Parallelism (PAR)

Analysis of parallel applications

Eduard Ayguade, Josep Ramon Herrero and Daniel Jiménez
({eduard,josepr,djimenez}@ac.upc.edu)

Computer Architecture Department
Universitat Politècnica de Catalunya, UPC–BarcelonaTECH

2014/15-Fall

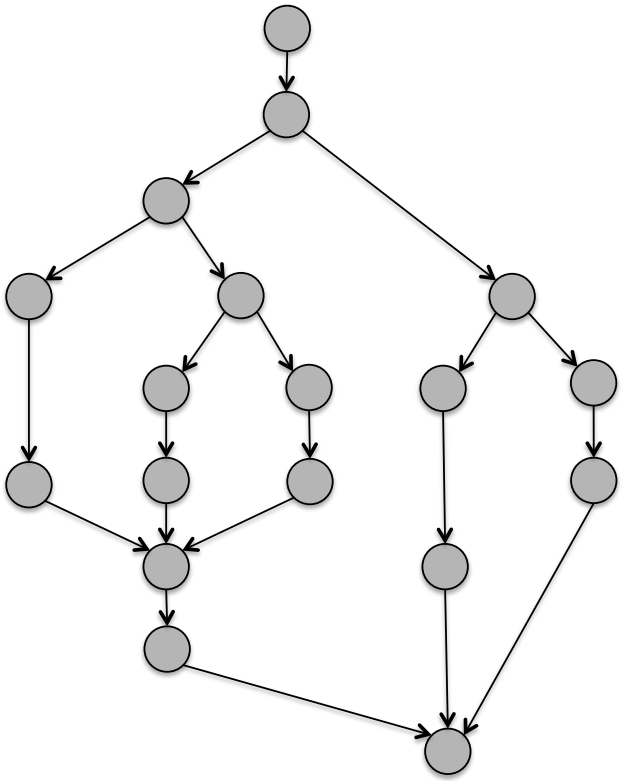
Outline

Finding concurrency/parallelism

- Can the computation be divided in parts?[1]
  - Based on the processing to do:
    - Task decomposition (e.g. functions, loop iterations)
  - Based on the data to be processed:
    - Data decomposition (e.g. matrix) (implies task decomposition)
- There may be (data or control) dependencies between tasks
- The decomposition determines the potential paralellism that could be obtained

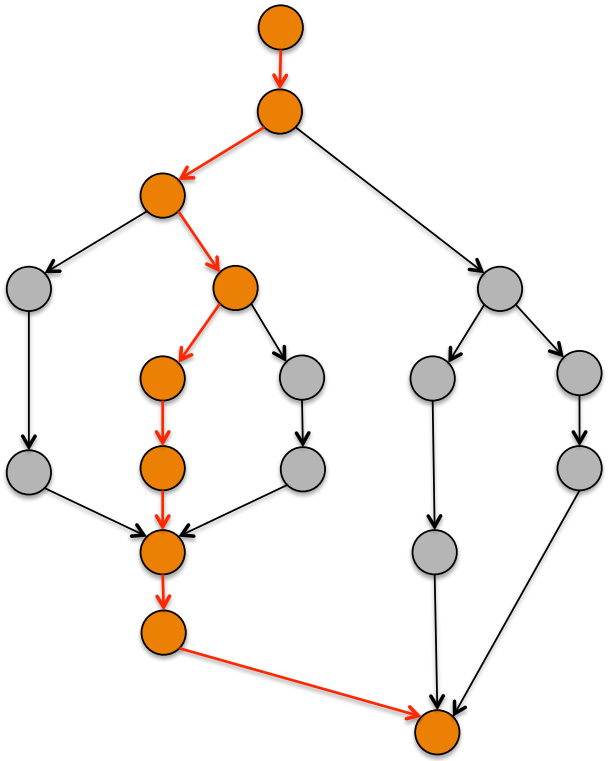
[1] Different strategies covered later in this course

Computing the parallelism



- Computation task graph abstraction
  - Directed Acyclic Graph
  - Node = arbitrary sequential computation: task
  - Edge = successor node can only execute after predecessor node has completed: dependence
- Processor abstraction (simplification)
  - P identical processors
  - Each processor executes one node at a time

Computing the parallelism



- $T_1 = \sum_{i=1}^{nodes}(work\_node_i)$
- $T_\infty = \sum_{i \in criticalpath}(work\_node_i)$, assuming sufficient resources
- $Parallelism = T_1/T_\infty$, independent of number of processors P
- P_{min} is the minimum number of processors necessary to achieve $Parallelism$

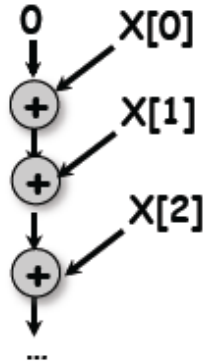
Example: vector sum

Compute the sum of elements `X[0] ... X[n-1]` of a vector `X`

Sequential algorithm.

```
sum = 0; for ( i=0 ; i< n ; i++ ) sum += X[i];
```

- Computation graph



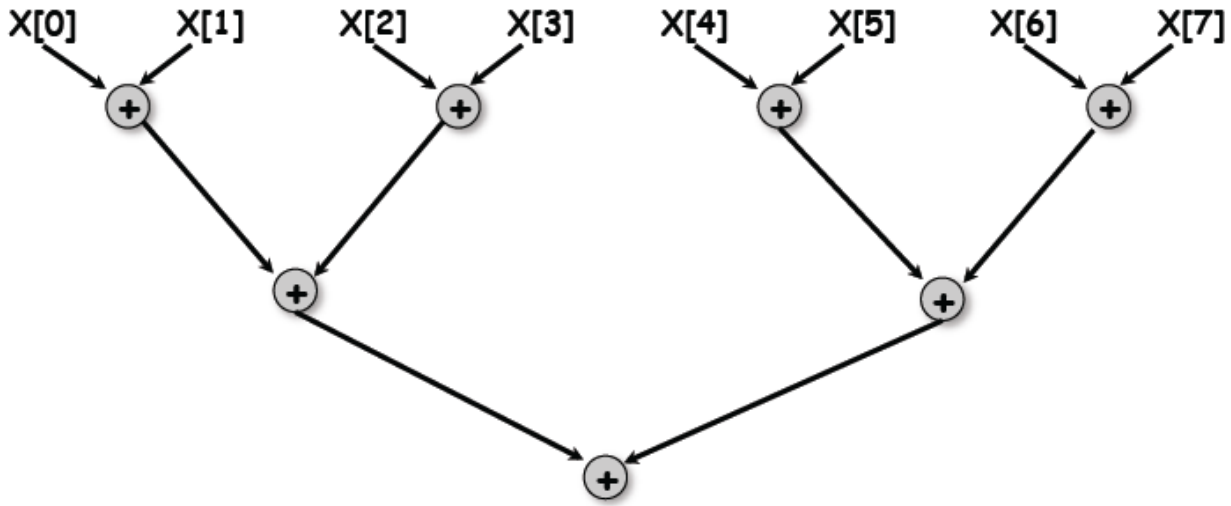
$$T_1 = O(n)$$
$$T_\infty = O(n)$$
$$Parallelism = O(1)$$

How can we design an algorithm (computation graph) with more parallelism?

Example: vector sum

Compute the sum of elements `X[0]` ... `X[n-1]` of a vector `X`

- An alternative computation graph



- $T_1 = O(n)$; $T_\infty = O(log\ n)$; $Parallelism = O(n/(log\ n))$
- How to restructure the sequential algorithm to have this computation graph? (iterative vs. recursive solutions)

Example: vector sum

Compute the sum of elements `X[0]` ... `X[n-1]` of a vector `X`

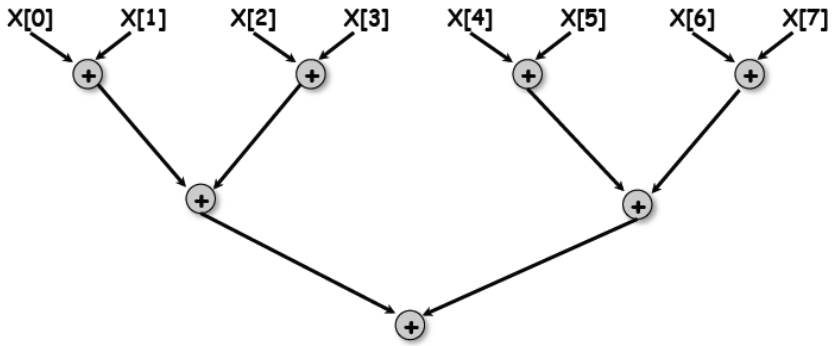
Recursive algorithm:

```
int recursive_sum(int *X, int n)
{
   int ndiv2 = n/2;
   int sum=0;

   if (n==1) return X[0];

   sum = recursive_sum(X, ndiv2);
   sum += recursive_sum(X+ndiv2, n-ndiv2);
   return sum;
}

void main()
{
   int sum, X[N];
   ...
   sum = recursive_sum(X,N);
   ...
}
```



Example: database query processing

Consider the following database with 10 records:

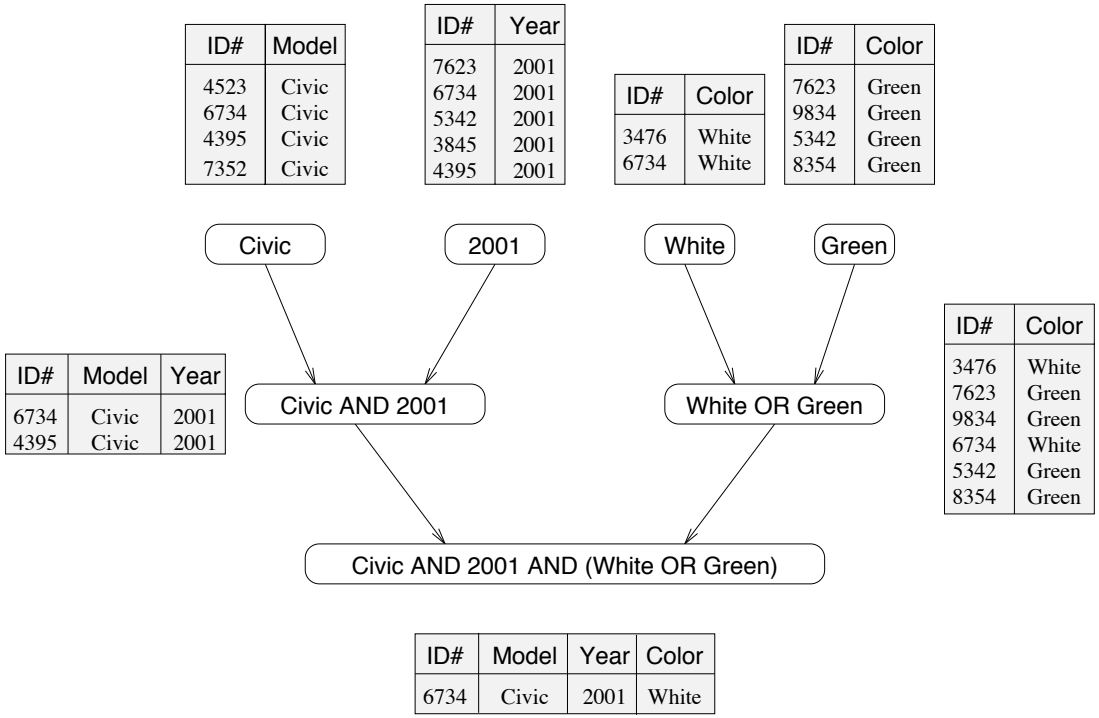
ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

and execution of the query:

```
MODEL = ‘‘CIVIC’’ AND YEAR = 2001 AND
(COLOR = ‘‘GREEN’’ OR COLOR = ‘‘WHITE)
```

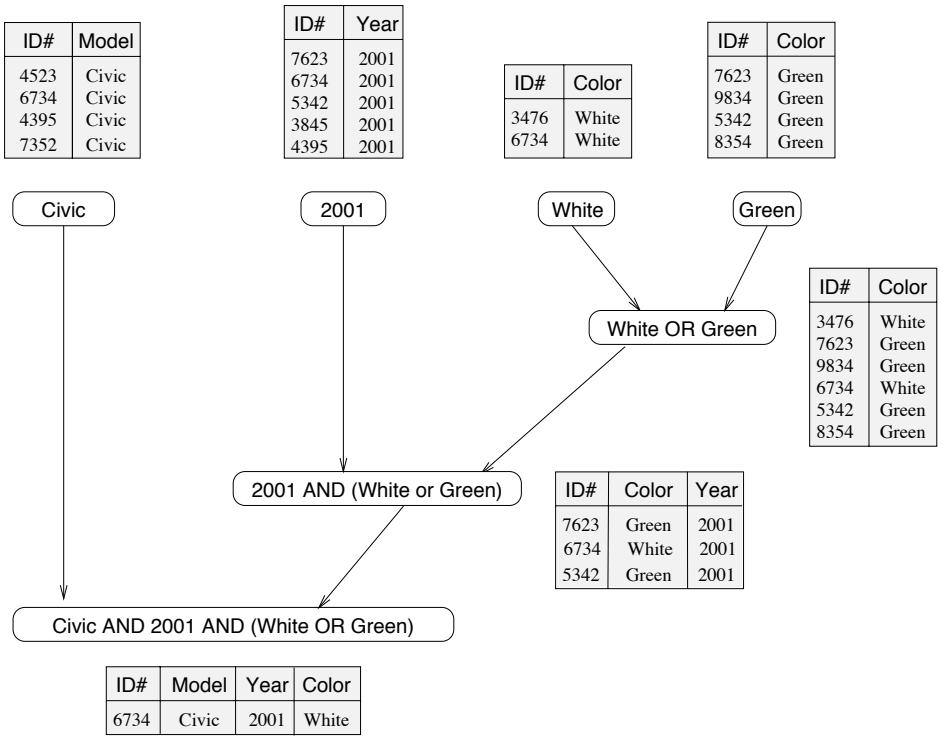
Example: database query processing

The execution of the query can be divided into tasks in various ways. Each task can be thought of as generating an intermediate table of entries that satisfy a particular clause.



Example: database query processing

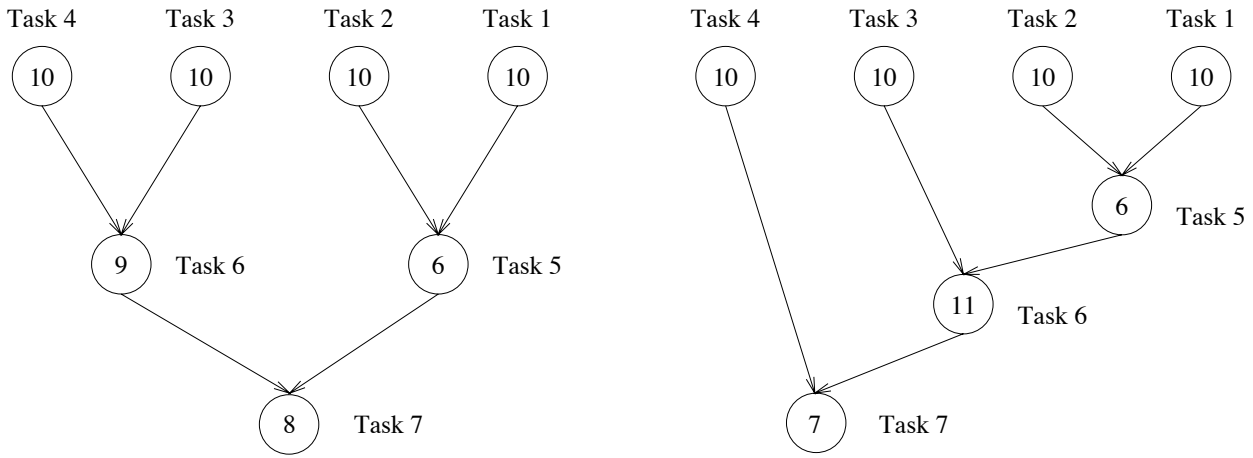
Note that the same problem can be decomposed into tasks in other ways as well.



Different task decompositions may lead to significant differences with respect to their eventual parallel performance.

Example: database query processing

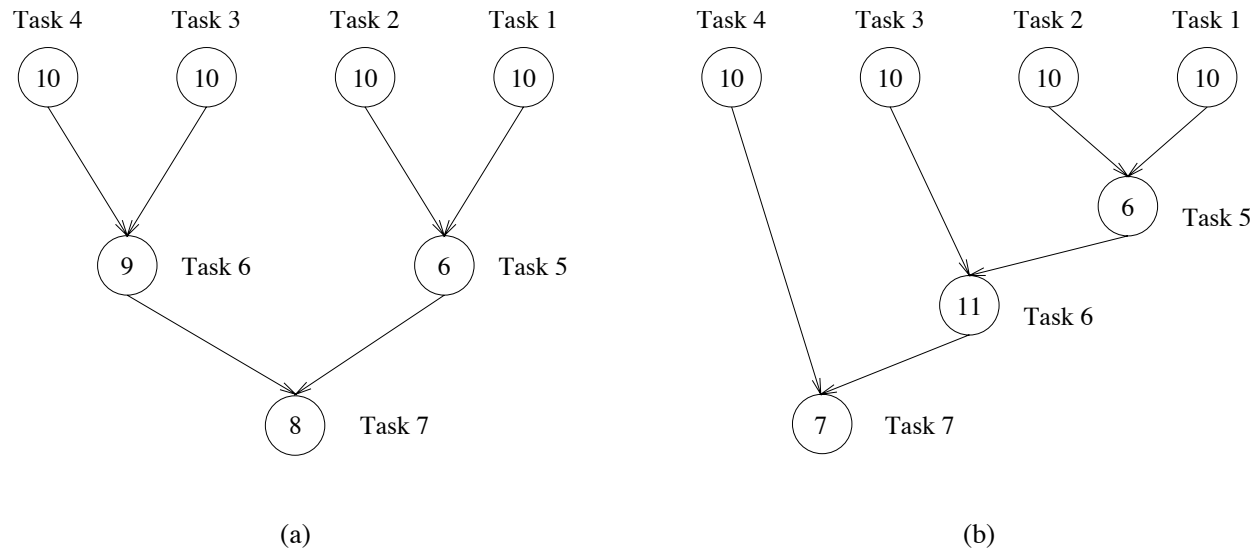
Consider the task dependency graphs of the two database query decompositions[2]:



Question: Which are the T_1, T_∞ and $Parallelism$ in each case?

[2] *work_node* is equal to the number of inputs to be processed by the node.

Example: database query processing



$$T_1^{(a)} = 63, T_\infty^{(a)} = 27, Parallelism^{(a)} = 63/27 = 2.33$$

$$T_1^{(b)} = 64, T_\infty^{(b)} = 34, Parallelism^{(b)} = 64/34 = 1.88$$

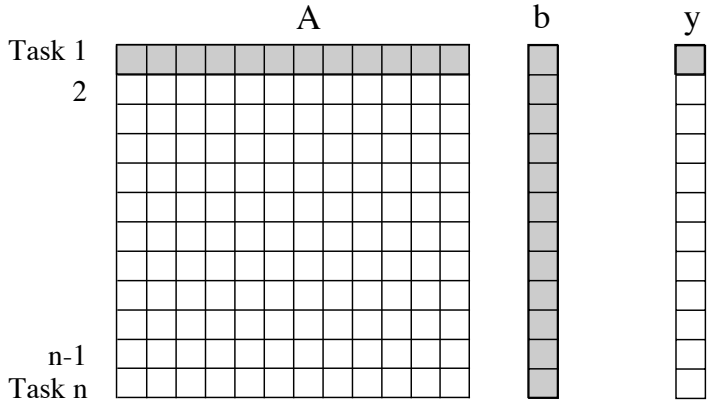
Question: How many processors P_{min} are needed in each case to achieve this $Parallelism$?

Granularity and parallelism

- Each node in the computation task graph represents a sequential computation
- The granularity of the decomposition is determined by the size of each node in the computation graph
- Fine-grained tasks vs. coarse-grained tasks: the degree of parallelism increases as the decomposition becomes finer in granularity and vice versa

Granularity and parallelism: fine-grained decomposition

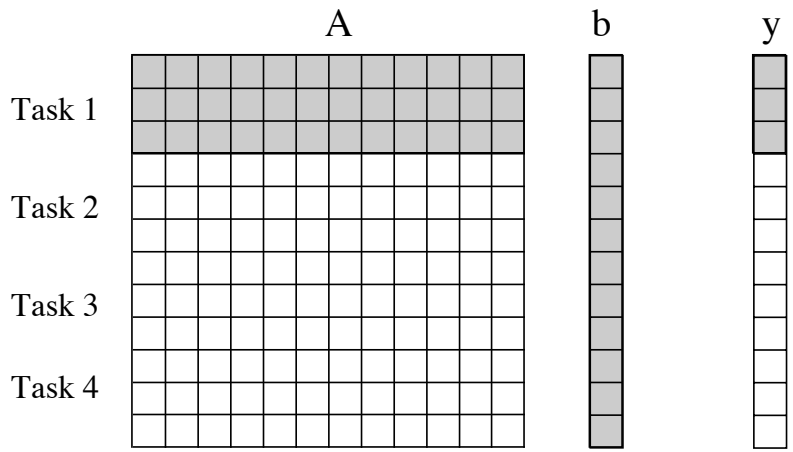
Example: matrix-vector product (n by n matrix):



- A task could be each individual $\times$ or $+$ in the dot product that computes an element of y ($y[i] = y[i] + A[i][j] * b[j]$)
- A task could also be each complete dot product to compute an element of y ($y[i] = \sum_{j=1}^{j=n} (A[i][j] * b[j])$)

Granularity and parallelism: coarse-grained decomposition

- A task could be in charge of computing a number of consecutive elements of y (e.g. three elements)



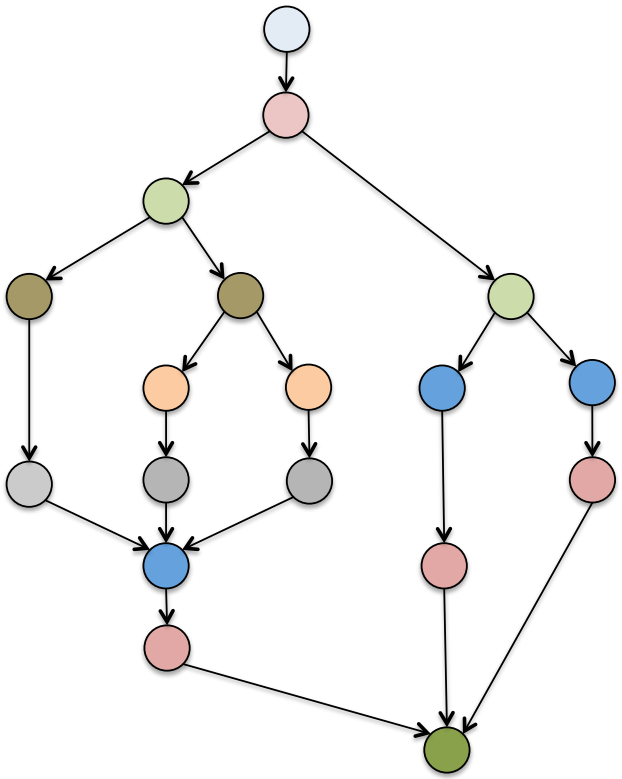
- A task could be in charge of computing the whole vector y

Granularity and parallelism: fine vs. coarse-grained

- It would appear that the parallel time can be made arbitrarily small by making the decomposition finer in granularity but...
  - Inherent bound on how fine the granularity of a computation can be
    - *e.g. matrix-vector multiply:* (n^2) *concurrent tasks.*
  - Tradeoff between the granularity of a decomposition and associated overheads (sources of overhead commented later, e.g. creation of tasks, synchronization, exchange of data between tasks, ...)
  - The granularity may determine performance bounds

Outline

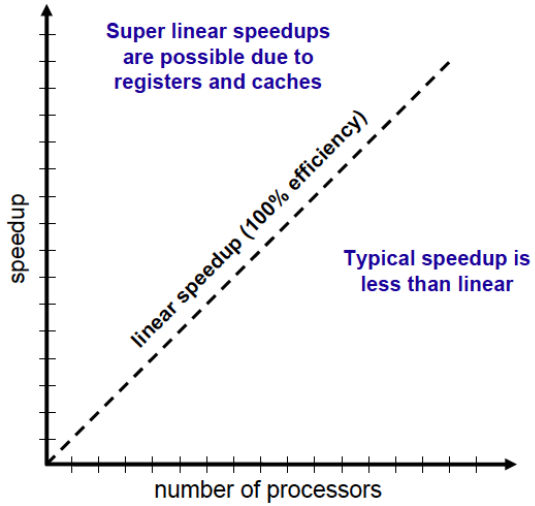
Speedup



- T_p = execution time on P processors (depends on the schedule of the graph nodes on the processors)
- Lower bounds
  - $T_p \geq T_1/P$
  - $T_p \geq T_\infty$
- $Speedup$ on P processors: $S_p = T_1/T_p$

Speedup vs. efficiency

- Speedup S_p: relative reduction of execution time when using P processors with respect to sequential

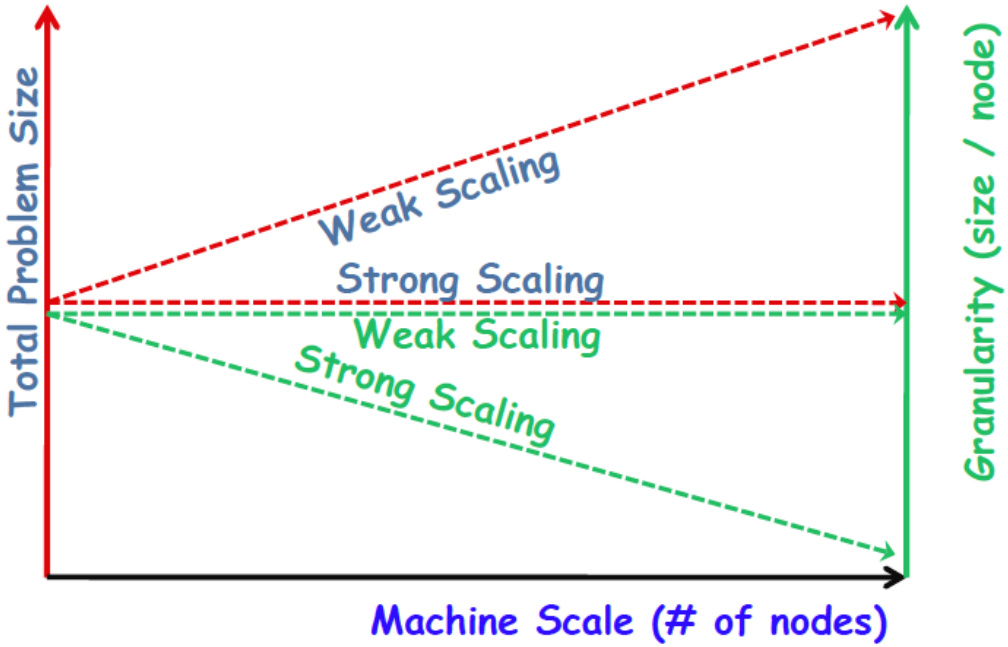


- Efficiency Eff_p: it is a measure of the fraction of time for which a processing element is usefully employed
  - $Eff_p = T_1/(T_p \times P)$
  - Also, $Eff_p = S_p/P$

Strong vs. weak scaling

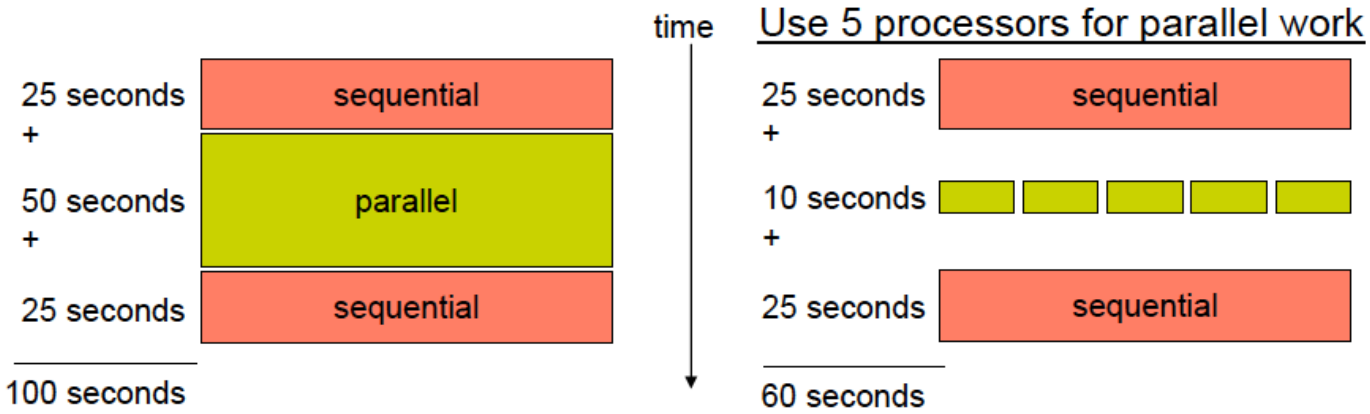
How does the efficiency behave when ...

- the problem size is constant (strong scaling)
- the task granularity is constant (weak scaling)



Amdahl's law

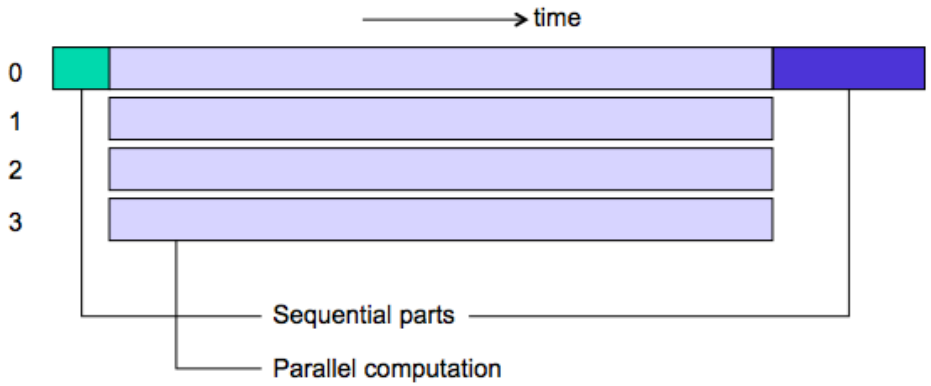
The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used



- Parallel part is 5 times faster: $Speedup_{parallel\_part} = 50/10 = 5$
- Parallel version is just 1.67 times faster: $S_p = 100/60 = 1.67$

Amdahl's law

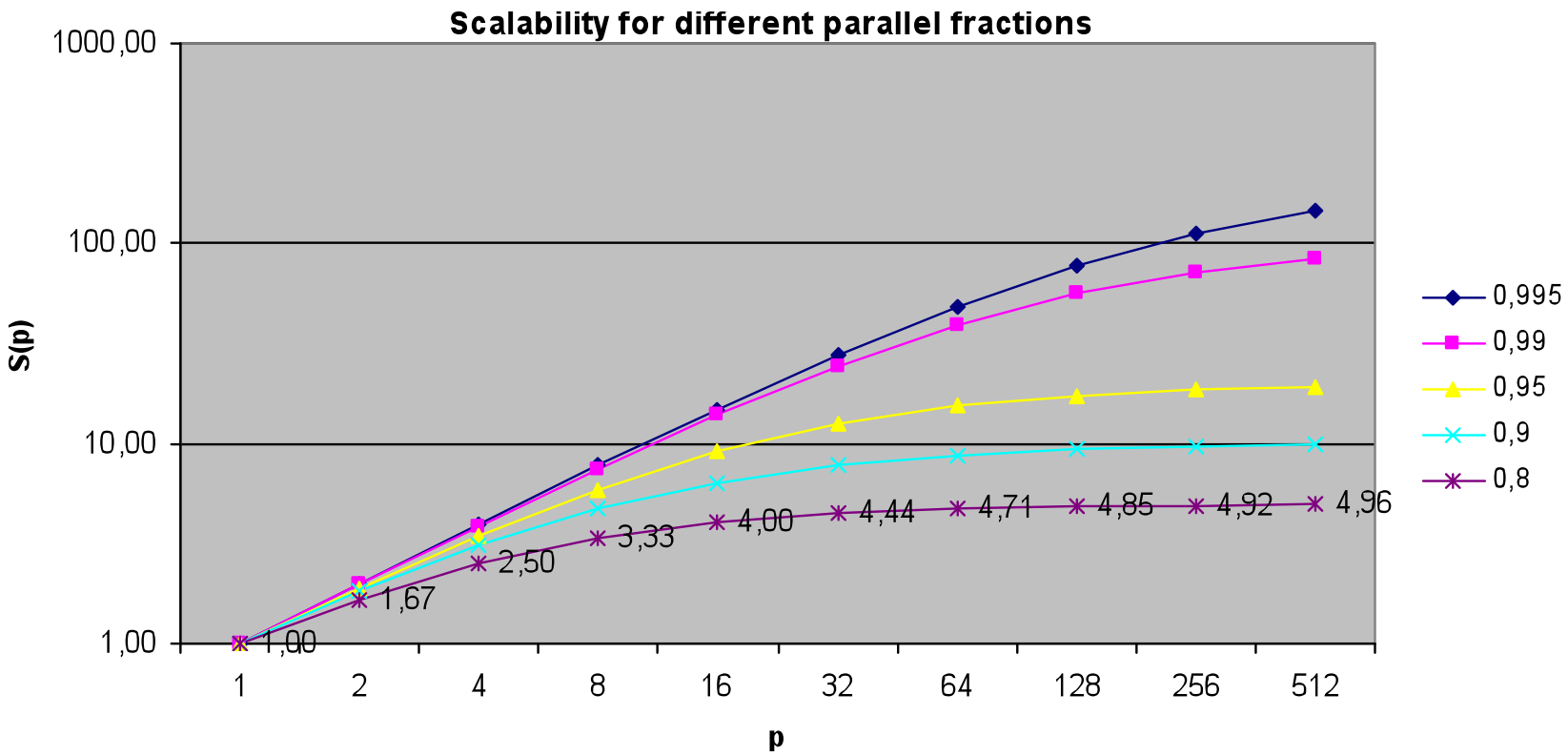
- Performance improvement is limited by the fraction of time the program runs in parallel (e.g. the parallel fraction φ)



$$T_1 = T_{seq} + T_{par} = (1-\varphi) \times T_1 + \varphi \times T_1$$
$$T_p = (1-\varphi) \times T_1 + (\varphi \times T_1/P)$$
$$S_p = \frac{T_1}{T_p}$$

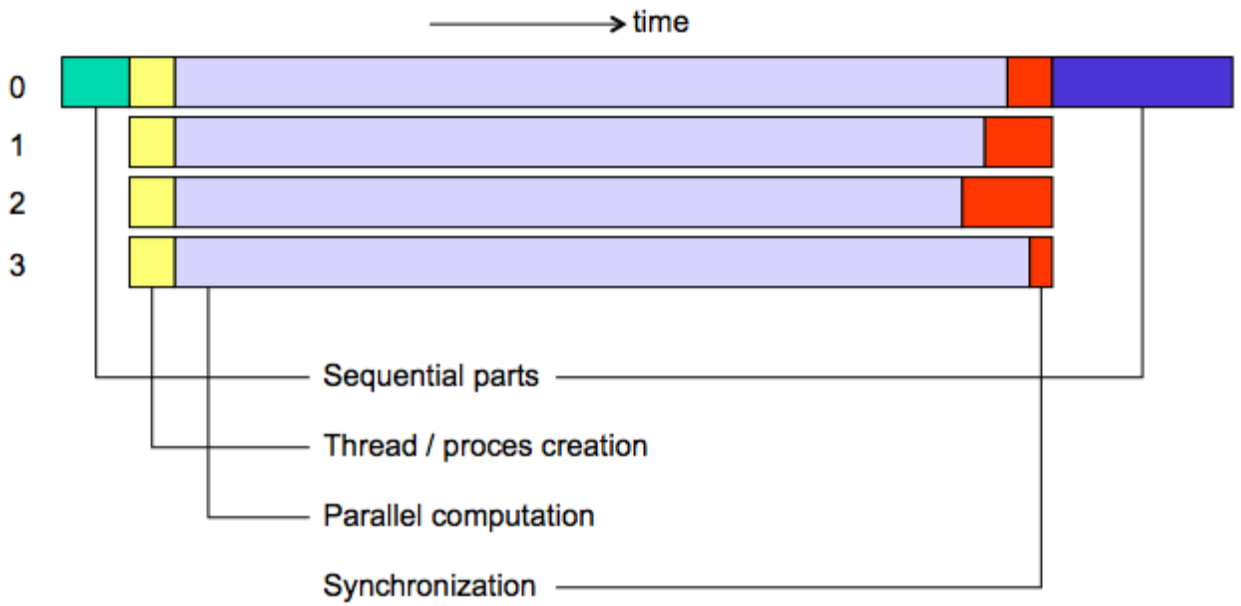
$$S_p = \frac{1}{((1-\varphi) + \varphi/P)}$$
$$S_p \rightarrow \frac{1}{(1-\varphi)} \; when \; P \rightarrow \infty$$

Amdahl's law


Scalability for different parallel fractions
1000,00
100,00
10,00
1,00
S(p)
1
2
4
8
16
32
64
128
256
512
p
1,00
1,67
2,50
3,33
4,00
4,44
4,71
4,85
4,92
4,96
0,995
0,99
0,95
0,9
0,8

Sources of overhead

Parallel computing is not free, we should account overheads (i.e. any cost that gets added to a sequential computation so as to enable it to run in parallel)



Sources of overhead

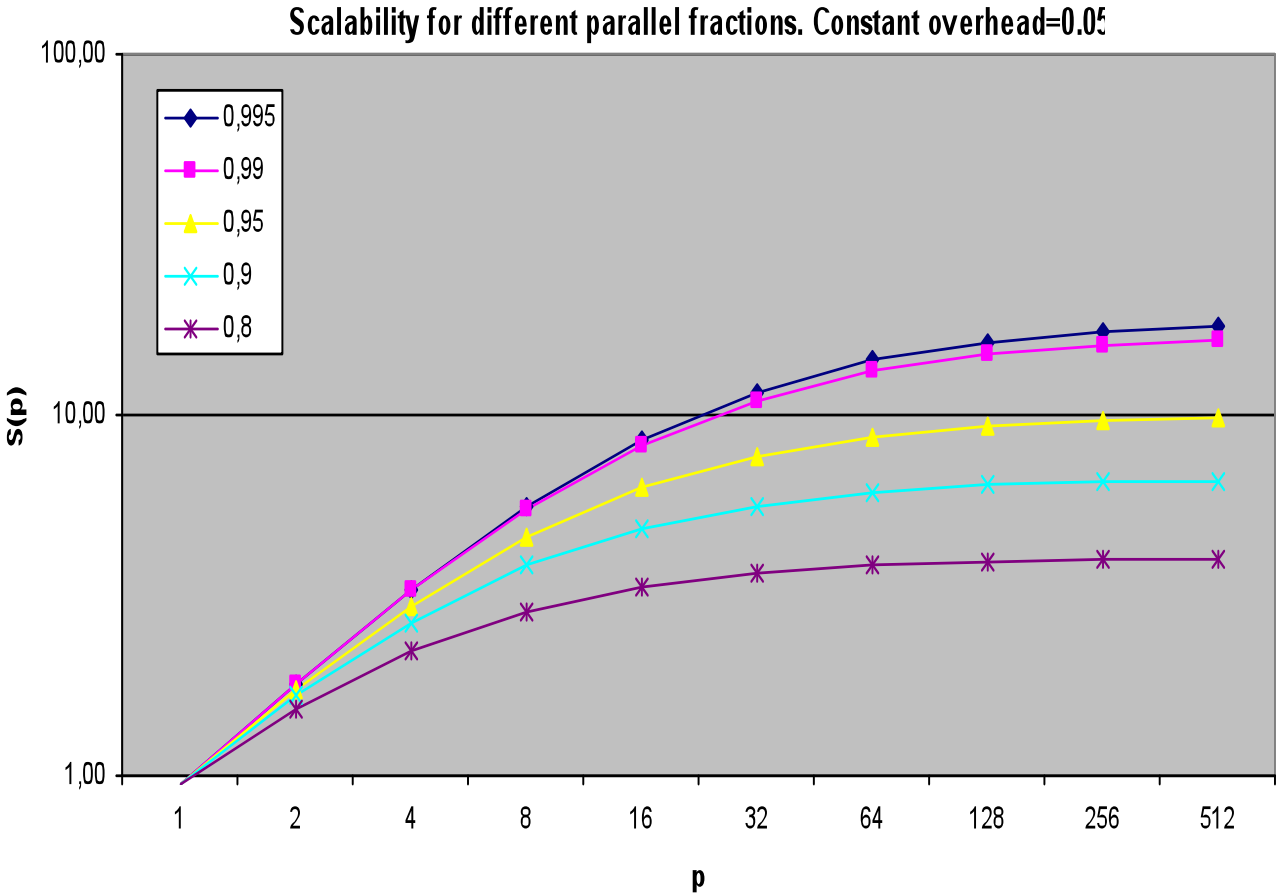
- **Task creation and termination:** extra processing performed at the start and end of each task
- **Synchronization:** extra processing to ensure that dependences in computation graph are satisfied
- **Data sharing:** can be explicit via messages, or implicit via a memory hierarchy (caches)

Sources of overhead (cont.)

- **Idleness:** thread cannot find any useful work to execute (e.g. dependences, load imbalance, poor communication and computation overlap or hiding of memory latencies, ...)
- **Computation:** extra work added to obtain a parallel algorithm (e.g. replication)
- **Memory:** extra memory used to obtain a parallel algorithm (e.g. impact on memory hierarchy, ...)
- **Contention:** competition for the access to shared resources (e.g. memory, network)

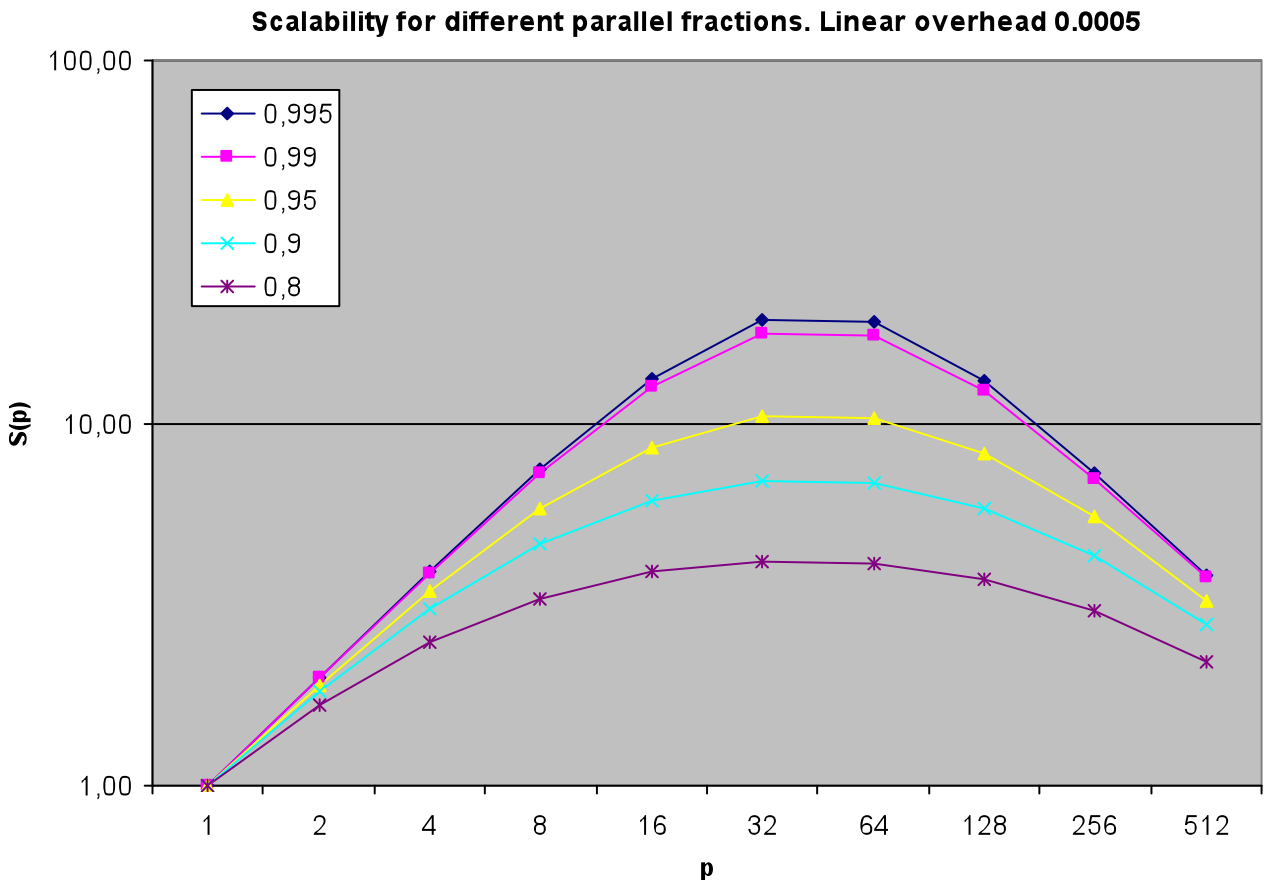
Amdahl's law (constant overhead)

$$T_p = (1 - \varphi) \times T_1 + \varphi \times T_1/p + overhead$$



Amdahl's law (linear overhead)

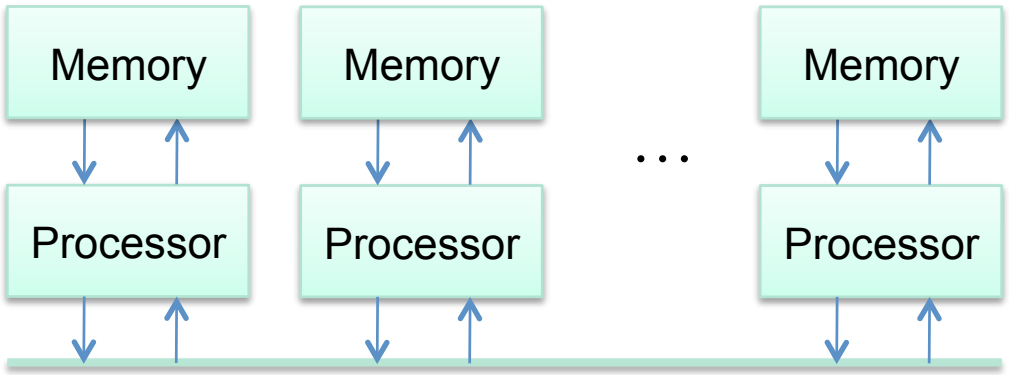
$$T_p = (1 - \varphi) \times T_1 + \varphi \times T_1/p + overhead(p)$$



Outline

How to model data sharing overhead?

We start with a simple architectural model in which each processor P_i has its own memory, interconnected with the other processors through an interconnection network.



- Processors access to local data (in its own memory) using regular load/store instructions
- We will assume that local accesses take zero overhead.

How to model data sharing overhead?

- Processors can access remote data (in other processors) using a message-passing model (remote load instruction[3])
- To model the time needed to access remote data we will use two components:
  - Start up: time spent (t_s) in preparing the remote access
  - Transfer: time spent in transferring the data (number of bytes m, time per byte t_w) from/to the remote location

$$T_{access} = t_s + m \times t_w$$

- Synchronization between the two processors involved may be necessary to guarantee that the data is available. We will assume synchronization costs are negligible

[3] Remote store is also possible, not used in our model.

How to model data sharing?

Assumptions (to make simpler the model)

- At a given moment, a processor P_i can only execute one remote memory access
- At a given moment, a processor P_i can only serve one remote memory access from another processor P_j
- At a given moment, a processor P_i can execute a remote memory access to P_j and serve another one from P_k

Parallel time, speedup and efficiency example

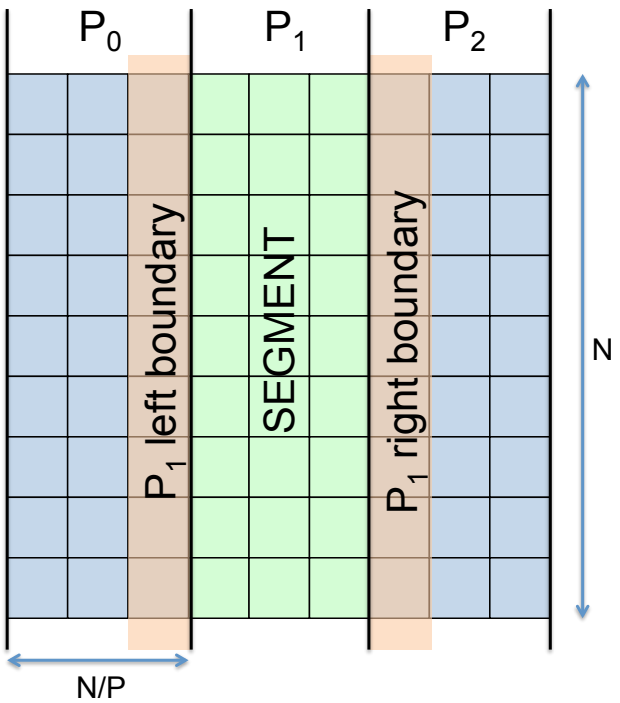
- Stencil algorithm to compute each element of matrix `utmp` using 4 neighbor values of matrix `u`

```
#include <math.h>
void compute( int N, double *u, double *utmp) {
    int i, k;
    double tmp;

    for ( i = 1; i < N-1; i++ ) {
       for ( k = 1; k < N-1; k++ ) {
          tmp = u[N*(i+1) + k] + u[N*(i-1) + k] + u[N*i + (k+1)] + u[N*i + (k-1)]
                      - 4 * u[N*i + k];
          utmp[N*i + k] = tmp/4;
       }
    }
}
```

- If t_c is the computation time for one element, which is the sequential time for an $N \times N$ matrix?

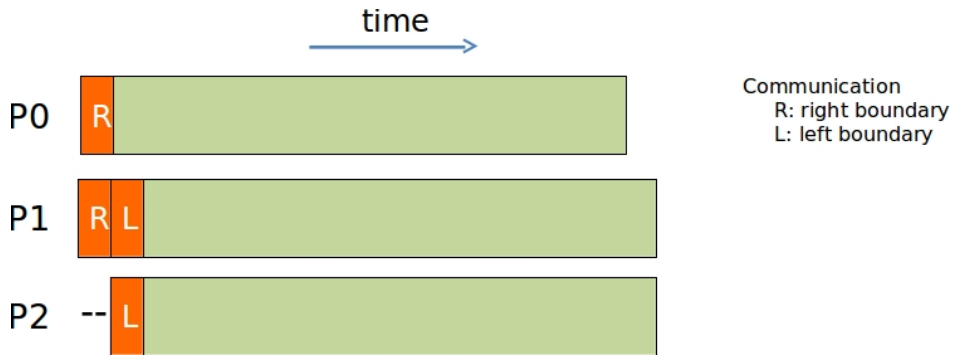
Parallel time, speedup and efficiency example (cont.)



- Data decomposition:
  - Column decomposition, each processor with N^2/P elements of `u` matrix and N^2/P elements of `utmp` matrix (segment)
  - Left and right boundaries, each with N elements
  - No need to gather `utmp` in one of the processors at the end of the computation

Parallel time, speedup and efficiency example (cont.)

- Parallelization strategy:
  1. Exchange boundaries with the two adjacent processors
  2. Each processor computes the elements of its `utmp` segment



- Questions:
  1. What is the data sharing time per segment assuming each boundary is accessed using a single message?
  2. What is the total time (computation and data sharing)?
  3. Obtain the expression for the speed–up S_P

Parallel time, speedup and efficiency example (cont.)

Assuming that N is very large, so that $N - 2 \simeq N$

- The total time for the algorithm is:

$$T_P = t_c \frac{N^2}{P} + 2(t_s + t_w N)$$

- The corresponding expression for the speedup is

$$S_P = \frac{t_c N^2}{t_c \frac{N^2}{P} + 2(t_s + t_w N)}$$

Blocking parallelization example

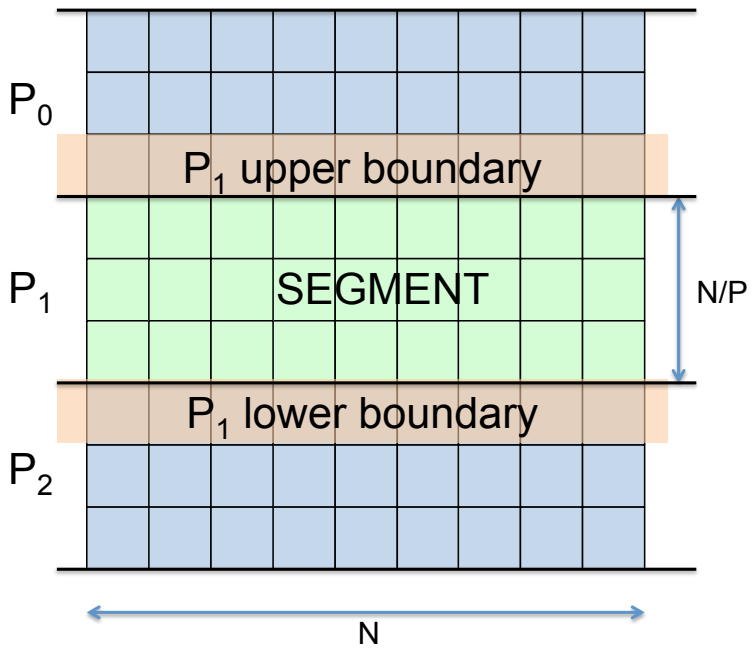
- Stencil algorithm to **update** (in place) each element of matrix u using its 4 neighbors

```
#include <math.h>
void compute( int N, double *u) {
    int i, k;
    double tmp;

    for ( i = 1; i < N-1; i++ ) {
       for ( k = 1; k < N-1; k++ ) {
          tmp = u[N*(i+1) + k] + u[N*(i-1) + k] + u[N*i + (k+1)] + u[N*i + (k-1)]
                      - 4 * u[N*i + k];
          u[N*i + k] = tmp/4;
       }
    }
}
```

- If t_c is the computation time for one element, which is the sequential time for an $N \times N$ matrix?

Blocking parallelization example (cont.)

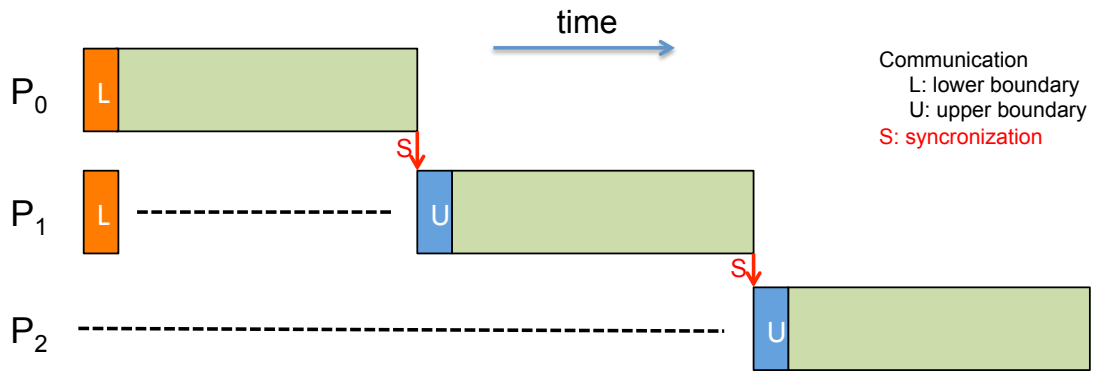


Data decomposition:

- Row distribution, each processor with N^2/P elements (segment)
- Upper and lower boundaries, each with N elements
- No need to gather final `u` in one of the processors

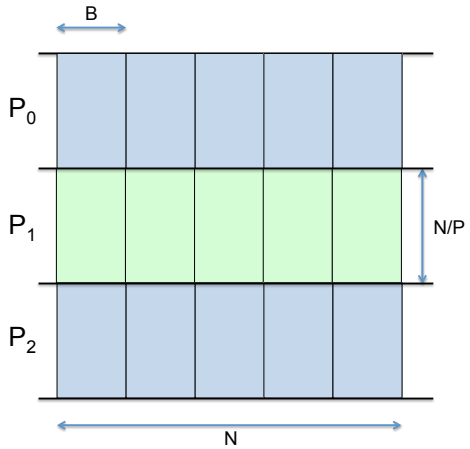
Blocking parallelization example (cont.)

- Parallelization strategy:
  1. Communication of lower boundary (no dependence)
  2. Wait for upper boundary (dependence with previous processor)
  3. Communication of the upper boundary
  4. Apply stencil algorithm to segment



- Questions:
  1. What is the data sharing time per segment?
  2. What is the total time (computation and data sharing)?

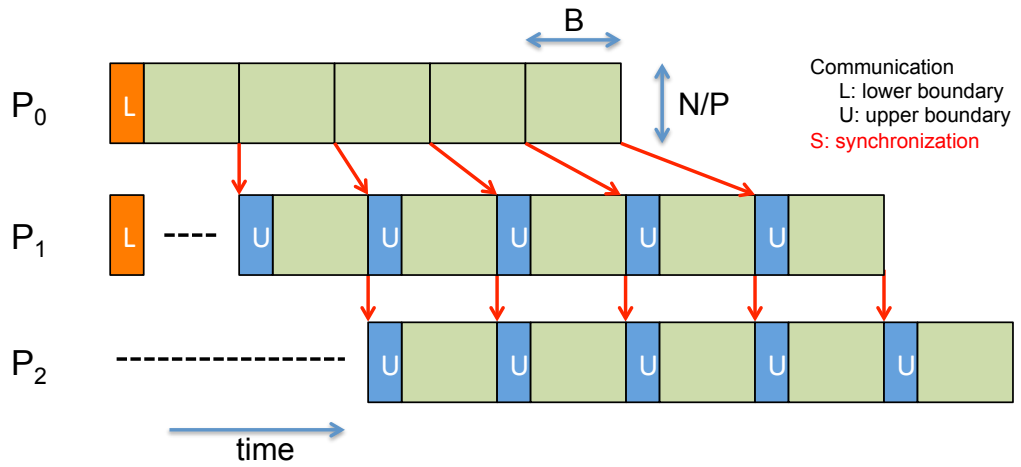
Blocking parallelization example (cont.)



- Data decomposition:
  - Row distribution, each processor with N^2/P elements (segment, logically divided in blocks of B columns)
  - Upper and lower boundaries, each with N elements
- Parallelization strategy:

  Once: Communication of lower boundary (no dependence)
  For each block:
  - Wait for termination of block in previous processor (dependence)
  - Communication of B elements for the upper boundary
  - Apply stencil algorithm to block (B columns) in segment

Blocking parallelization example: T_p



Questions:

1. What is the overhead of data sharing (before and during the parallel computation)?
2. What is the total time (computation and data sharing)?
3. Is there an optimum value for B?

Blocking parallelization example: T_p simplified

Assuming that N is very large, so that $N - 2 \simeq N$

$$T_p = (t_s + Nt_w) + (\frac{N}{B} + P - 2)(t_s + t_w B) + (\frac{N}{P} B)(\frac{N}{B} + P - 1)t_c$$

In order to obtain the optimum block size B we have to apply the derivative and equal it to zero. Assuming $P >> 2$:

$$T_p \simeq (t_s + Nt_w) + (\frac{N}{P} B)(\frac{N}{B} + P)t_c + (\frac{N}{B} + P)(t_s + t_w B)$$

Blocking parallelization example: optimum B computation

The optimum B block size is:

$$\frac{\partial T}{\partial B} = Nt_c - t_s\frac{N}{B^2} + t_wP = 0$$

$$B_{opt} = \sqrt{\frac{t_sN}{Nt_c + t_wP}} = \sqrt{\frac{t_s}{t_c + t_w\frac{P}{N}}}$$

If $N >> P$ then

$$B_{opt} \simeq \sqrt{\frac{t_s}{t_c}}$$

Parallelism (PAR)

Analysis of parallel applications

Eduard Ayguade, Josep Ramon Herrero and Daniel Jiménez
({eduard,josepr,djimenez}@ac.upc.edu)

Computer Architecture Department
Universitat Politècnica de Catalunya, UPC–BarcelonaTECH

2014/15-Fall