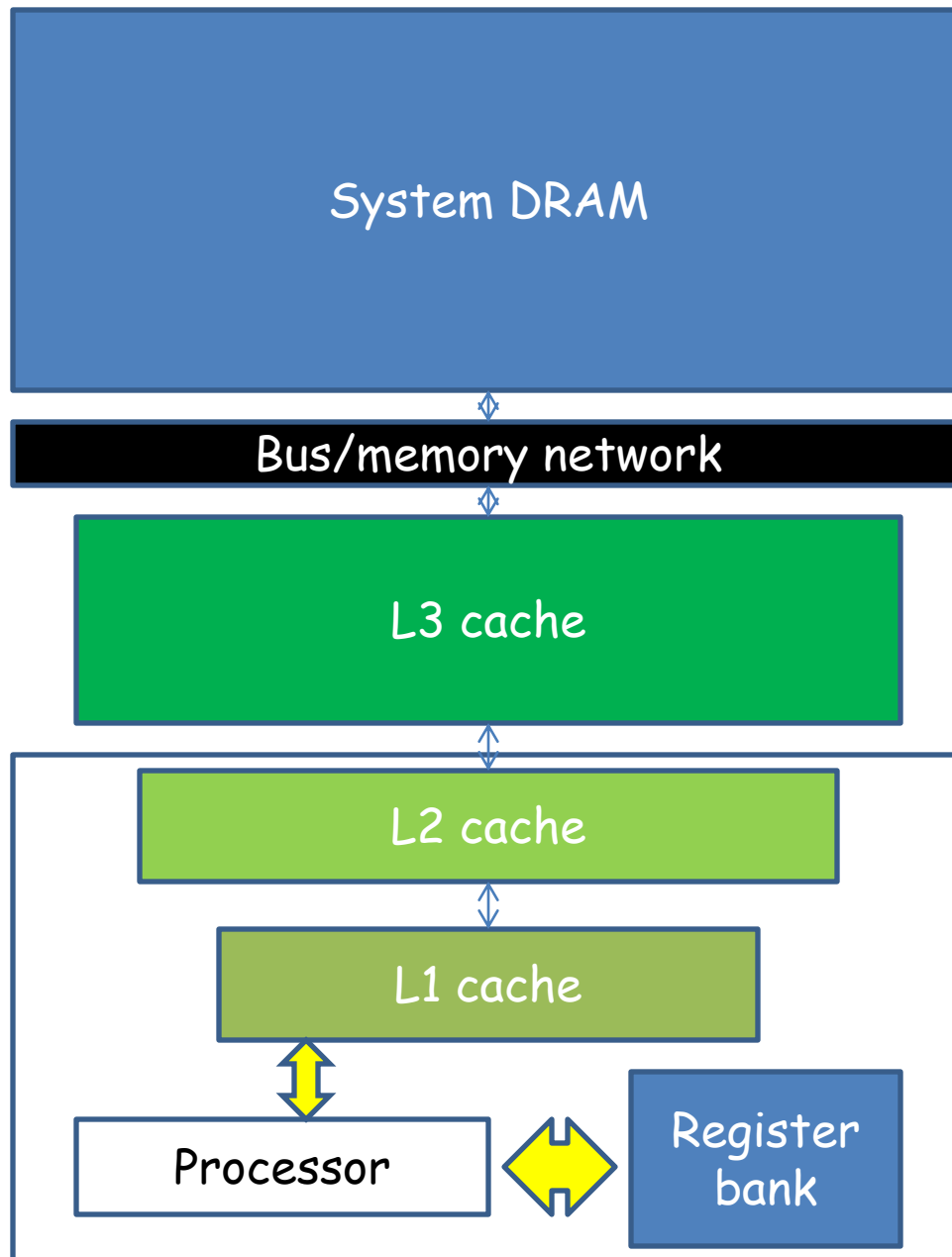
Sources of superlinear speedup:

Differences between sequential and parallel hardware:

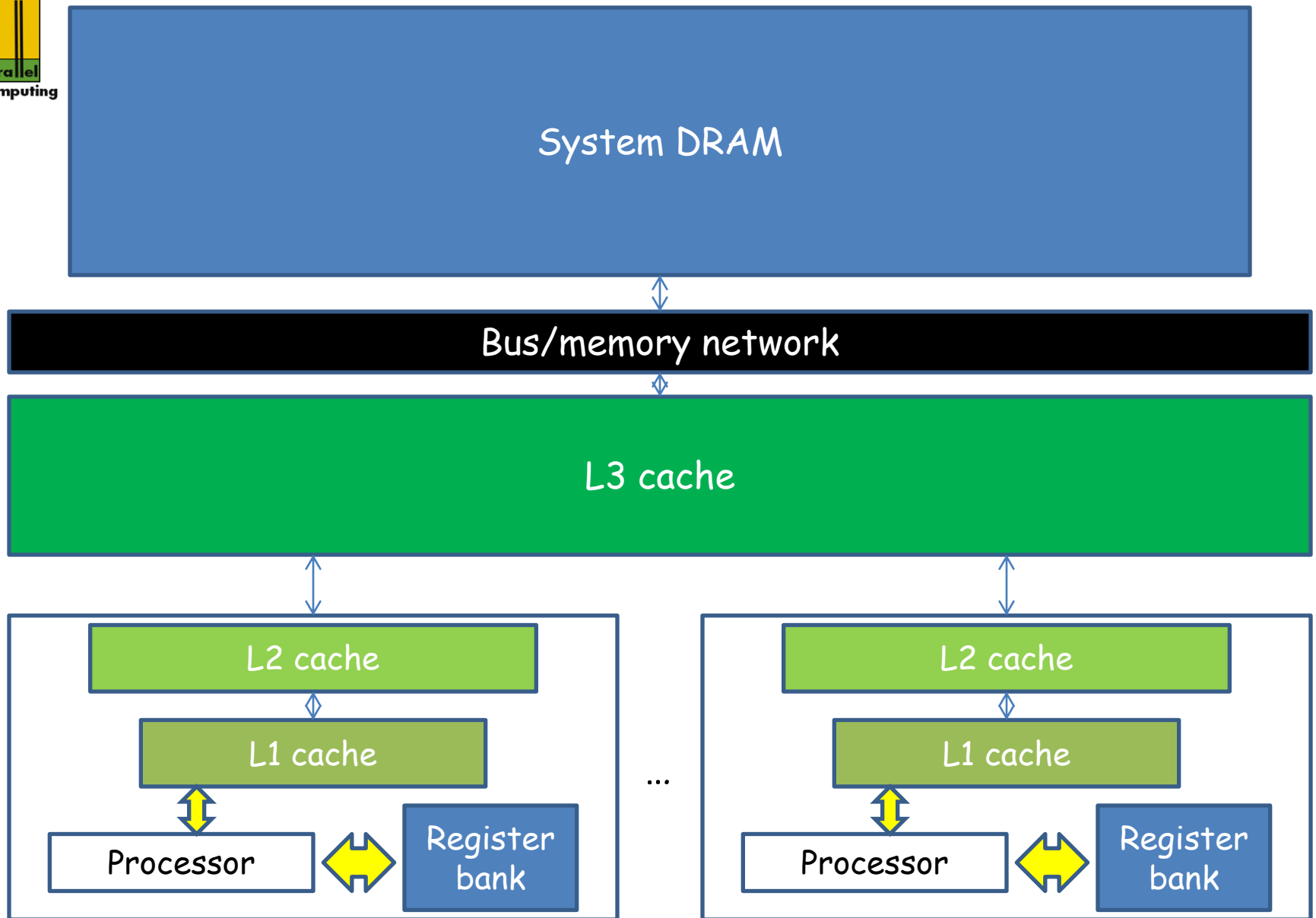
a) the memory hierarchy

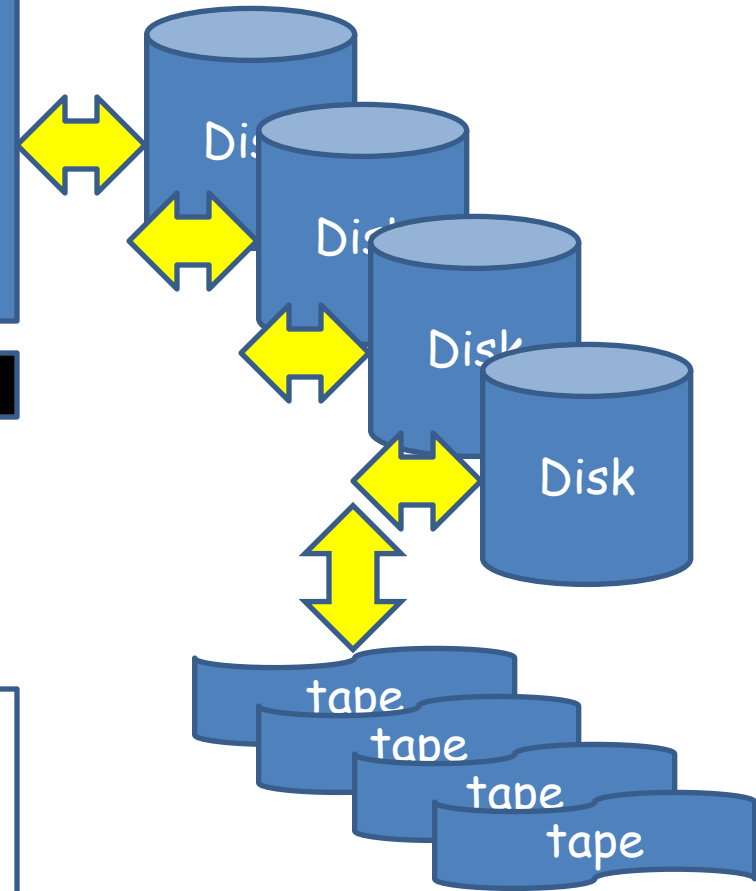
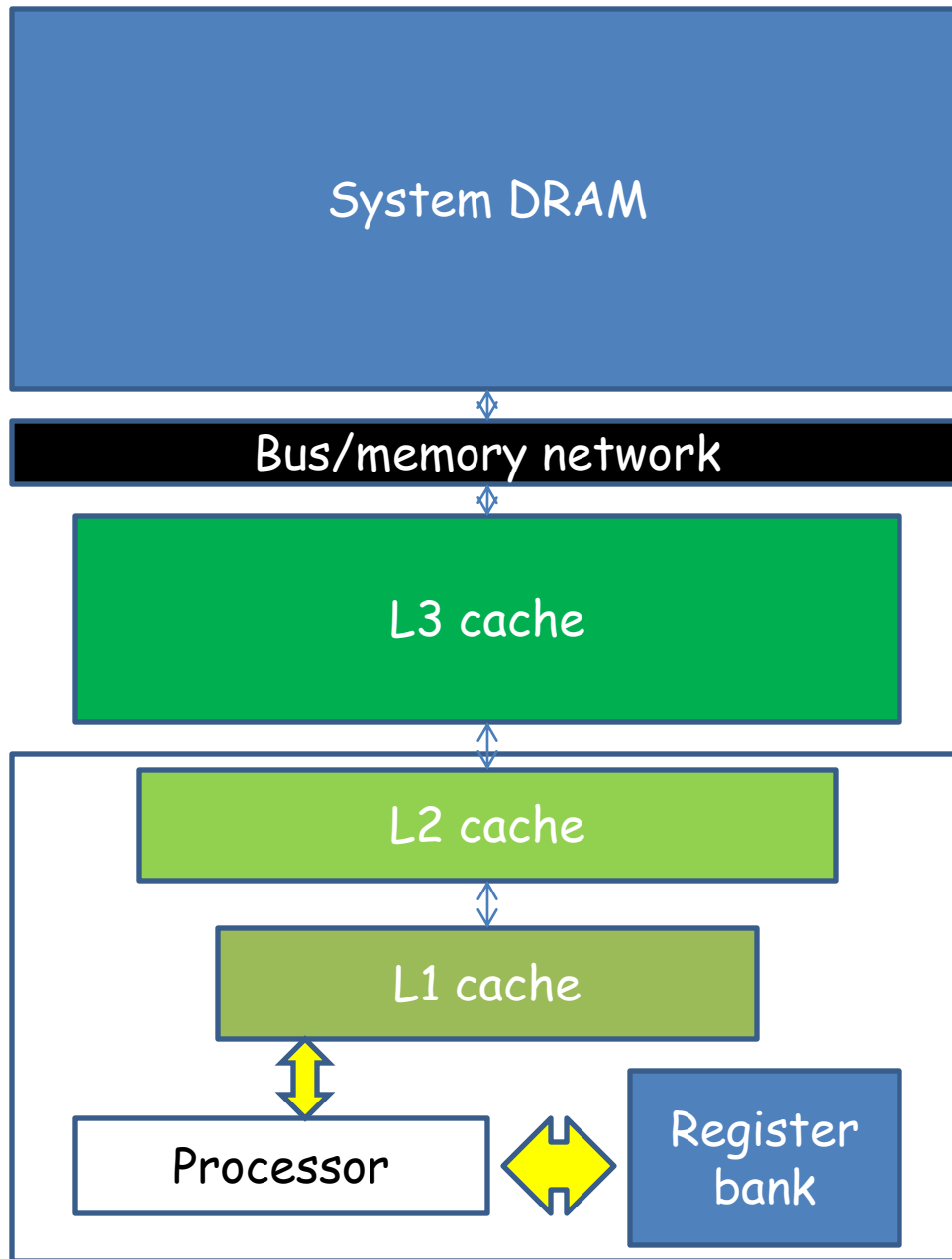


Exploit temporal and spatial **locality** often present in programs



- Several levels of caches
- Banked memories
- Memory network for multiprocessors



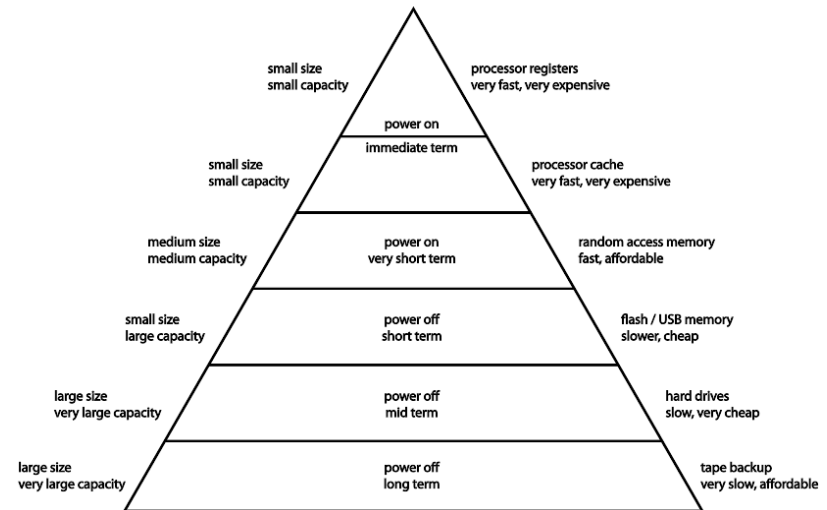


Typical HPC/data center server

Typical values for memory hierarchy:

Registers:	0 cycles
L1 cache:	1 cycles
L2 cache	10 cycles
L3 cache	30 cycles
Main memory:	100 cycles
Disk:	100,000 cycles
Tape:	10,000,000 cycles

Computer Memory Hierarchy



[Bryant, O'Halloran: Computer Systems, Prentice-Hall, 2011]

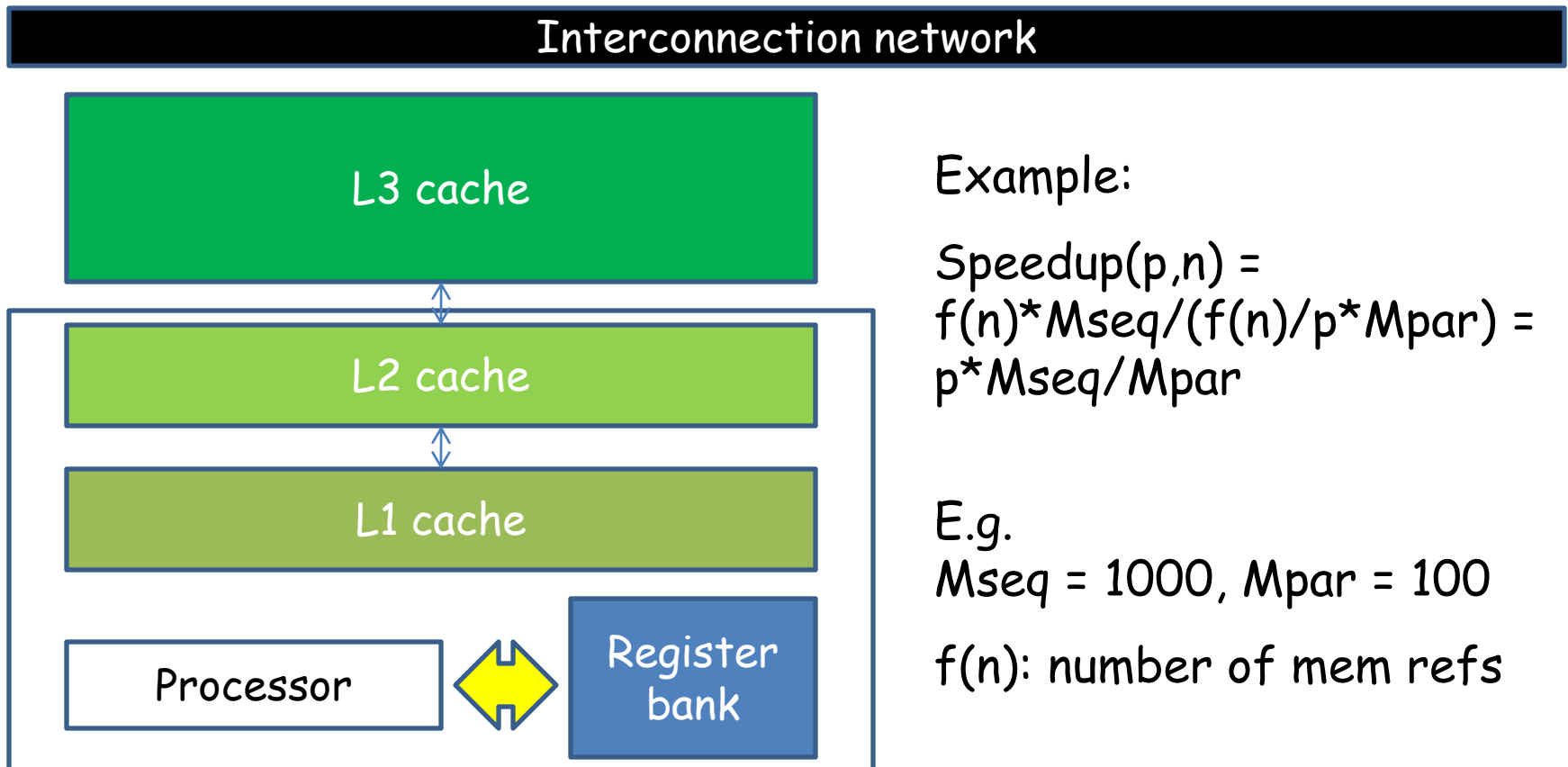
Sequential algorithm on huge data size n , that needs to use full memory hierarchy

vs.

Parallel algorithm on distributed data n/p where each processor may work on data in main memory, or even cache

Non-trivial parallel algorithm
needs to communicate, trades
memory refs for communication

Processor j , $0 \leq j < n$



Example:

$$\text{Speedup}(p,n) = \frac{f(n) * M_{\text{seq}}}{(f(n)/p * M_{\text{par}})} = p * M_{\text{seq}} / M_{\text{par}}$$

E.g.

$$M_{\text{seq}} = 1000, M_{\text{par}} = 100$$

$f(n)$: number of mem refs

Observation/lesson:

In Scientific Computing/HPC Speedup often not relevant, problem too large to fit in memory of single processor

Instead:

scalability - can the algorithm scale (strongly or weakly) from 100 to 10,000 processor cores? How much must the problem size increase

Note:

HPC systems may have constant or slowly declining memory/processor as p grows; not reasonable to expect that memory/processor grows with p , $\Omega(p)$

Advanced note:

A programming interface that requires each process to keep state information for all other processes will be in trouble as p grows

[Balaji et al.: MPI on millions of cores. Parallel Proc. Letters, 21(1),45-60,2011]

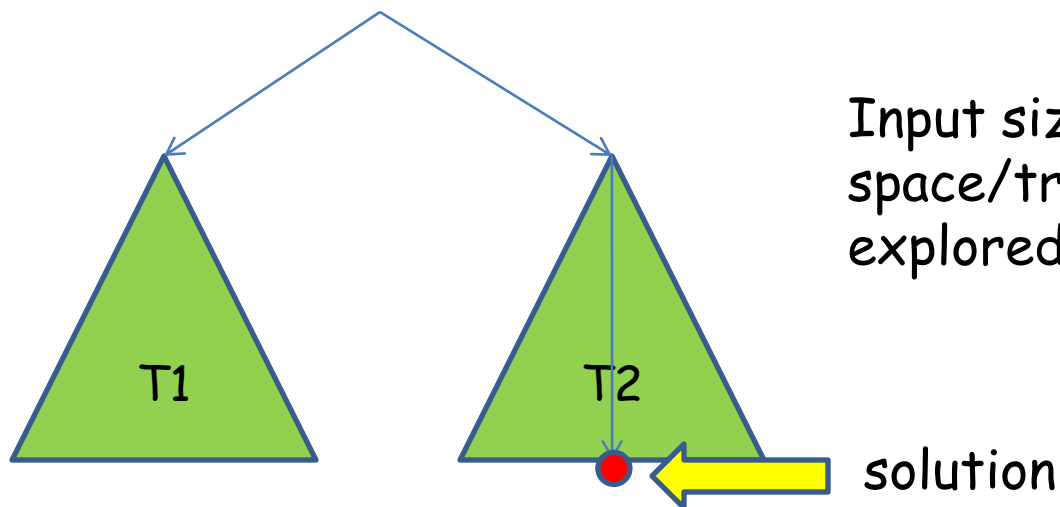
Sources of superlinear speedup:

Differences between sequential and parallel hardware:

- a) the memory hierarchy
- b) Algorithmic reasons

Example: tree search

Sequential algorithm explores all of T1 and **one path** in T2

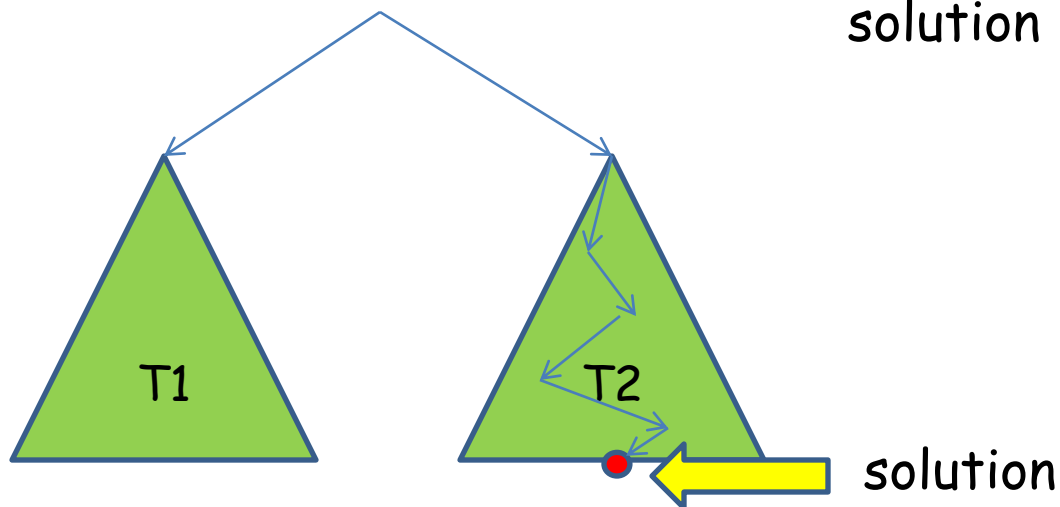


Input size n , solution space/tree of size 2^n to be explored

$$T_{seq}(n) = T1(n) + t2(n) = O(2^{(n-1)}) + O(n-1) = c1 \cdot 2^{(n-1)} + c2 \cdot (n-1)$$

Example: tree search

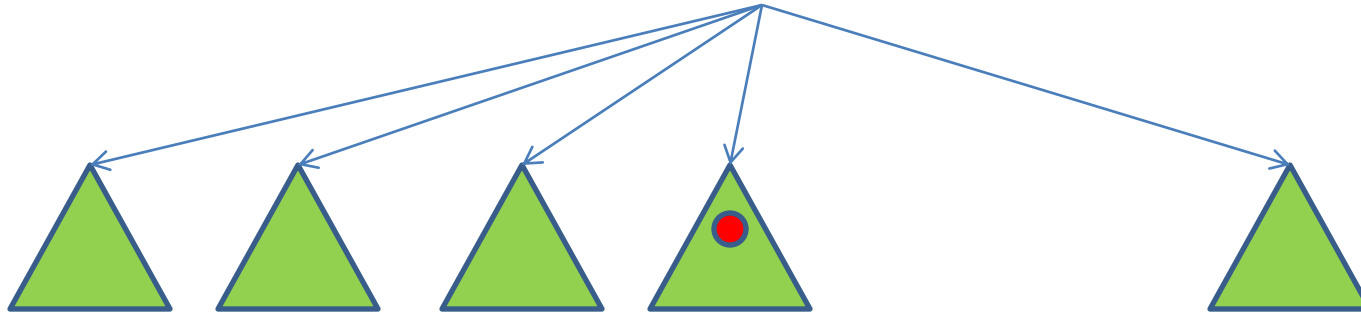
Parallel algorithm explores T1 and T2 in parallel, terminates as soon as solution is found



$$T_{\text{par}}(2, n) = t'_2(n) = O(n) = c_3 \cdot n$$

$$\text{Speedup}(p, n) = (c_1 \cdot 2^{(n-1)} + c_2 \cdot (n-1)) / c_3 \cdot n = c' \cdot 2^{(n-1)} / n + 1 \rightarrow \infty \text{ as } n \text{ grows}$$

General: p subtrees, explored in parallel, termination as soon as solution is found in one



Superlinear speedup often found in parallel branch-and-bound search algorithms (solution of hard problems)

Sources of superlinear speedup:

Differences between sequential and parallel hardware:

- a) The memory hierarchy
- b) Algorithmic reasons
- c) Non-determinism
- d) Randomization

b)-d): sequential and parallel algorithms are not doing the same things

Reduction, prefix sums

Reduction problem: given sequence $x_0, x_1, x_2, \dots, x_{(n-1)}$, compute

$$y = \sum x_i = x_0 + x_1 + x_2 + \dots + x_{(n-1)}$$

- x_i can be integers, real numbers, vectors, ...
- „+“ can be some applicable operator, sum, product, min, max, bitwise and, logical and, vector sum, ...

Algebraic properties of „+“: **associative**, commutative, ...

Reduction operations in programming models/languages

Set of processes „collectively“ invoke „reduce“ operation, each contribute a subset of the n elements

- **Reduction to one**: all processes participate in the operations, resulting „sum“ stored with one process
- **Reduction to all**: all processes participate, results available to all processes
- **Reduction with scatter**: reduction of vectors, result vector stored in blocks over the processes

Prefix sum: sum of the first i elements of x_i sequence

$$y_i = \sum_{0 \leq j < i} x_j = x_0 + x_1 + x_2 + \dots + x_{(i-1)}$$

Exclusive prefix ($i > 0$): x_i **not** included in sum

$$y_i = \sum_{0 \leq j \leq i} x_j = x_0 + x_1 + x_2 + \dots + x_{(i-1)}$$

Inclusive prefix: x_i **included** in sum.

Note: inclusive prefix trivially computed from exclusive prefix (add x_i), not vice versa unless „+“ has inverse

Parallel prefix sums problem: compute all prefix sums $y_0, y_1, \dots, y_{(n-1)}$

Prefix/scan operations in programming models/languages

Set of processes „collectively“ invoke „reduce“ operation, each contribute a subsequence/segment of the n elements

- **Scan**: all **inclusive prefix** sums for process's segment computed at process
- **Exscan**: all **exclusive prefix** sums for process's segment computed at process

Reductions and prefix-sums/scans typically found in parallel languages/interfaces. A parallel programming model can be defined around the concept of collective operations

Sequential, simple scan through array

```
y[1] = x[0];  
for (i=2; i<n; i++) {  
    y[i] = y[i-1]+x[i-1];  
}  
sum = y[n-1]+x[n-1]; // reduction
```

Parallel?

$T_{seq}(n) = n-1$ summations

Application: cutoff computation

```
// Parallelizable part
do {
  for (i=0; i<n; i++) {
    x[i] = f(i);
  }
  // check for convergence
  done = ...;
} while (!done)
```

done: if $x[i] < \epsilon$ for all i

Each process locally computes

localdone = $(x[i] < \epsilon)$ for all local i

done = allreduce(localdone, AND);

Application: array compaction, load balancing

Given arrays *a* and *active*, execute data parallel loop efficiently in parallel:

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

Work $O(n)$, although number of active elements may be much smaller. Assume *f* an expensive operation

```
for (i=0; <n; i++) index[i] = active[i] ? 1 : 0;
Exscan(index,n); // exclusive prefix computation
m = index[n-1]+(active[n-1] ? 1 : 0);
for (i=0; i<n; i++) {
    if (active[i]) {
        aa[index[i]] = a[i];
        bb[index[i]] = b[i];
        cc[index[i]] = c[i];
    }
}
for (i=0; i<m; i++) {
    aa[i] = f(bb[i]+cc[i]);
}
```

index: **0 1 0 0 0 1 1 0 0 1 0 1 1 0 0 1**

Exscan 

0 0 1 1 1 1 2 3 3 3 4 4 5 6 6 6

Application: partitioning for Quicksort

Task parallel Quicksort algorithm

Quicksort(a, n):

1. Select pivot $a[k]$
2. Partition a into $a[0, \dots, n_1-1]$, $a[n_1, \dots, n_2-1]$, $a[n_2, \dots, n-1]$ of elements smaller, equal, and larger than pivot
3. In parallel: Quicksort(a, n_1), Quicksort($a+n_2, n-n_2$)

Partition:

1. Mark elements smaller than $a[k]$, compact into $a[0, \dots, n_1-1]$
2. Mark elements equal to $a[k]$, compact into $a[n_1, \dots, n_2-1]$
3. Mark elements greater than $a[k]$, compact into $a[n_2, \dots, n-1]$

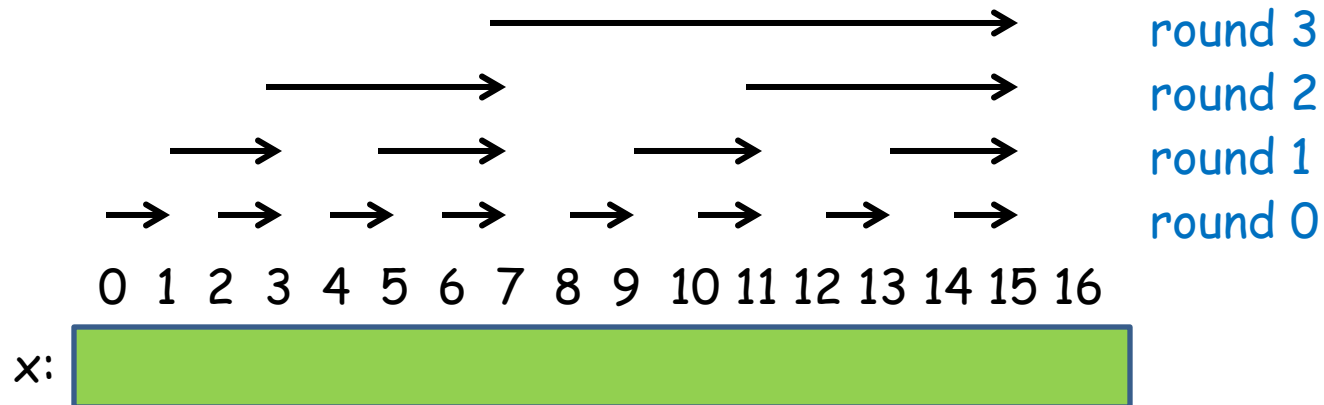
```
for (i=0; i<n; i++) index[i] = (a[i]<a[k]) ? 1 : 0;
Exscan(index, n); // exclusive prefix computation
for (i=0; i<n; i++) {
    if (a[i]<a[k]) aa[index[i]] = a[i];
}
...
```

...and many other (less trivial) applications

Parallel solution?

Key: + is associative

$$x_0 + x_1 + x_2 + \dots + x_{(n-2)} + x_{(n-1)} = ((x_0 + x_1) + (x_2 + \dots)) + \dots + (x_{(n-2)} + x_{(n-1)})$$



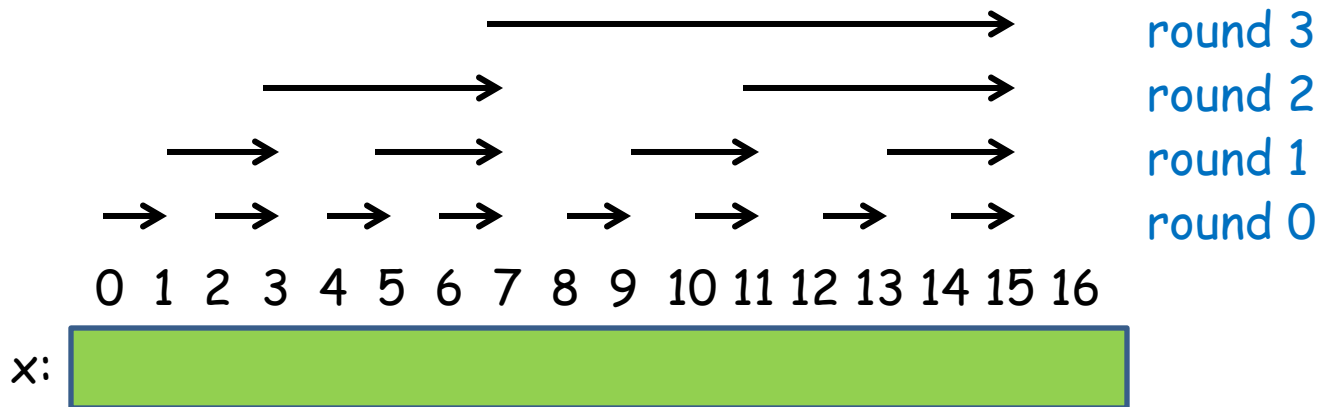
And almost done, $x[2^k - 1] = \sum_{0 \leq i < 2^k} x_i$

Lemma:

Reduction can be performed out in $r = \log_2 n$ synchronized rounds, for n a power of 2. Total number of + operations are $n/2 + n/4 + n/8 + \dots < n$

Recall, geometric series: $\sum_{(0 \leq k \leq n): ar^k} = a(1 - r^{(n+1)}) / (1 - r)$

- Shared memory (programming) model: **synchronization** after each round
- Distributed memory programming model: \longrightarrow represents **communication**



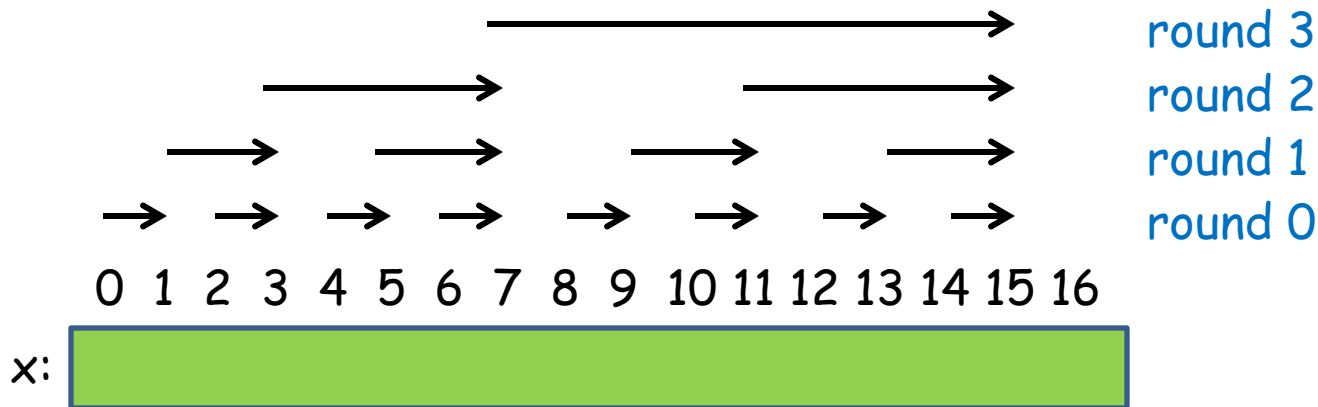
```

for (k=1; k<n; k=kk) {
  kk = k<<1; // double
  for (i=kk-1; i<n, i+=kk) {
    x[i] = x[i-k]+x[i];
  }
  barrier;
}

```

← Data parallel loop,
 $n/2^{(k+1)}$ operations for
round r , $r=0, 1, \dots$

↑
**But beware of
dependencies**



```

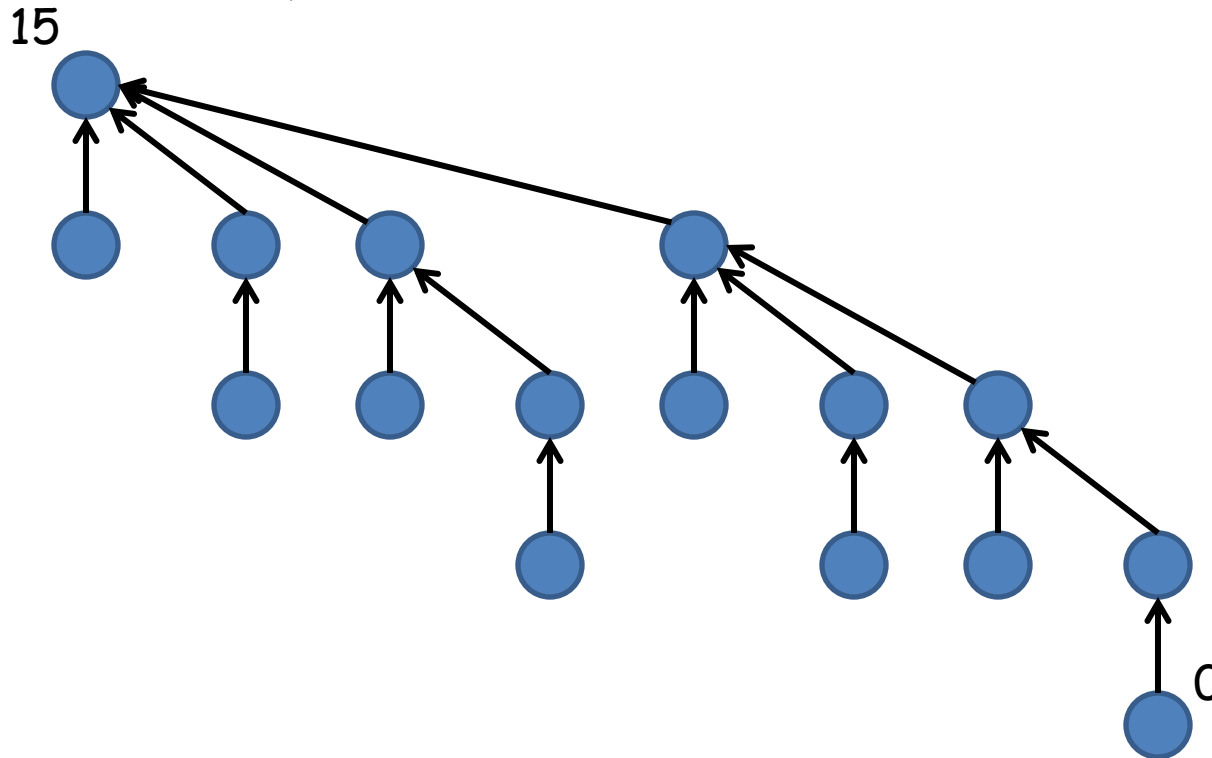
for (k=1; k<n; k=kk) {
  kk = k<<1; // double
  for (i=kk-1; i<n, i+=kk) {
    x[i] = x[i-k]+x[i];
  }
  barrier;
}

```

← Data parallel loop,
 $n/2^{(k+1)}$ operations for
round $r, r=0, 1, \dots$

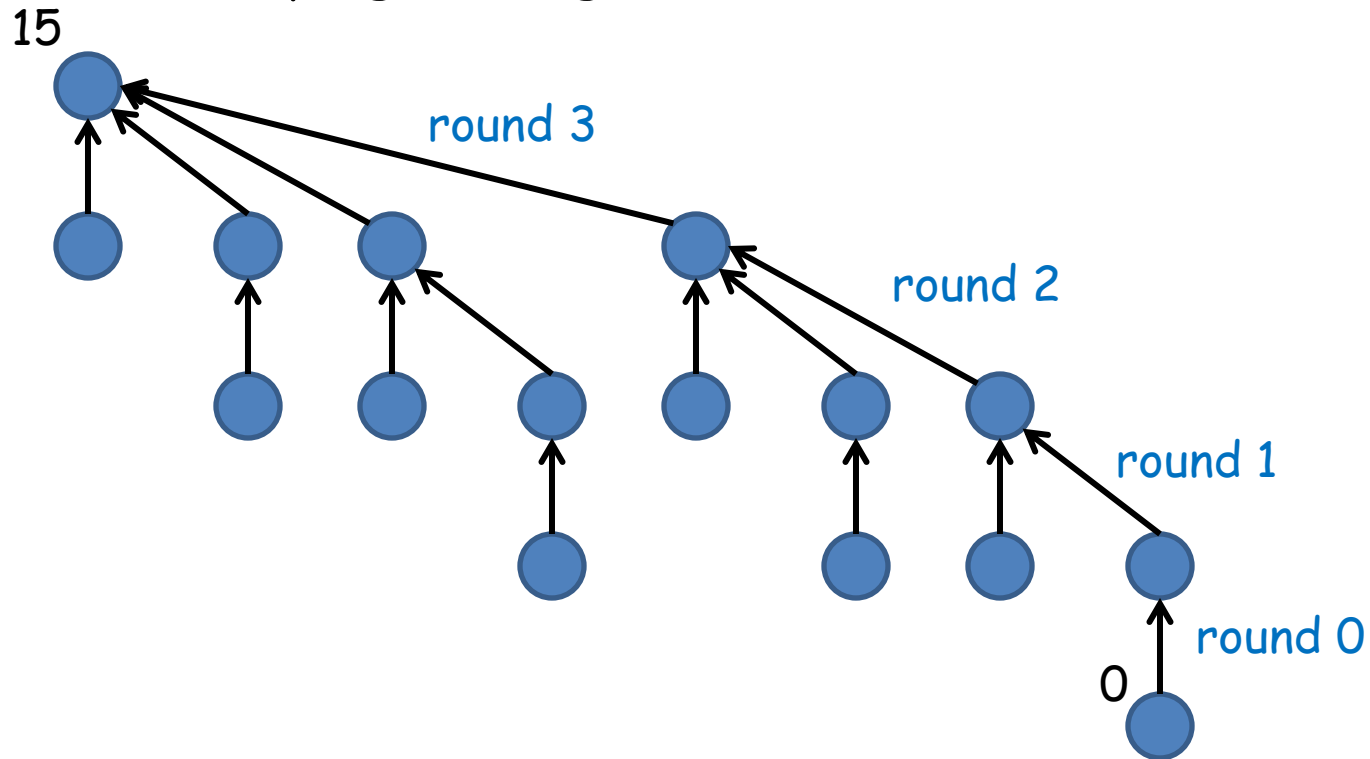
Here
unproblematic, why?

Distributed memory, message passing
(programming) model: arrows are communication



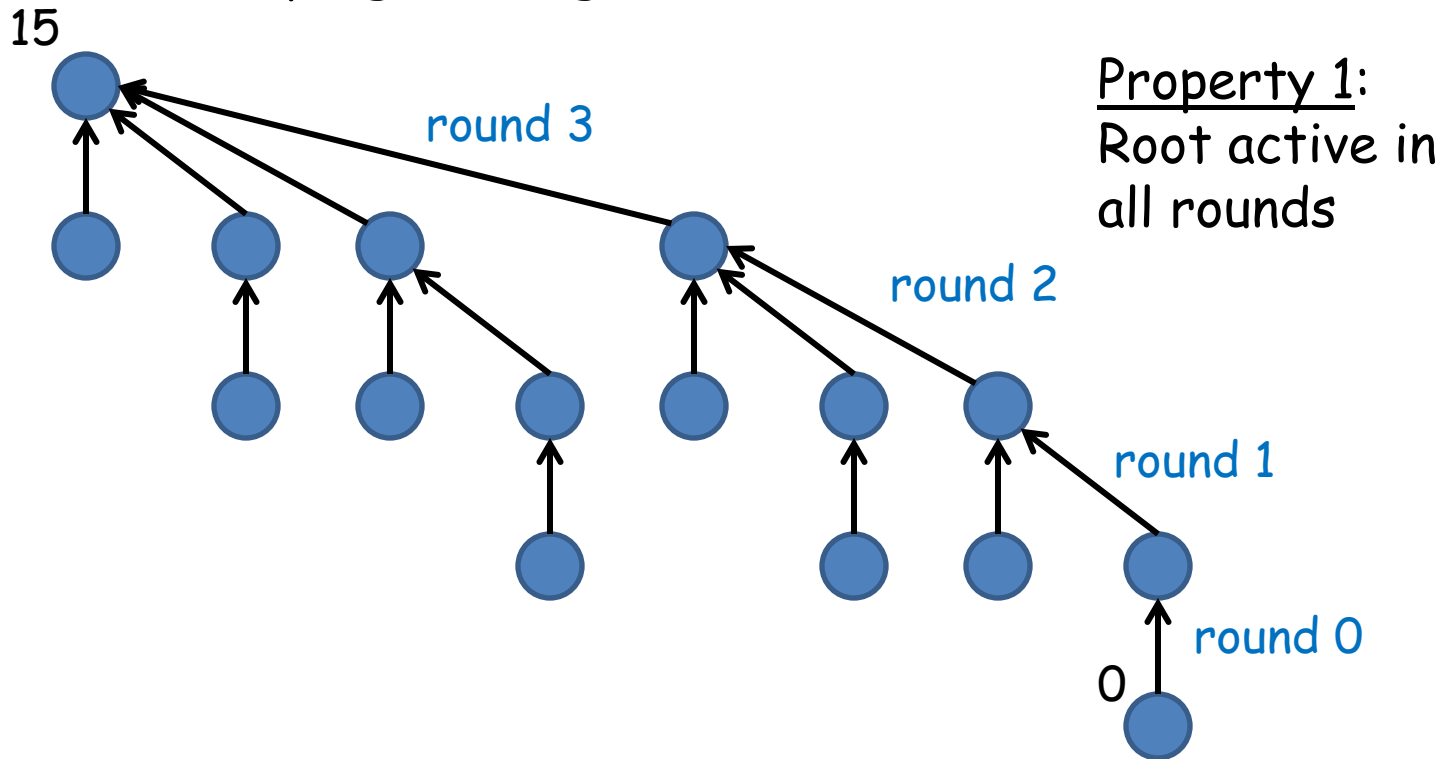
Communication pattern of the algorithms forms a **binomial tree**

Distributed memory, message passing
(programming) model: arrows are communication



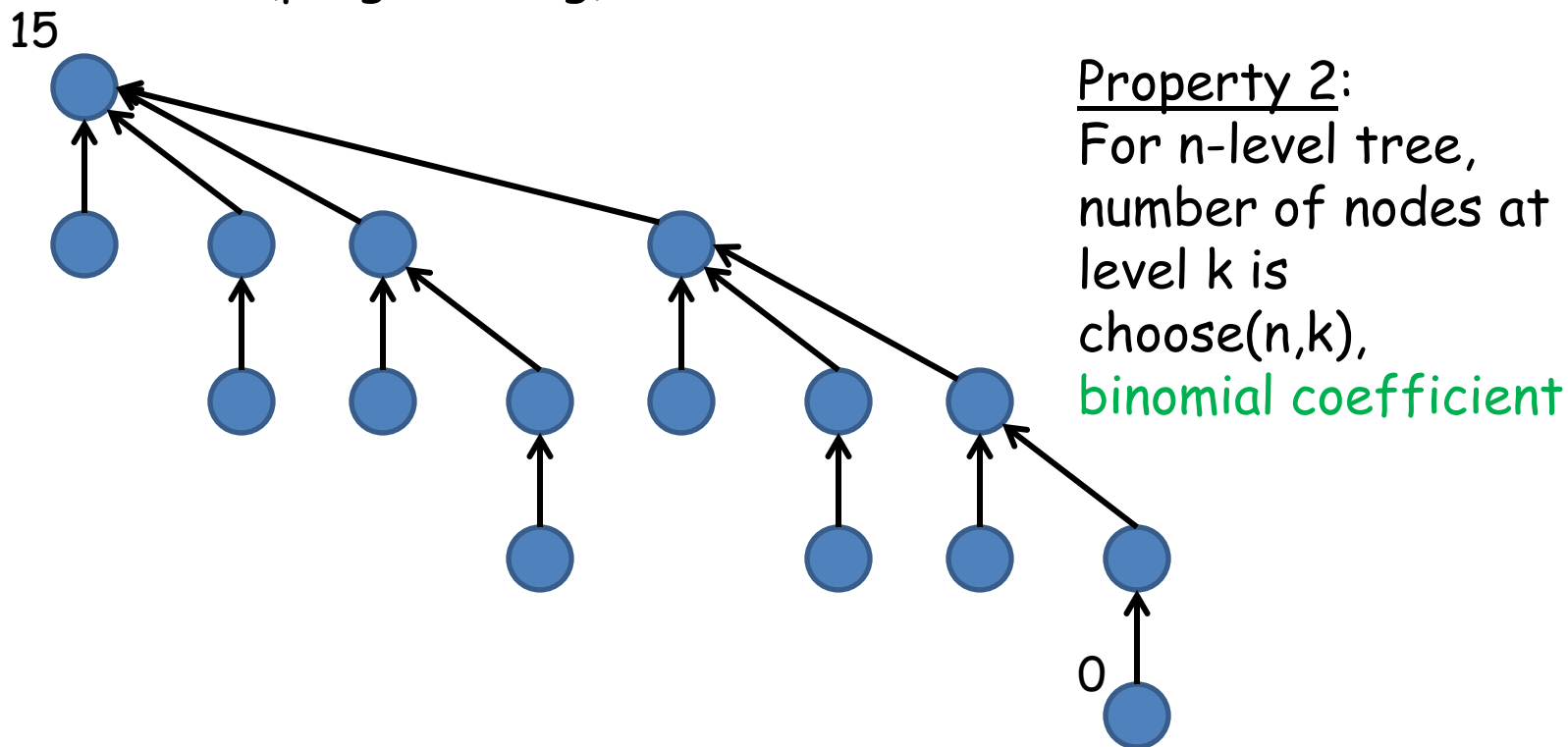
Communication pattern of the algorithms forms a **binomial tree**

Distributed memory, message passing
(programming) model: arrows are communication



Communication pattern of the algorithms forms a **binomial tree**

Distributed memory, message passing
(programming) model: arrows are communication



Communication pattern of the algorithms forms a binomial tree

Problems:

- n not a power of 2?
- parallel prefix sums?

Observation/invariant: X original content of array x
After round k, $k=0, \dots, \log n$

$$x[i \cdot 2^{(k+1)} - 1] = X[i \cdot 2^{(k+1)} - 1 - 2^k] + \dots + X[i \cdot 2^{(k+1)} - 1]$$

Last update on $x[i]$ in round k where $i \neq j \cdot 2^{(k+1)} - 1$

Prefix sums for certain segments computed, use $\log p$ rounds to hand on

Recursive formulation

```
Scan(x, n)
{
  if (n==1) return;

  for (i=0; i<n/2; i++) y[i] = x[2*i]+x[2*i+1];

  Scan(y, n/2);

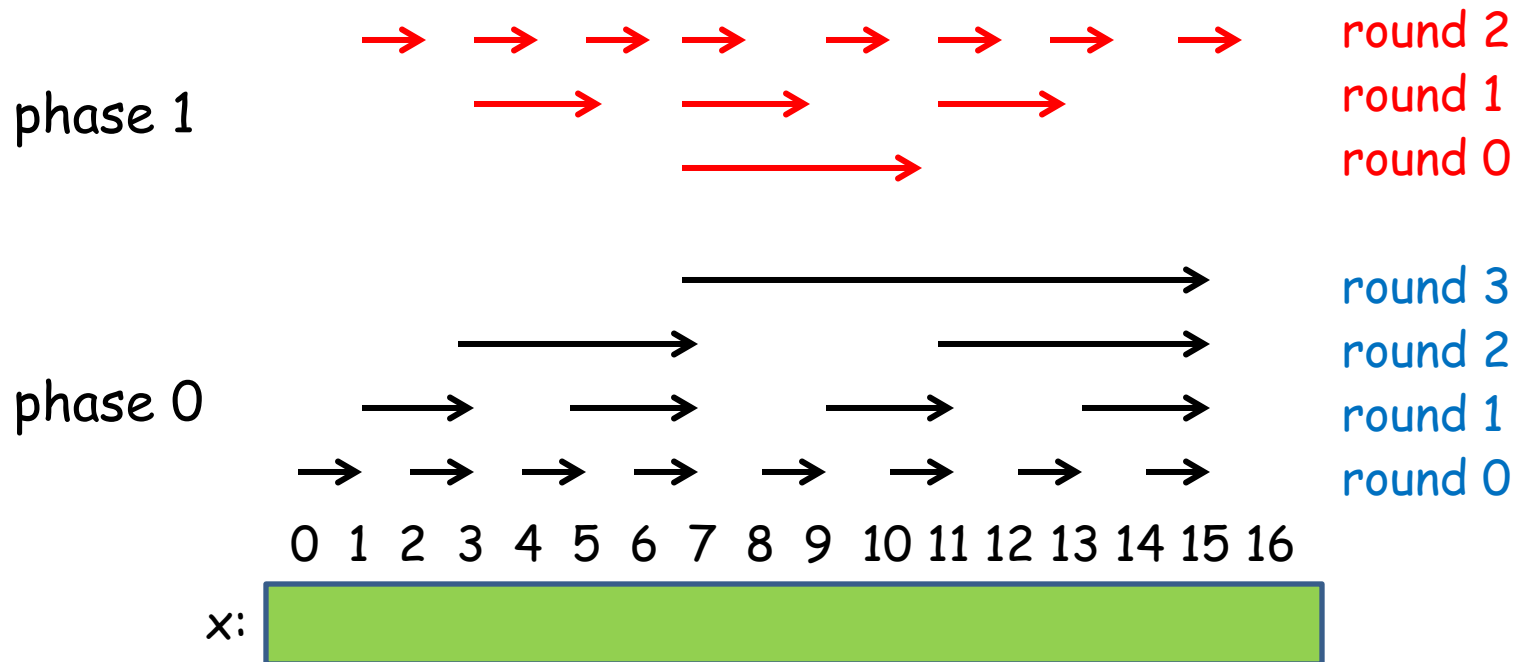
  x[1] = y[0];
  for (i=1; i<n/2; i++) {
    x[2*i] = y[i-1]+x[2*i];
    x[2*i+1] = y[i];
  }
  if (odd(n)) x[n-1] = y[n/2-1]+x[n-1];
}
```

Reduce problem

Solve recursively

Take back

What the recursive algorithm does:



Non-recursive, data parallel implementation

```

for (k=1; k<n; k=kk) {
  kk = k<<1; // double
  for (i=kk-1; i<n, i+=kk) {
    x[i] = x[i-k]+x[i];
  }
  barrier;
}

```

„up-phase“:
 $\log_2 n$ rounds,
 $n/2+n/4+n/8+\dots < n$
 summations

This could be data dependencies, but are not

```

for (k=k>>1; k>1; k=kk) {
  kk = k>>1; // halve
  for (i=k-1; i<n-kk; i+=k) {
    x[i+kk] = x[i]+x[i+kk];
  }
  barrier;
}

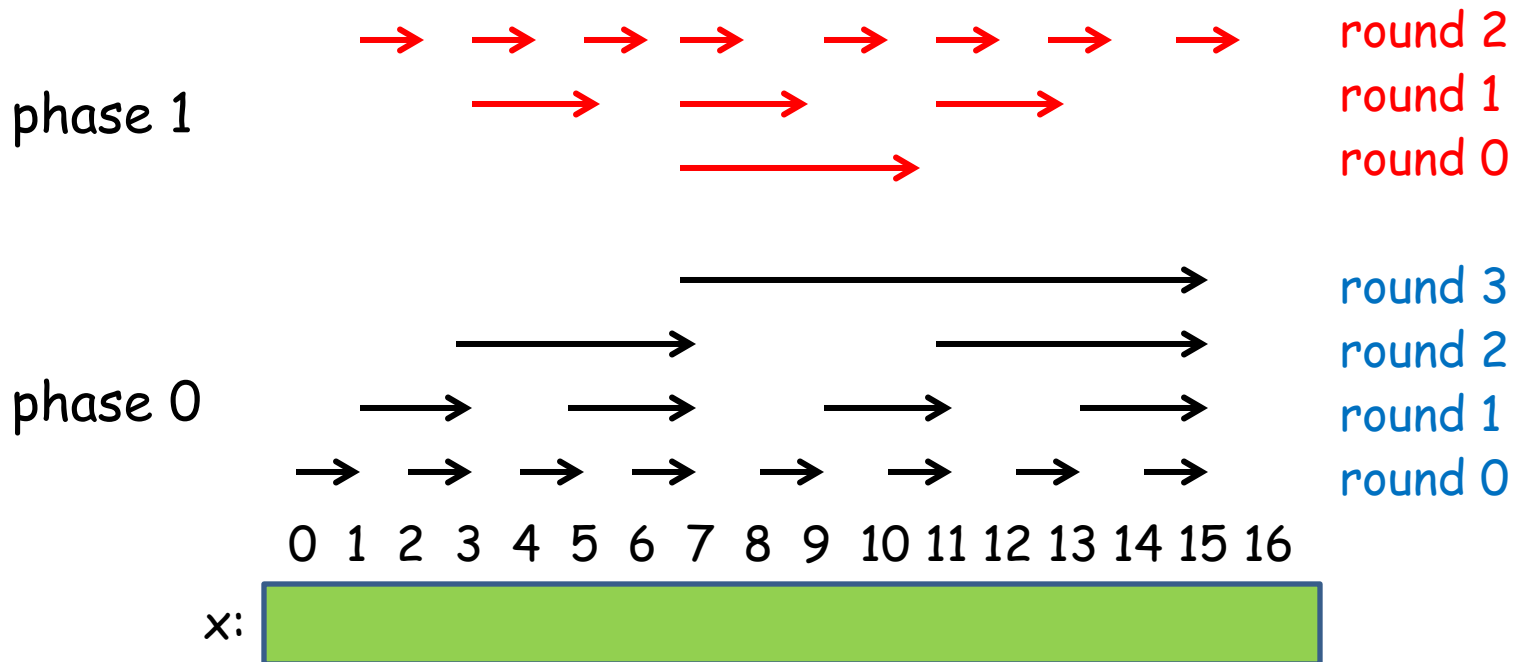
```

„down phase“:
 $\log_2 n$ rounds,
 $n^2/2+n/4+n/8+\dots < n$
 summations

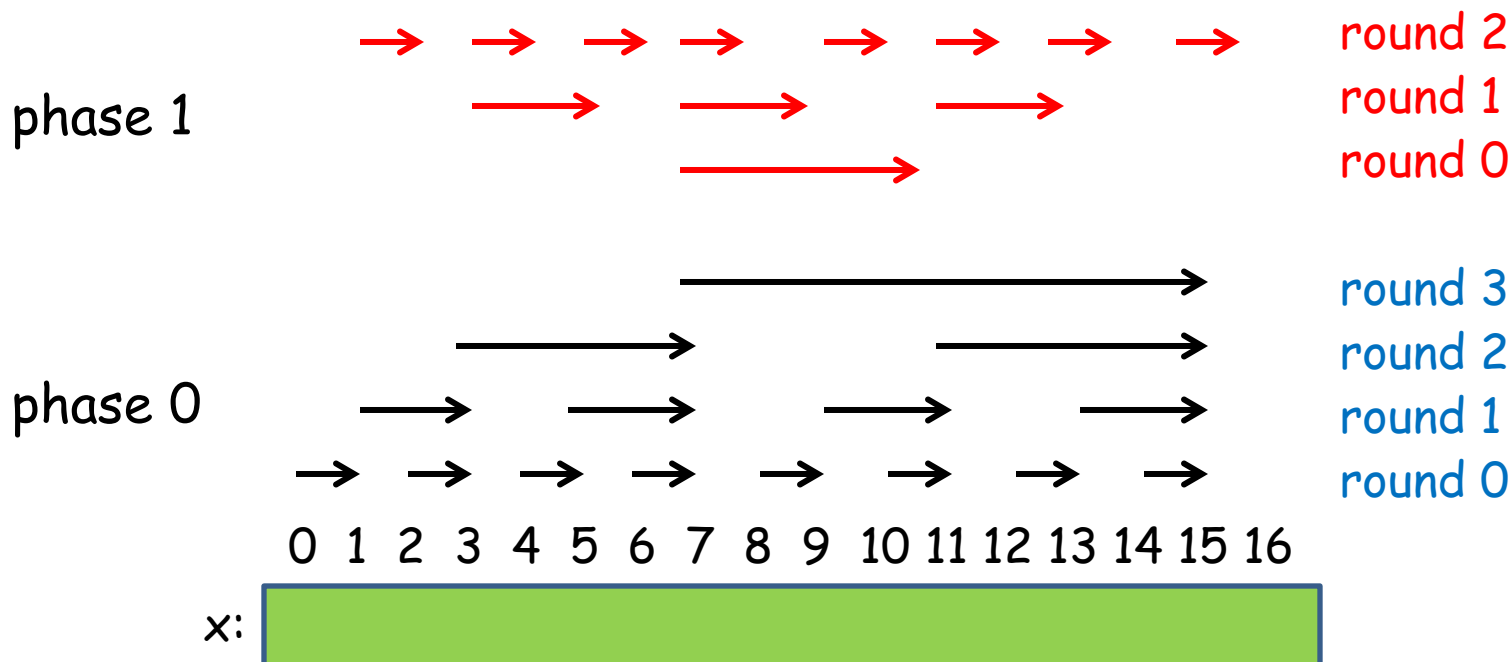
Total work $\approx 2n = O(T_{seq}(n))$

But: factor 2 off!!

Speedup(p) at most $p/2$ - half the processors are lost



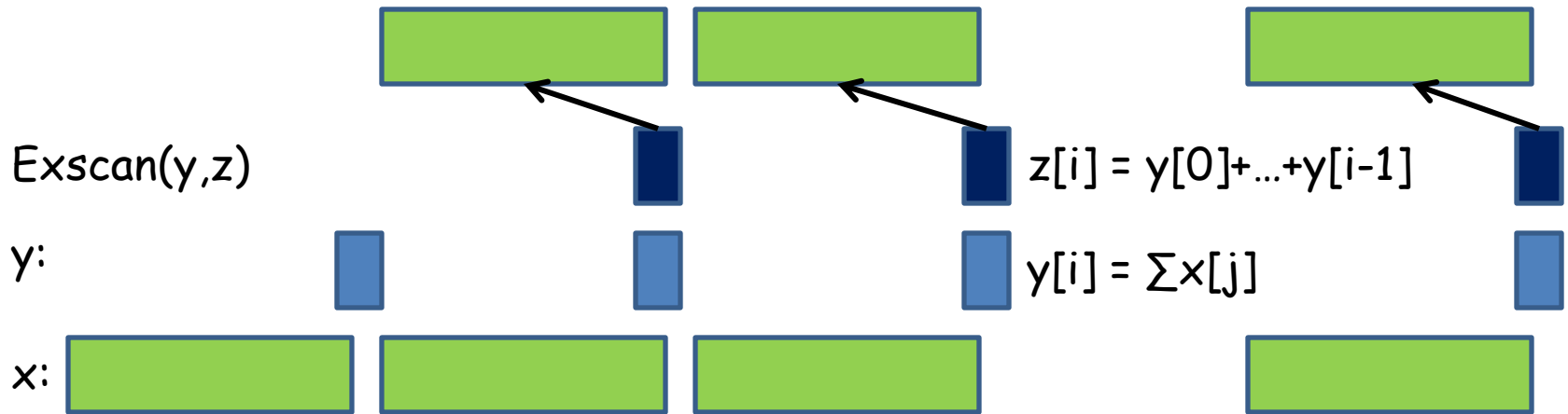
For $p=n$: work optimal, but not **cost optimal** - p processors occupied in $2\log p$ rounds = $O(p \log p)$



Strategy for distributed memory models

1. Each process has block $x[0, \dots, n/p-1]$
2. Compute prefix sums locally, store local sum
3. Exscan(local sums);
4. Add exclusive prefix locally to all $x[0, \dots, n/p-1]$

Observation: total work is $2*n + p \log p$, **twice** $T_{seq}(n)$



Lesson: work-optimal parallel algorithms often have larger constant factors than best sequential algorithm. **Inherently?**

Yet another data parallel prefix-sums algorithm

```
for (k=1; k<n; k<<=1) {  
  for (i=k; i<n; i++) x[i] = x[i-k]+x[i];  
  barrier;  
}
```



Why might it not work? Dependencies!!

- Why does this work? Invariant?
- All indices active in all rounds, work $O(n \log n)$
- But **only** $\log n$ rounds

[Hillis, Steele: Data Parallel Algorithms. *CACM* 29(12), 1170-1183, 1986]

Other „collective“ operations

Mostly for distributed memory programming models. A subset of processes collectively carry out operation

- **Broadcast**: one process has data, after operation all processes have data
- **Scatter**: data of one process distributed in blocks to other processes
- **Gather**: blocks from all processes collected at one process
- **Allgather**: blocks from all processes collected at all processes
- **Broadcast-to-all**: **same**
- **Alltoall**: each process has blocks of data, one block for each other process

For performance reasons (locality), can make sense also in shared memory programming and architecture models