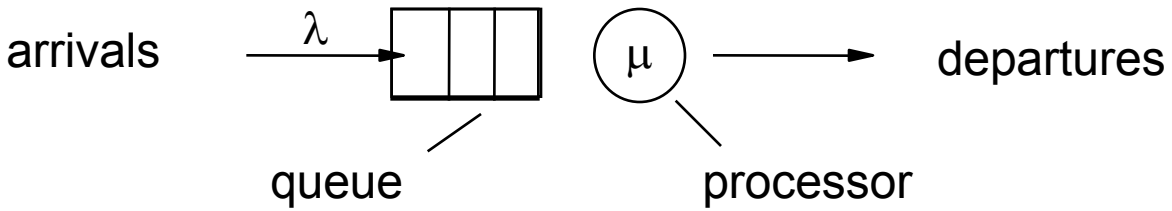
Chapter 3

Performance aspects

3.1 Multicomputer and Multiprocessor Systems

- Using queuing theory, computer systems can be modeled as stochastic processes.
- A single computer is considered as a service station at which „jobs“ (or customers) arrive, are serviced and leave the system.



- If the processor is occupied, arriving jobs are queued in a FIFO.
- The times between successive arrivals and the service times are stochastic variables.
- In the most simple case, the times follow an exponential distribution with parameter λ (arrival rate) or parameter μ (service rate), respectively. The following must hold: $\lambda < \mu$.
- The mean time between two arrivals is $1/\lambda$, the mean service time is $1/\mu$.

Performance Measures

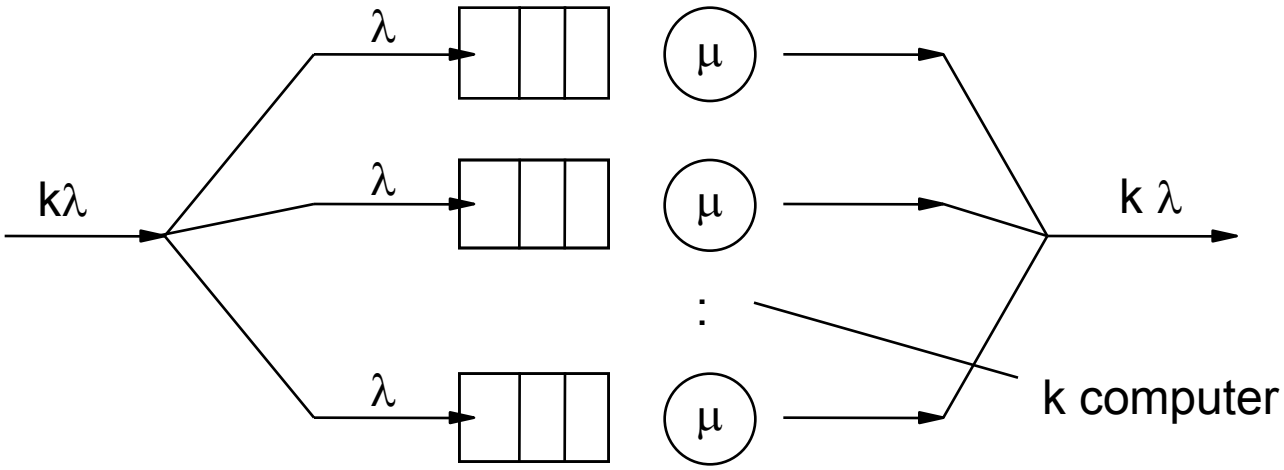
- The basic model is called „M|M|1“-system, with the first parameter indicating the type of distribution of the arrivals, the second parameter characterizing the distribution of the service time and the third parameter indicating the number of processors. M stands for „Markov“ and means exponentially distributed.
- For the **stationary behavior** (steady state) the following can be derived (see e.g. L. Kleinrock: Queuing Theory, Vol.1)
- *Utilization U* (Prob that service station is not idle): $U = \lambda/\mu$
- *Throughput TP* (Number of jobs serviced per time): $TP = \lambda$
- *Mean response time R* (time between arrival and departure):
  $R = 1 / (\mu - \lambda)$
- *Mean number N* (Jobs at station): $N = \lambda / (\mu - \lambda)$
- Between number, response time and arrival rate there exists a relation called *Little‘s law*:

  $N = \lambda \ R$

  „Mean number of jobs = arrival rate x mean response time “

Distributed Systems (Multicomputer Systems)

- A distributed system can approximately be modeled as a set of *k* such M|M|1- service stations.
- All have the same service rate μ and are fed by a job arrival stream with a joint rate of λ.
- Regarding the complete set, we get :
- Throughput: $TP = k\,\lambda$
- Response time: $R_1 = 1 / (\mu - \lambda)$



Multiprocessor System

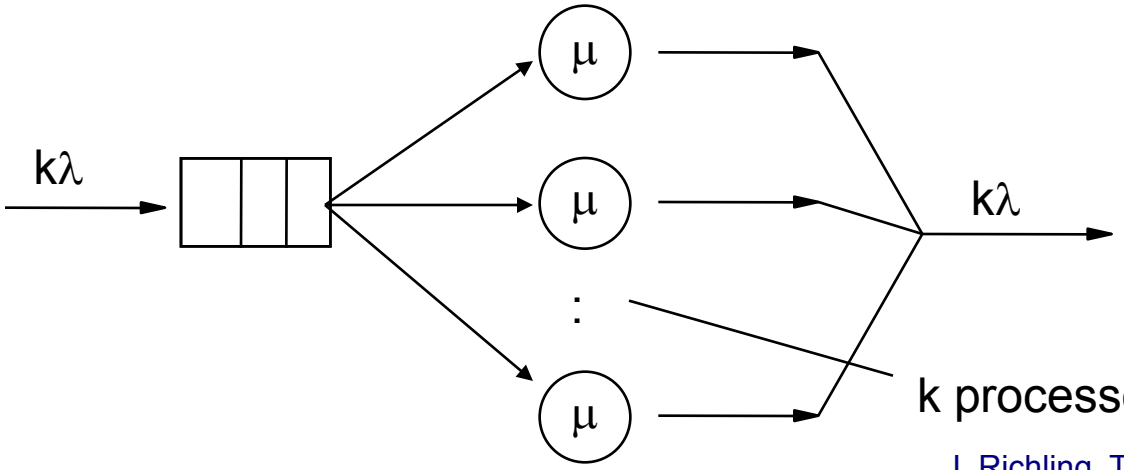
- A multiprocessor on the opposite provides a shared memory and therefore a shared single copy of an operating system instance. All ready threads are kept in a joint ready queue.
- Throughput $TP = k \cdot \lambda$
- Response time

$$R_2 = \frac{1}{\mu} + \frac{(k\lambda/\mu)^k}{k!\,(1-\lambda/\mu)^2} \cdot \frac{1}{\mu} \cdot p_0$$

with

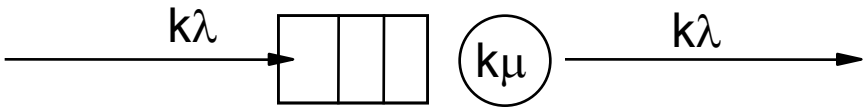
$$p_0 = \left(\sum_{i=0}^{k-1} \left(\frac{k\lambda}{\mu} \right)^i \frac{1}{i!} + \left(\frac{k\lambda}{\mu} \right)^k \frac{1}{k!(1-\lambda/\mu)} \right)^{-1}$$

as the “idle probability”



Fast Monoprocessor System

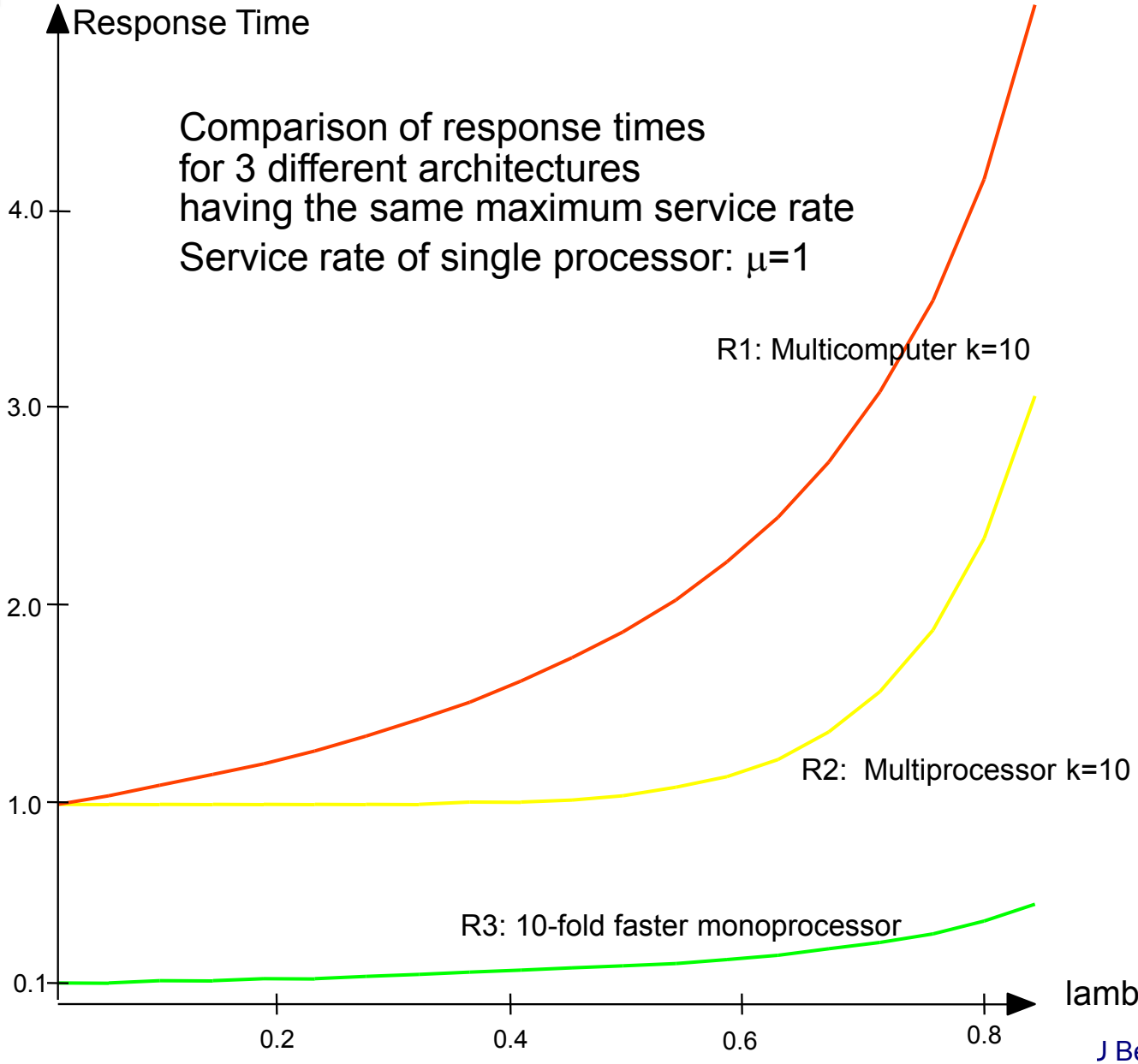
- For comparison, we present the monoprocessor system that has the same peak performance as the parallel one, i.e. a service rate of kμ.



- Throughput $TP = k\,\lambda$

- Response time $R_3 = 1 / (k\mu - k\lambda)$

Comparison


Response Time
Comparison of response times
for 3 different architectures
having the same maximum service rate
Service rate of single processor: μ=1
R1: Multicomputer k=10
R2: Multiprocessor k=10
R3: 10-fold faster monoprocessor
4.0
3.0
2.0
1.0
0.1
0.2
0.4
0.6
0.8
lambda

Explanation:

- Multicomputer and multiprocessor system are close together for light load (small λ) :

  In both cases an arriving job will find an idle processor with high probability.
- At higher loads the missing load balancing in multicomputers shows some effect:

  Due to the random distribution of jobs to processor nodes it can happen that at some nodes jobs are queuing up while other nodes stay idle.

  It is therefore the goal of load balancing mechanisms to achieve the same performance as in multiprocessor systems.
- The k-fold faster monoprocessor can display its speed even when less than k jobs are present. Each job is executed at a k-fold speed.

Conclusion:

- A k-fold faster monoprocessor is better than a parallel system with the same peak service rate with regard to the mean response time.

3.2 Parallel Programs

- Let be

  $T(1)$ the execution time on one processor

  $T(p)$ the execution time on a p processor system

- The gain by parallel computing is expressed by

  $S(p) := T(1) / T(p)$ *Speed-up*

- Normalizing the Speed-up by dividing by the number *p* of processors is defined as the *efficiency*:

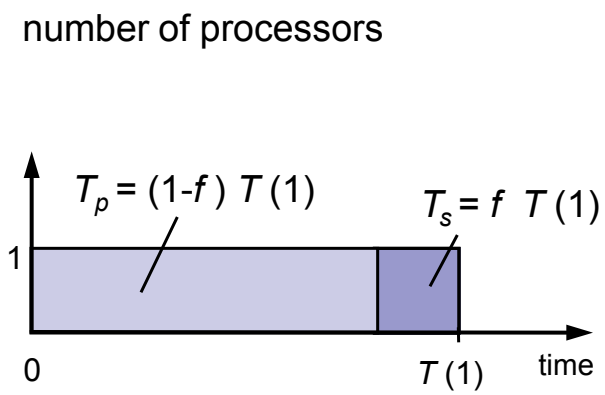
  $E(p) := S(p) / p$ Efficiency

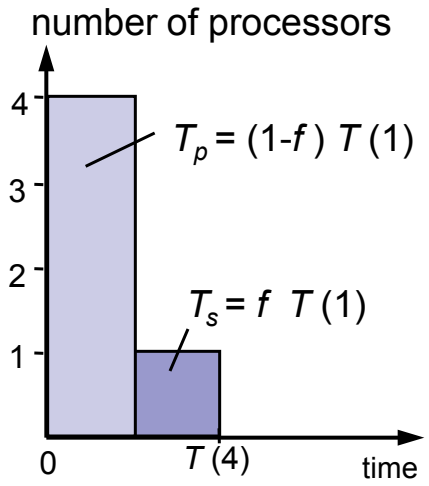
Amdahl's Law

- Parallel programs also contain sequential parts.
- Splitting the execution time into a sequential and a parallel part yields:

  $T(1) = T_s + T_p$
- Let $f := T_s / (T_s + T_p)$, $(0 \le f \le 1)$ be the sequential fraction of the program. Then we get for the speed-up:

$$T(p) = fT(1) + \frac{(1-f)T(1)}{p} = T_s + \frac{T_p}{p} \qquad \text{(Amdahl's Law)}$$





Amdahl's Law

- Using Amdahl's approach we get for the speed-up

$$S(p) = \frac{T(1)}{T(p)} = \frac{p}{fp + (1-f)} = \frac{1}{f + \frac{1-f}{p}}$$

- and for the efficiency

$$E(p) = \frac{S(p)}{p} = \frac{1}{fp + 1 - f}$$

- We also get:

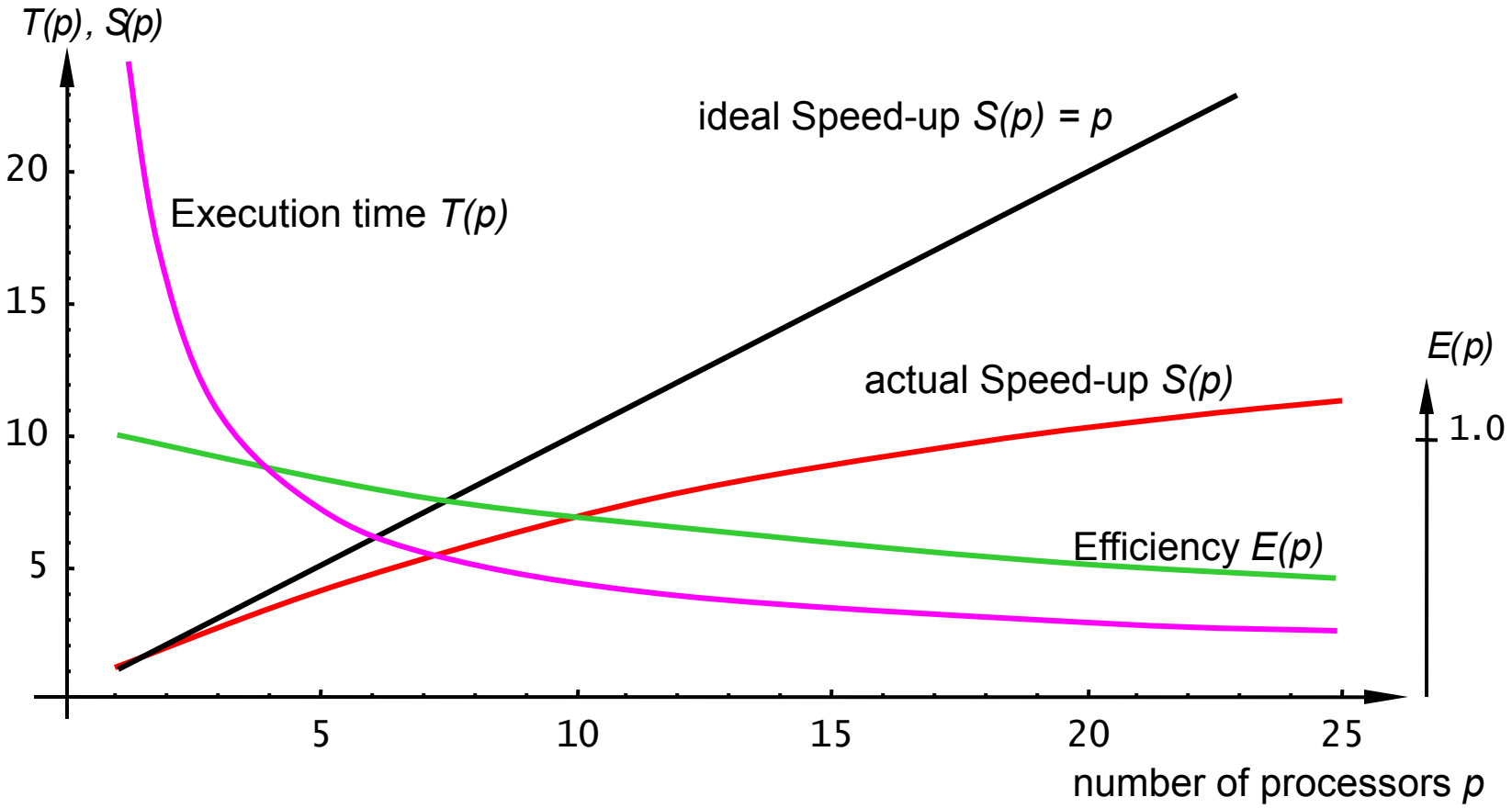
$$T(\infty) := \lim_{p \to \infty} T(p) = fT(1) = T_s$$

$$S(\infty) := \lim_{p \to \infty} S(p) = 1/f$$

$$E(\infty) := \lim_{p \to \infty} E(p) = 0$$

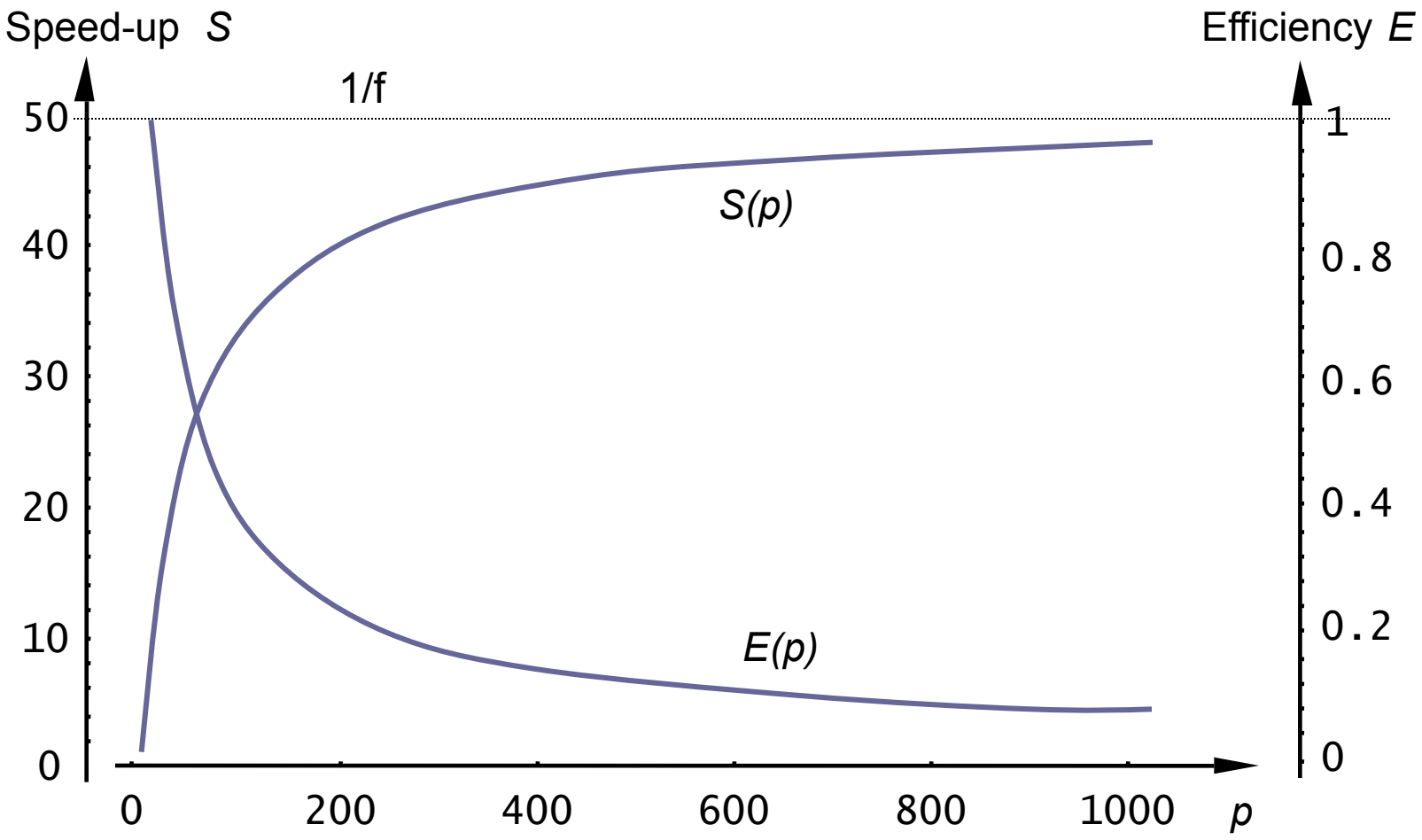
Behavior of different Quantities using Amdahl's Model

for $T(1)=30$ and $f = 0.05$.



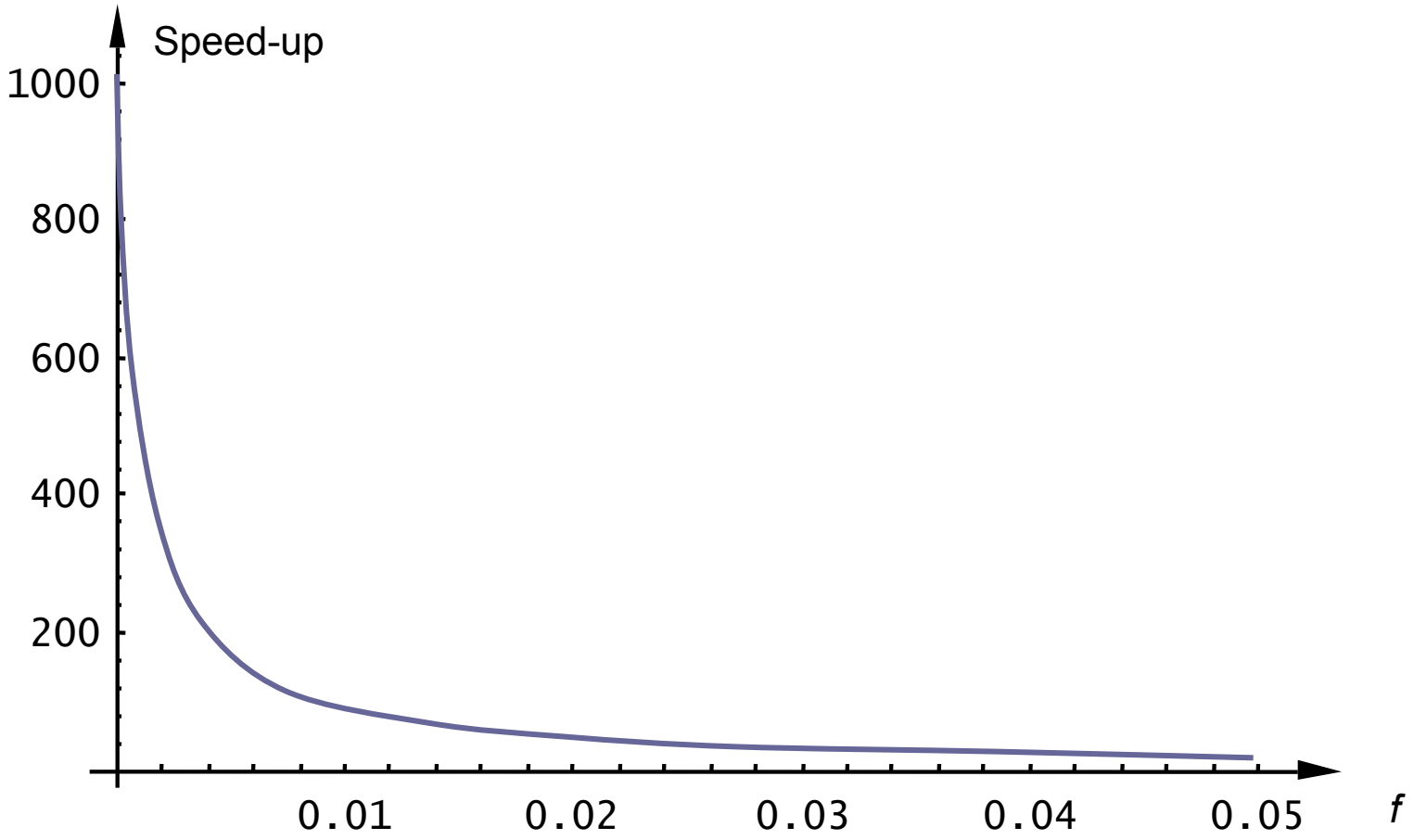
Speed-up *S* and Efficiency *E* as Functions of the number of processors *p*

for $f = 0.02$



Speed-up as a Function of the sequential fraction *f*

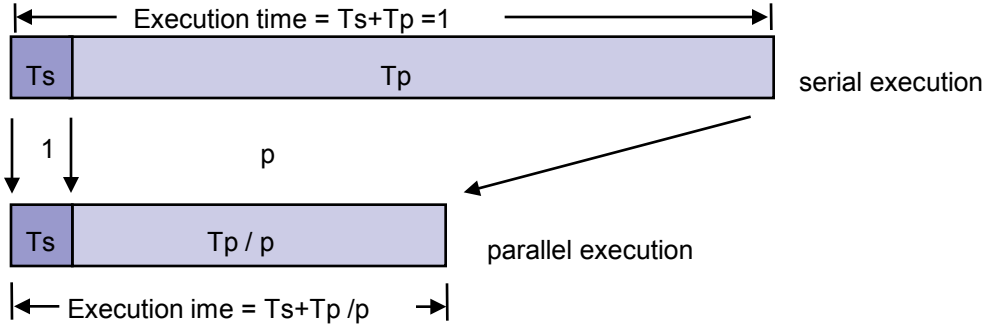
for $p = 1024$.



Scale-up

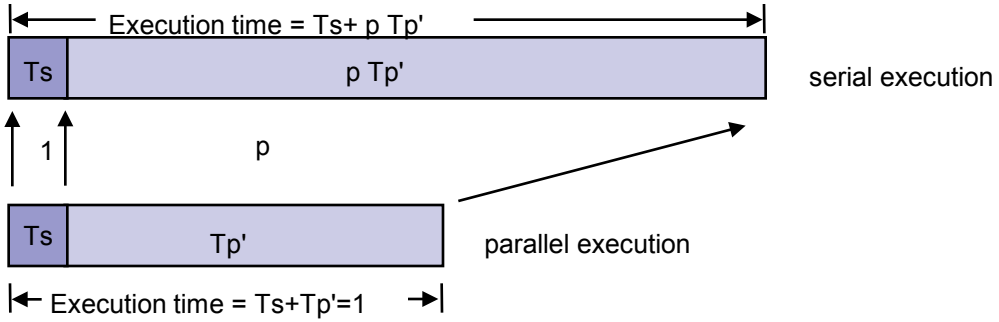
Speed-up:

How much faster can a given problem be solved with *p* processors?



Scale-up (scaled Speed-up):

How much larger can a problem be solved by *p* processors in the same time?



Scale-up

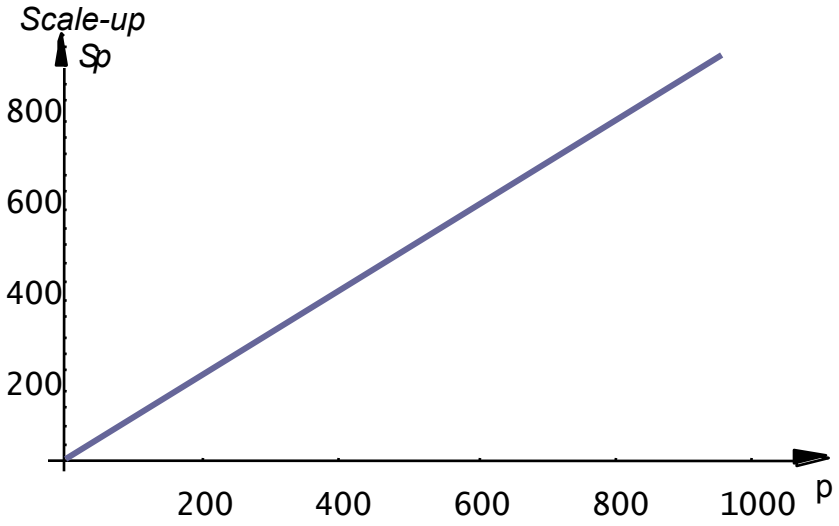
We assume for the scale-up an execution time that is normalized to 1:

$$T(p) = f + (1 - f) = 1$$

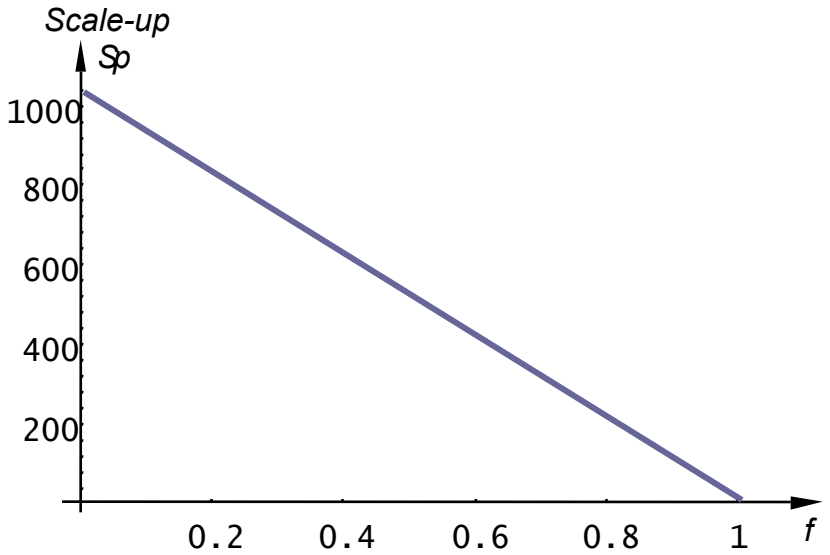
Using only 1 processor, the parallel fraction must be executed sequentially and we obtain $T(1) = f + (1 - f)\, p$

Thus, we get a *scale-up (*or *scaled speed-up*) $S_p(p)$:

$$S_p(p) = T(1)/T(p) = f + (1 - f)\, p = p - (p - 1)\, f$$



S_p as function of p at f = 0,02



S_p as function of sequential fraction f at p = 1024

Size of problem

- When executing a program we solve a problem of some size.
- The ***problem size w*** can be regarded as the total number of instructions to be executed (function of size of input data *n*).
- Let be t_c the time needed to execute one instruction. Then we get:

  $$w\ t_c = T(1)$$

- Let *T(p)* be the running time of a program on *p* processors for a problem size *w.*
- For the execution of a program applied to a problem of size *w* on *p* processors a total work of *p T(p)* is being done.
- Only a fraction of this work is useful work, the rest is „overhead“ T_o.

  $$T_o(p) = p\ T(p) - T(1) \text{ or}$$

  $$T(p) = (T(1) + T_o(p)\ ) / p, \text{ respectively}$$

Problem size

- The speed-up is then

$$S(p) = \frac{T(1)}{T(p)} = \frac{pT(1)}{T(1) + T_o(p)}$$

- The efficiency:

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{T(1) + T_o(p)} = \frac{1}{1 + \frac{T_o(p)}{T(1)}}$$

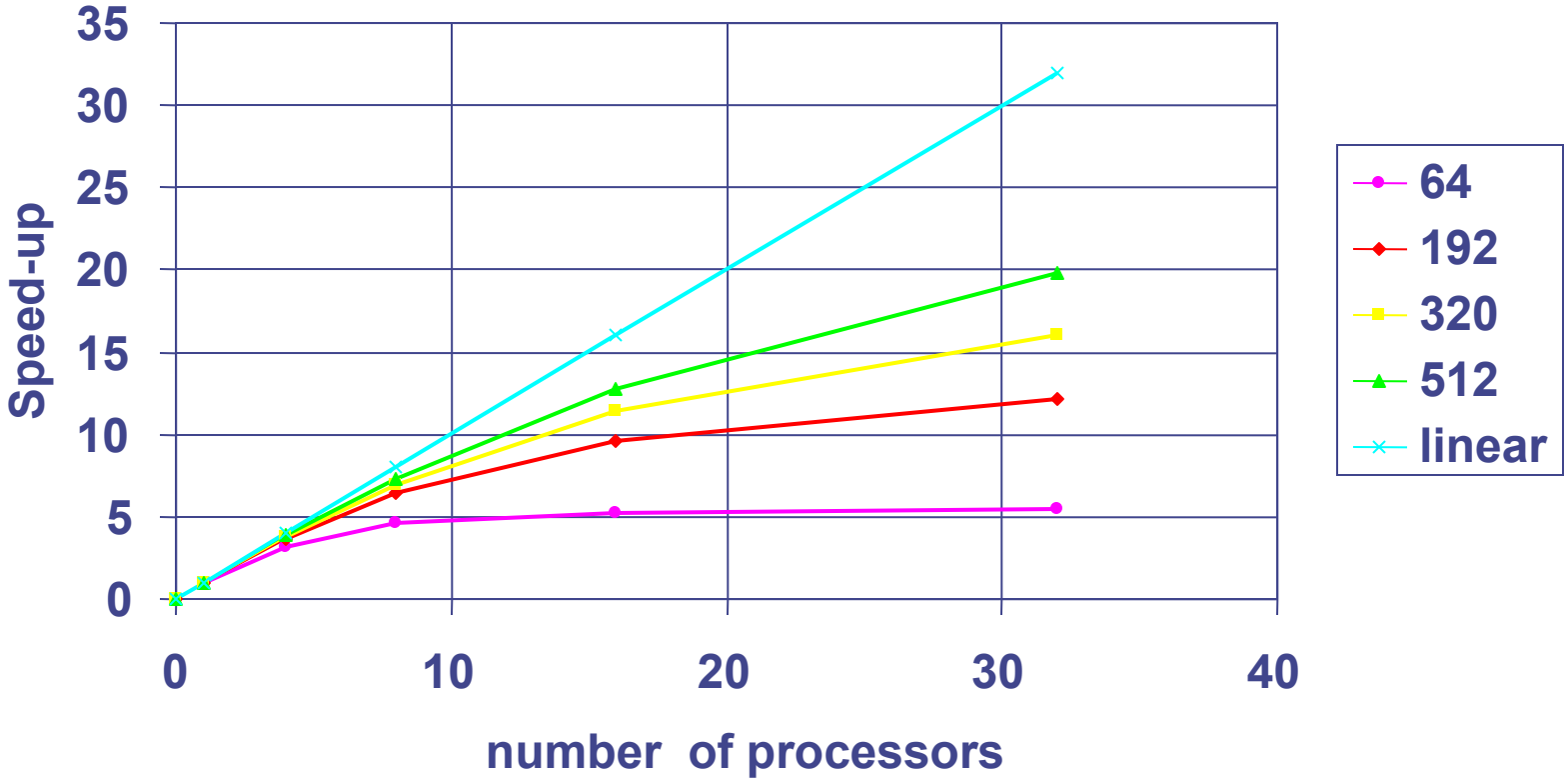
- The efficiency depends on the ratio of parallel overhead to the sequential execution time.
- It is decreasing for an increasing number of processors and increasing with the problem size.

Scalability

- Scalability means the performance capability for a growing number processors
- Intuitively, we can call a system scalable, if its performance increases with a simultaneous increase of problem size and number of processors.
- Speed-up and scale-up try to measure the scalability of a specific parallel program on a specific parallel machine.
- The discussion of the scale-up has shown that we can increase the efficiency of the system by increasing the problem size.
- The speed-up curve characterizes the scalability of a program at a particular input.
- Usually the speed-up curve flattens out, but the limit grows with the problem size.

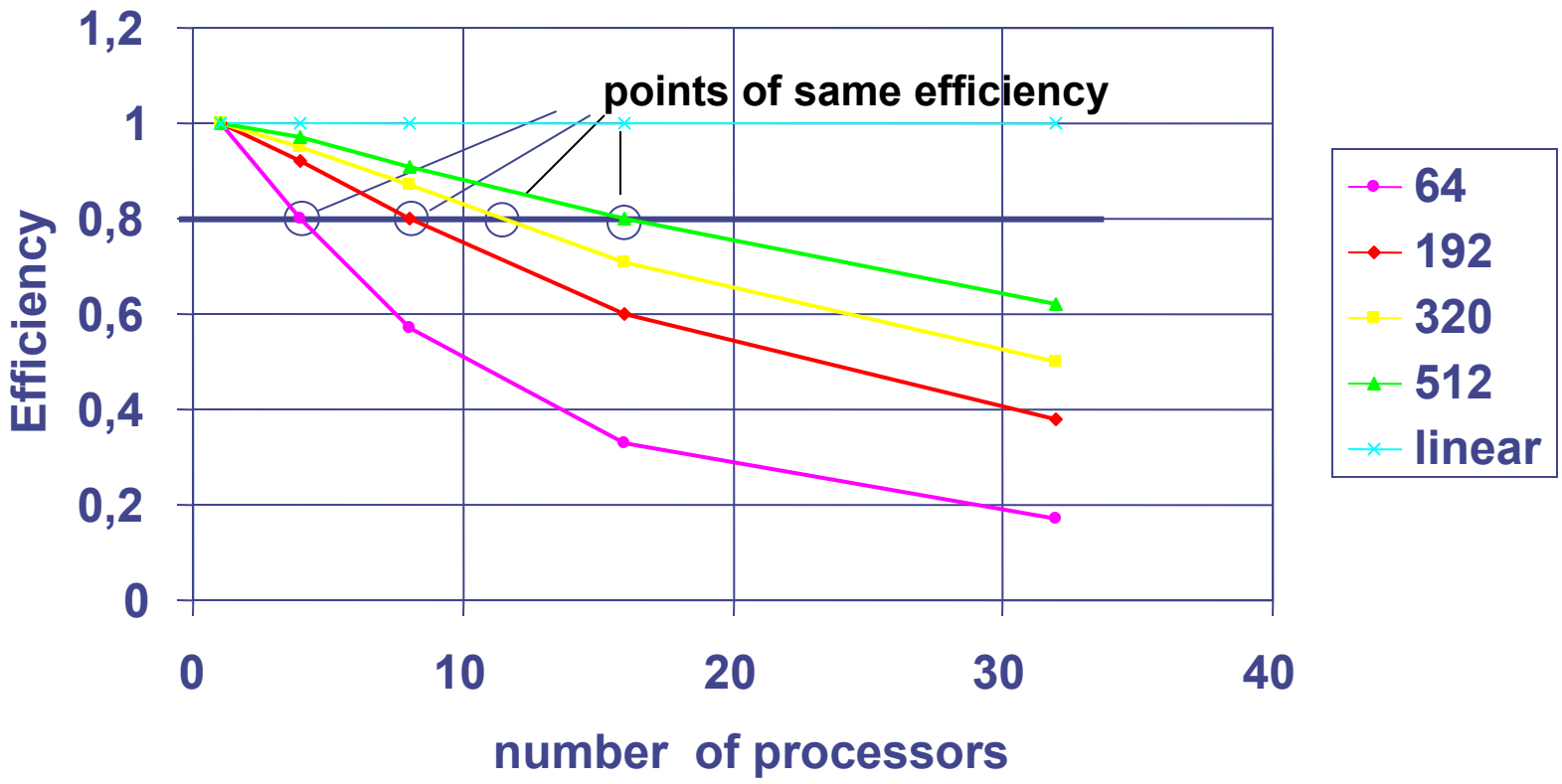
Example

Parallel summation of n=64, 192, 320, 512 numbers in a hypercube with p=1, 4, 8, 16, 32 processors



Example

How much bigger must the problem become that a specific efficiency (e.g. 0.8) is maintained?



Isoefficiency

- Plugging $T(1) = w\,t_c$ into equation from slide 3-18 yields

$$E(p) = \frac{1}{1 + \frac{T_o(p)}{T(1)}} = \frac{1}{1 + \frac{T_o(p)}{w\,t_c}}$$

- Solving for w results in

$$w = \frac{1}{t_c}\left(\frac{E(p)}{1 - E(p)}\right) T_o(p)$$

- *Defining* $K_E = E / (t_c\,(1 - E))$ as a constant depending on a specific efficiency level, we obtain:

$$w = K_E\, T_o(p)$$

- This establishes a functional relation between problem size w and number of processors p.
- The function is called **Isoefficiency** and indicates how much the problem size has to grow in order to maintain a given efficiency level for an increasing number of processors.

Properties of Isoefficiency

- If e.g. $w(p) = \Theta(p \log p)$ holds and we want to increase the number of processors from p to p', we have to increase the problem size w by a factor $(p' \log p') / (p \log p)$ to achieve the same efficiency.
- The smaller the asymptotic complexity of the isoefficiency function, the better the scalability, e.g.:
  - linear growth — good scalability
  - exponential growth: — very poor scalability
- Examples

Algorithm	Isoefficiency	Architecture
Factorization of scarce matrices	$\Theta(p \log^2 p)$	Clique
All-pairs-shortest Path (Dijkstra)	$O(p \log p)^{3/2}$	Hypercube
All-pairs-shortest Path (Dijkstra)	$O(p^{9/5})$	Grid

Limitation of parallelism degree

- Programs can usually not arbitrarily be parallelized.
- For each program there exists a maximum degree of parallelism

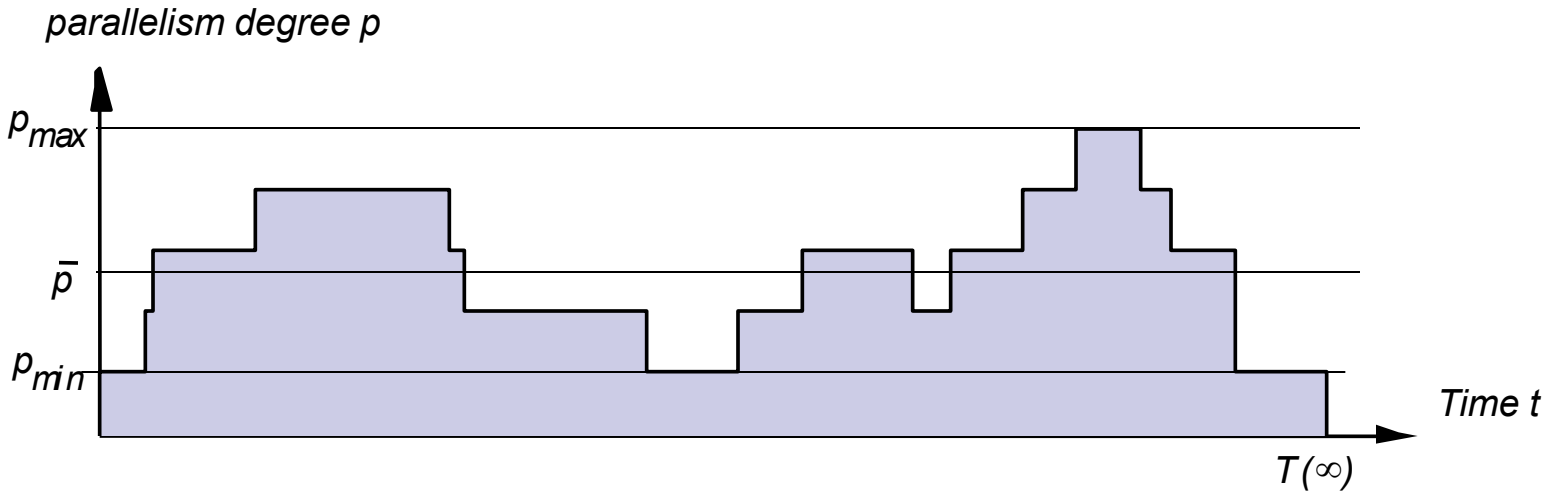
$$T(p) = \begin{cases} T_s + \frac{T_p}{p} & \text{für } p < p_{max} \\ T_s + \frac{T_p}{p_{max}} & \text{für } p \geq p_{max} \end{cases}$$

- In this case we get

$$T(\infty) = \lim_{p \to \infty} (T(p)) = T_s + \frac{T_p}{p_{max}}$$

Parallelism Profile of a Parallel Program

- Plotting the parallelism degree over the time we obtain the parallelism profile:



- *p(t)* can be interpreted as the number of processors that are active during the execution of the program – under the assumption that arbitrarily many processors are available.

Parallelism profile

- We get $\int_0^{T(\infty)} p(t)\,dt = T(1)$

  i.e. the area under *p(t)* indicates the amount of computation needed and corresponds to the execution time in the monoprocessor case.
- The average *degree of parallelism can be defined as*

$$\bar{p} = \frac{1}{T(\infty)} \int_0^{T(\infty)} p(t)\,dt$$

- Thus we obtain:

$$\bar{p} = \frac{T(1)}{T(\infty)} = S(\infty) = \lim_{p \to \infty} S(p)$$

  i.e. the asymptotic speed-up is equal to the average parallelism degree.
- Since *S(p)* is increasing monotonically, we can also write using Amdahl's law:

$$S(p) \leq \min \left\{ \bar{p}, \frac{1}{f + (1-f)/p} \right\}$$

Overhead due to thread interaction

- Threads as parts of a parallel program need to interact.
- The temporal overhead $T_C(p)$ (index „C“ for „*communication*“) for such interactions is a strictly monotonically increasing function of p.
- Using this, we obtain a new, extended approach for the overall execution time:

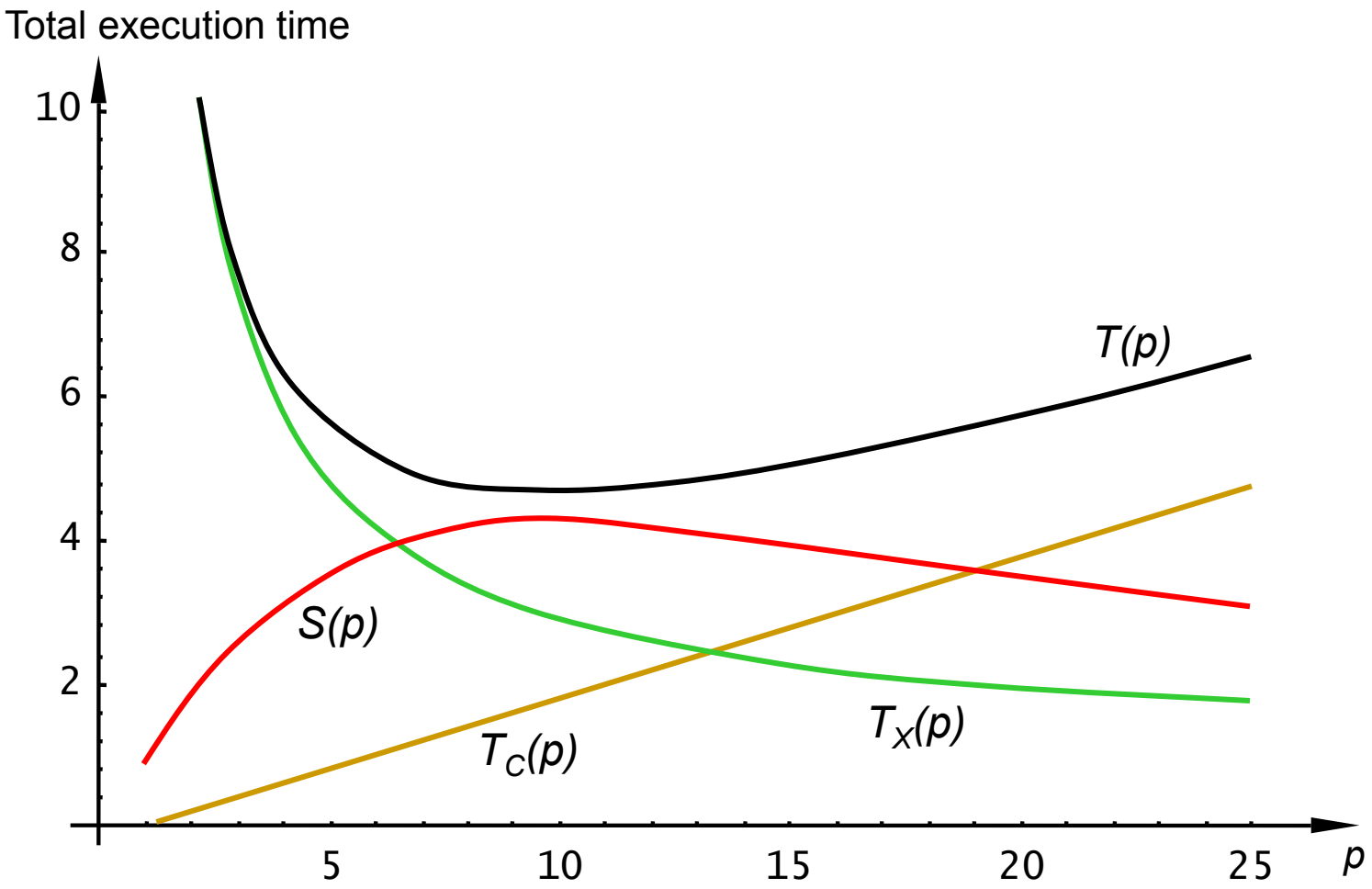
$$T(p) = T_X(p) + T_C(p)$$

with $T_C(1) = 0$ and $T_x(p) = \begin{cases} T_s + T_p / p & \text{for } p < p_{max} \\ T_s + T_p / p_{max} & \text{for } p \geq p_{max} \end{cases}$

as pure execution time of the program (index „x“ for „execution“).

- T_x is a monotonically decreasing function which is bounded below.
- Is T_C strictly monotonically increasing and not bounded above, then $T(p)$ necessarily increases beyond some p.

Effect of process interaction



Total execution time $T(p)$ is composed of pure execution time $T_X(p)$ and interaction overhead $T_C(p)$.

3.3 Impact of Communication Overhead

- Let a parallel program consist of *m* threads, each of which needing *R* time units of computation time.
- The *m* threads are distributed onto *p* processors :

$$\sum_{i=1}^{p} m_i = m$$

where m_i denotes the number of threads assigned to processor *i*.

- Each thread communicates with each other thread taking *C* time units in total.
- If the two communicating threads are assigned to the same processor the communication costs can be neglected (C=0).
- Assuming random distribution of the *m* threads, we obtain a total computation time $T_x(p)$ of

$$T_x(p) = R \max_i(m_i)$$

and a total communication time of

$$T_C(p) = \frac{1}{2}\sum_{i=1}^{p} m_i(m - m_i)\, C = \frac{C}{2}\left(m^2 - \sum_{i=1}^{p} m_i^2\right)$$

Communication Overhead

- Both parts together yield a total execution time

$$T(p) = T_x(p) + T_c(p) = R \max_i(m_i) + \frac{C}{2}\left(m^2 - \sum_{i=1}^{p} m_i^2 \right)$$

- If we distribute the threads evenly across the processors ($m_i = m/p$), the expression simplifies to

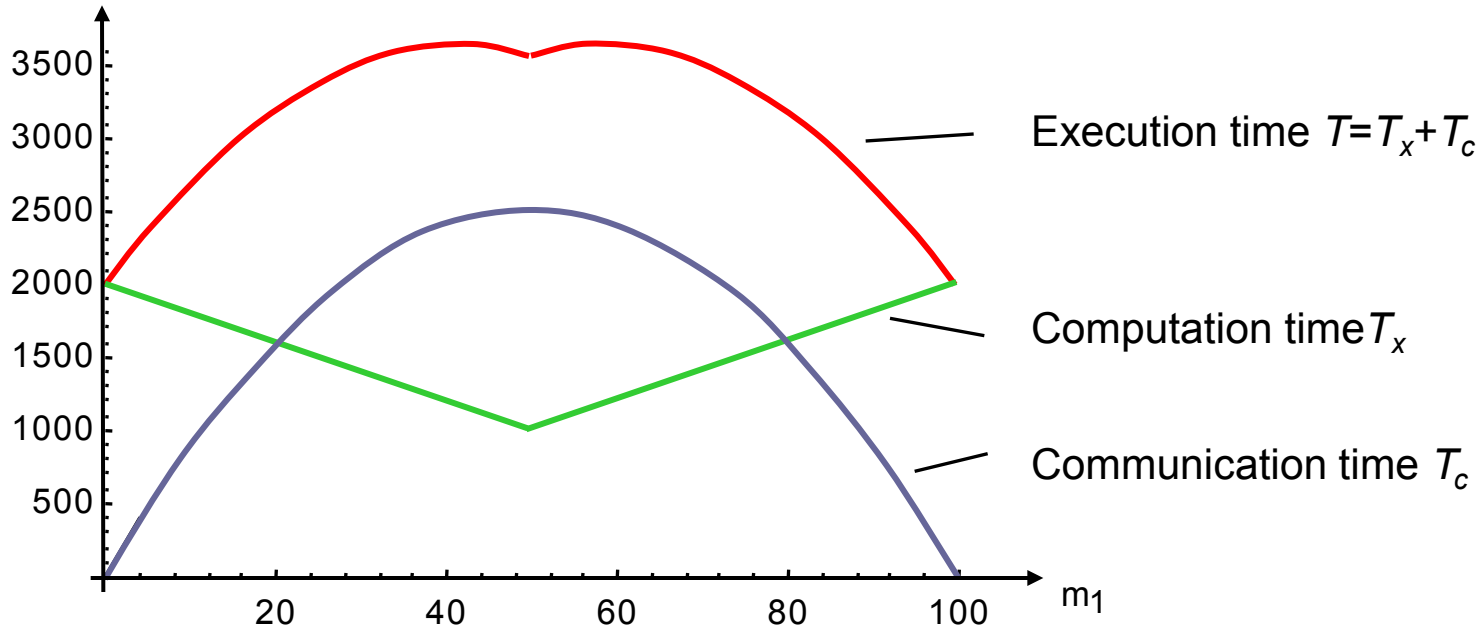
$$T(p) = R \frac{m}{p} + \frac{C}{2} m^2 \left(1 - \frac{1}{p}\right)$$

- The question now is: Does parallel computing pay off, i.e. is $T(p) < T(1)$ valid?

$$\begin{aligned} T(p) &< T(1) \\ R\frac{m}{p} + \frac{C}{2} m^2 \left(1 - \frac{1}{p}\right) &< R\,m \\ \frac{C}{2} m^2 \left(1 - \frac{1}{p}\right) &< R\,m\left(1 - \frac{1}{p}\right) \\ \frac{m}{2} &< \frac{R}{C} \end{aligned}$$

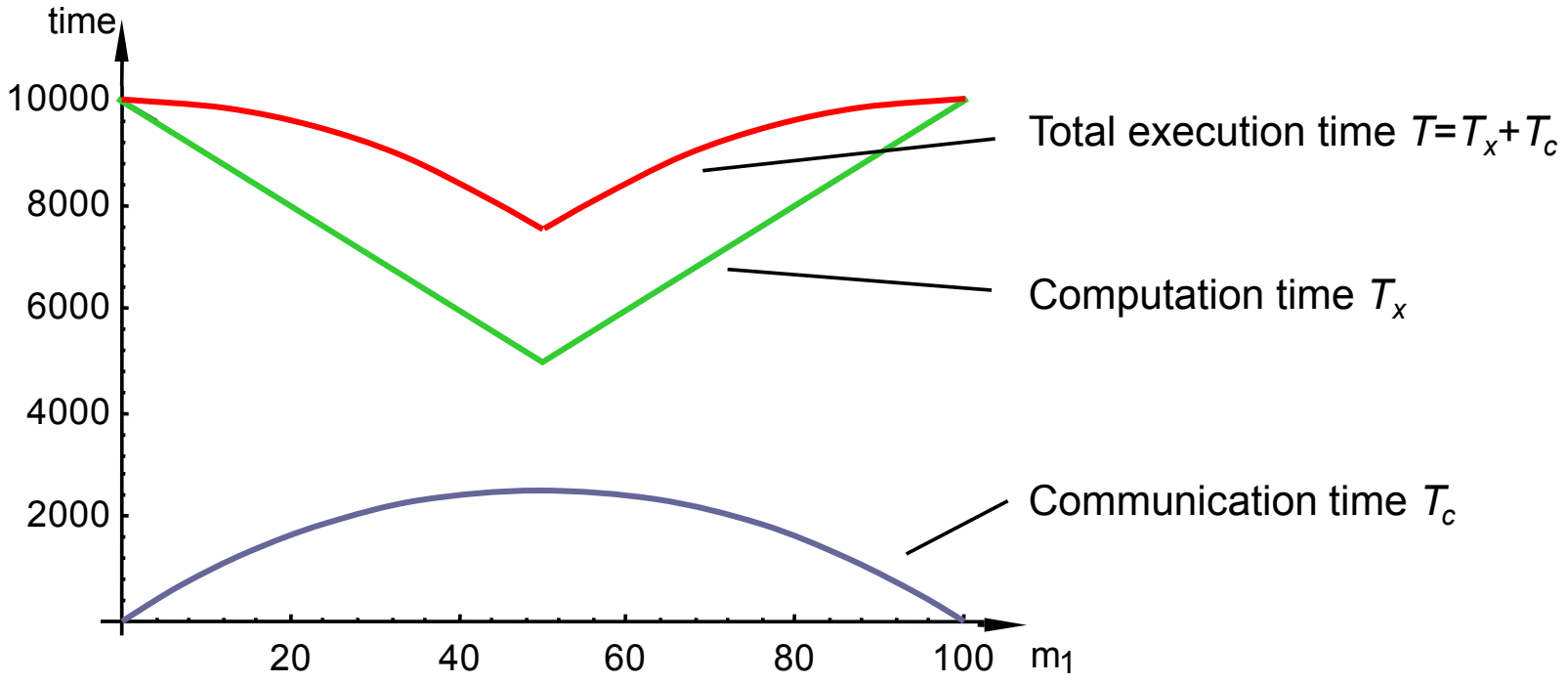
Communication Overhead

- A distribution and thus real parallelism only pays off if the ratio of computation time *R* to communication time *C* is greater than *m*/2.
- The ration *R*/*C* represents a critical quantity and indicates the *granularity* of parallel processing.
- Example (*p* =2 processors, *m* =100 threads, C= 1 (communication time))



Execution time as a function of the distribution for R=20 (i.e. R/C = 20)
Distribution does not pay off!

Communication Overhead



- Execution time as a function of the distribution for R=100 (i.e. R/C = 100)
  Distribution does pay off! (each processor takes 50 threads).

3.4 Multiprogramming in Parallel Computing

Given a set of parallel programs. Parallelism can be

(a) external, i.e. between the programs (*Inter*program parallelism)

and/or

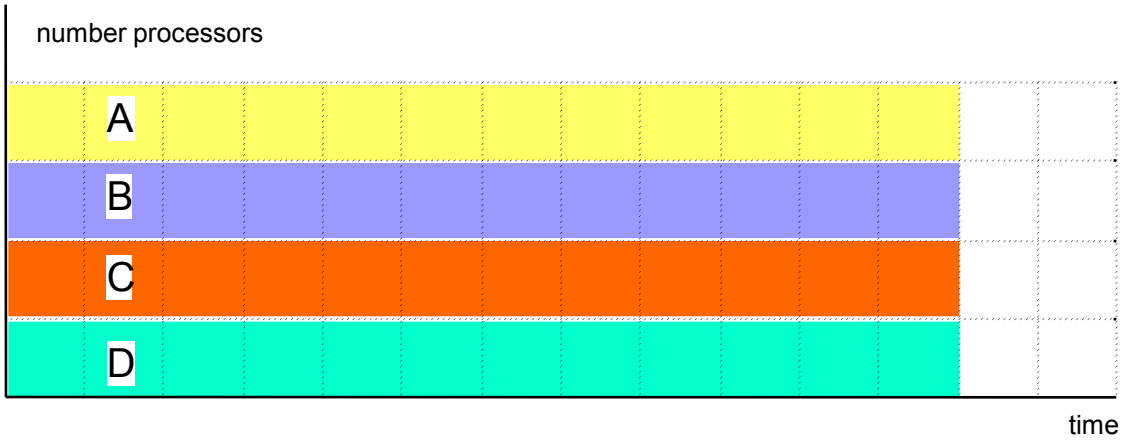
(b) internal, i.e. within a program (*Intra*program parallelism)

Therefore we get four combinations, with the ESIS variant irrelevant for parallel machines due to lacking parallelism

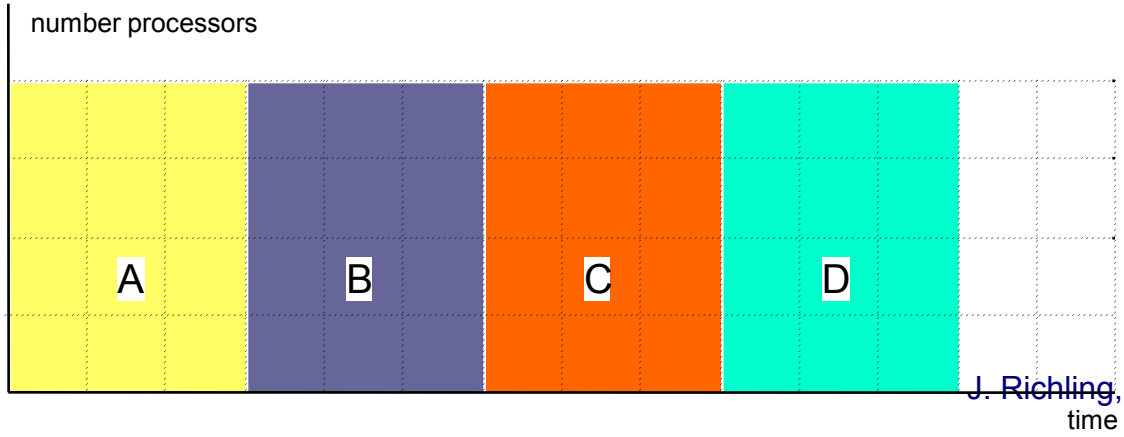
	internal sequential (Process level)	internal parallel (Process level)
external sequential (Program level)	ESIS	ESIP
extern parallel (Program level)	EPIS	EPIP

Multiprogramming

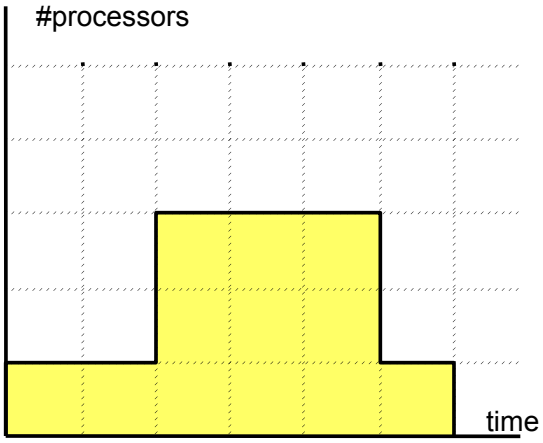
- EPIS: Parallel execution of sequential programs (interprogram parallelism only)



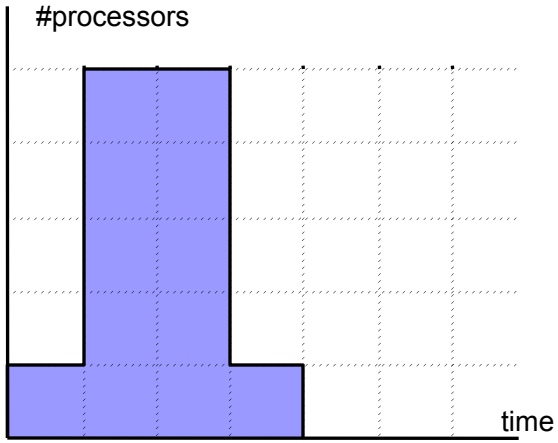
- ESIP: Sequential execution of parallel programs (Intraprogram parallelism only)



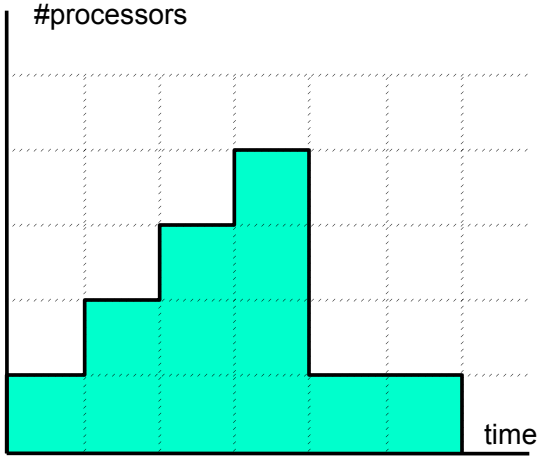
External and Internal Parallelism



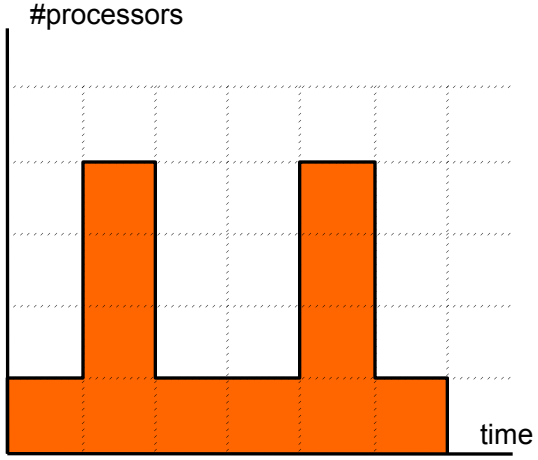
Program A



Program B

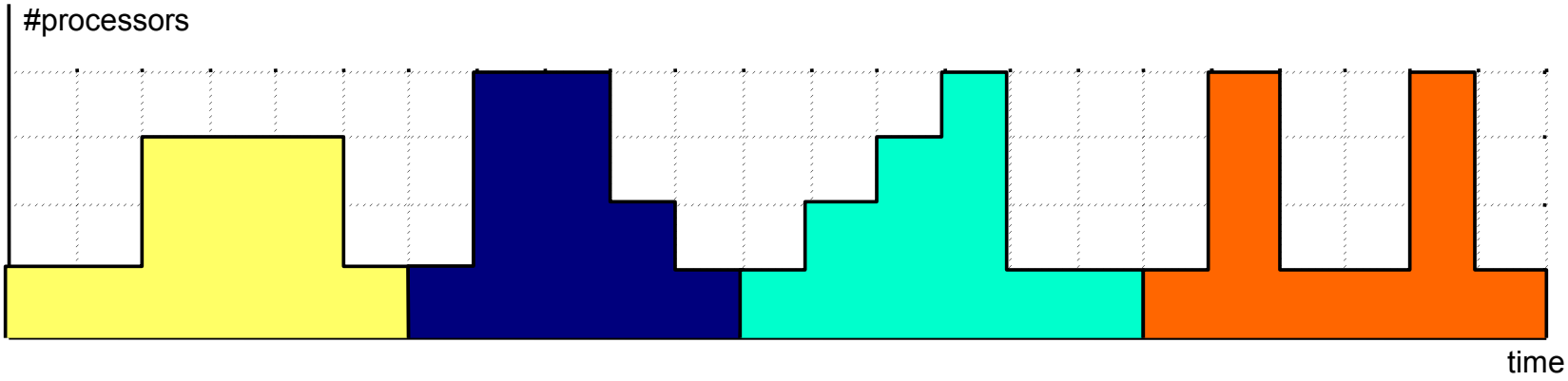


Program C

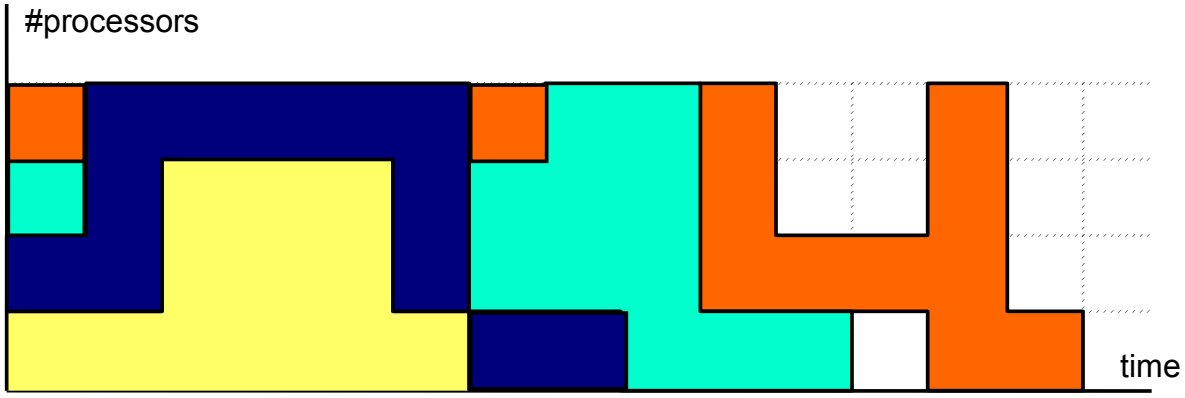


Program D

External and Internal Parallelism



ESIP: Prolongation of total execution time due to insufficient parallelism in the programs



EPIP: External and internal parallelism: possible separation in time and space

Further References

- Grama, A.; Gupta, A.; Karypis, G.: Introduction to Parallel Computing, Addison Wesley, 2003
- Quinn, M.J.: Parallel Programming in C with MPI and OpenMP. McGraw-Hill Education, 2003
- Chandra, R. et al.: Parallel Programming in OpenMP. Morgan Kaufmann Publishers, 2000

Parallel Addition