Parallelism (PAR)

Understanding parallelism

Eduard Ayguadé, José Ramón Herrero and Gladys Utrera

Computer Architecture Department
Universitat Politècnica de Catalunya

Course 2019/20 (Fall semester)

Additional learning material for this lesson

- Atenea: Unit 2.1 Understanding parallelism I
  - Video lesson 2 (partially covering slides 5–20)
  - Questions after video lesson 2
  - Going further: Excel to explore the effect of task decomposition overheads
- Atenea: Unit 2.2 Understanding parallelism II
  - Video lesson 3 (partially covering slides 27–38)
  - Questions after video lesson 3
  - Going further: Excel to explore the effect of data sharing overheads
- Collection of Exercises: problems in Chapter 2
- Selection of Exams (with Solutions)

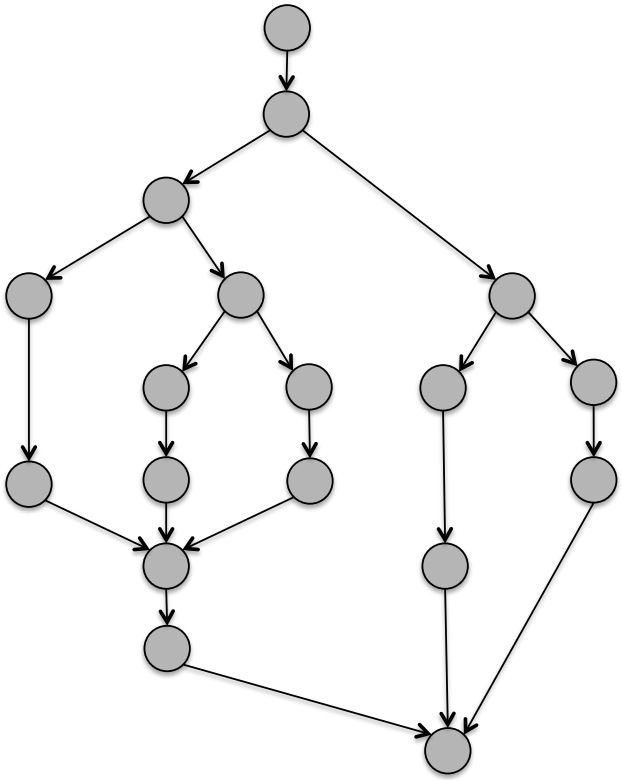
Outline

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. elements of a vector, rows of a matrix) (implies task decomposition)
- There may be (data or control) dependencies between tasks
- The decomposition determines the potential paralellism that could be obtained

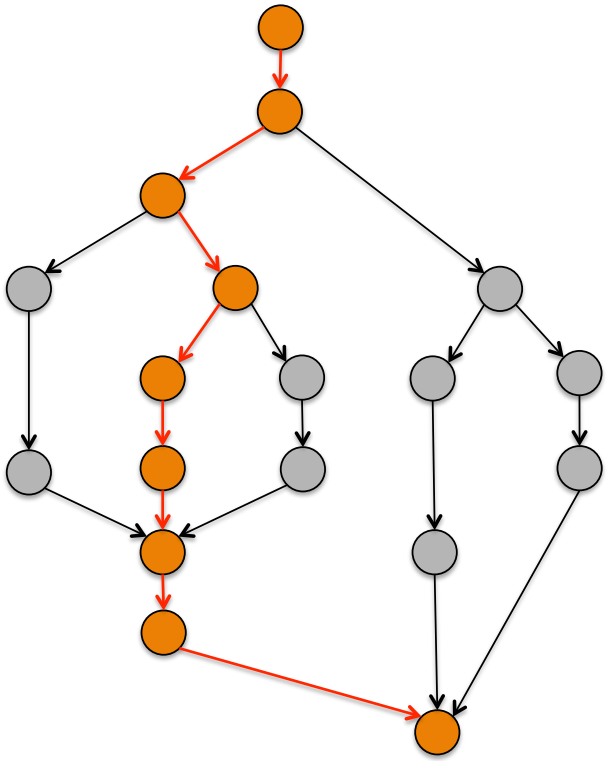
[1] Different strategies covered during this course

Task graph abstraction



- Directed Acyclic Graph
- Node = **task**
  - Arbitrary sequential computation
  - Its weight represents the amount of work to be done
- Edge = **dependence**
  - Successor node needs (reads) data produced (written) by predecessor node (other kind of dependences later)
  - Execution order between successor and predecessor

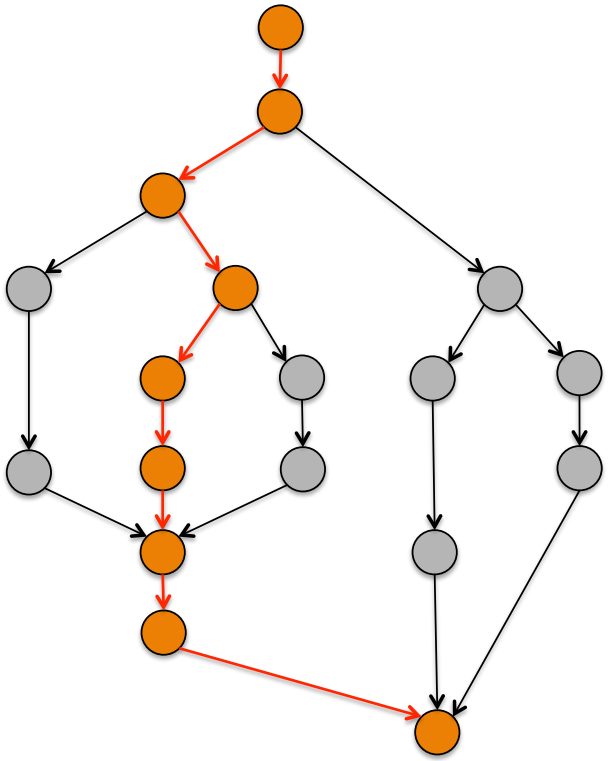
Computing the execution time



- Processor abstraction (simplification)
  - P identical processors
  - Each processor executes one node at a time

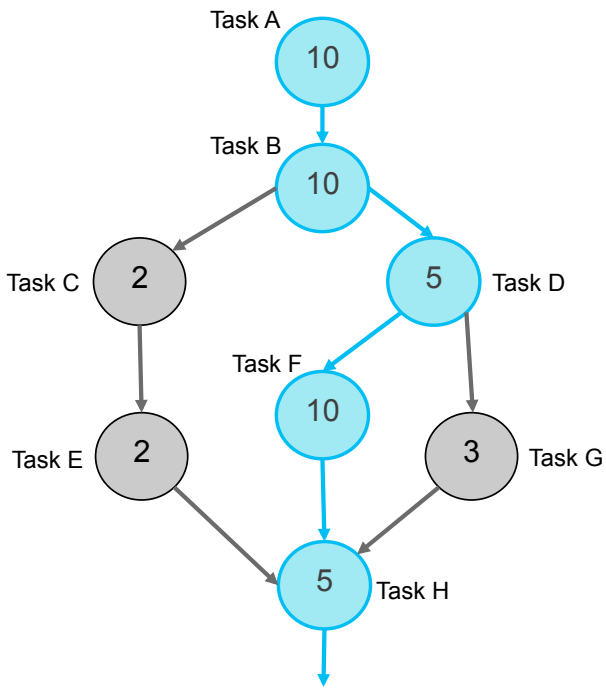
- $T_1 = \sum_{i=1}^{nodes}(work\_node_i)$
- $T_\infty = \sum_{i \in criticalpath}(work\_node_i)$, assuming sufficient (infinite) resources

Computing the parallelism



- $Parallelism = T_1/T_\infty$, independent of number of processors P
  - How fast would I go if sufficient (infinite) resources were available
- P_{min} is the minimum number of processors necessary to achieve $Parallelism$

Computing the parallelism: simple example



- $T_1 = Tasks_{ABCDEFGH} = 47$
- Possible paths:

$Tasks_{ABCEH} = 29$
$Tasks_{ABDFH} = 40$
$Tasks_{ABDGH} = 33$

- $T_\infty = Tasks_{ABDFH} = 40$
- $Parallelism = 47/40 = 1.175$
- $P_{min} = 2$

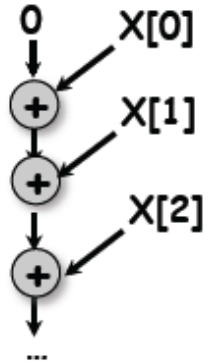
Example 1: vector sum

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



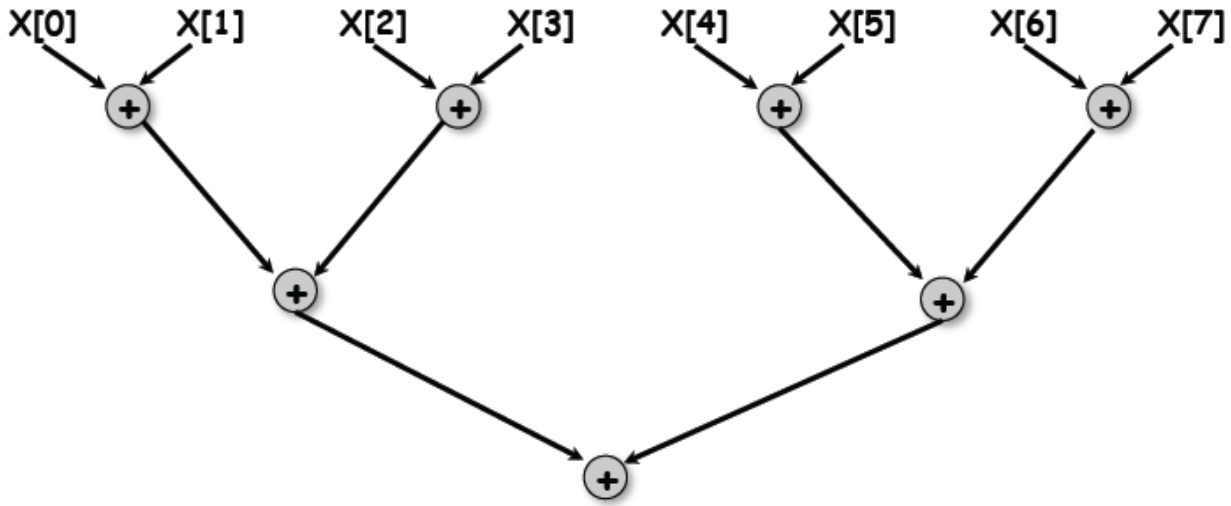
$$T_1 \propto n$$
$$T_\infty \propto n$$
$$Parallelism = 1$$

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

Example 1: vector sum

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

- An alternative computation graph



- $T_1 \propto n$; $T_\infty \propto \log_2(n)$; $Parallelism \propto (n \div \log_2(n))$
- How to restructure the sequential algorithm to have this computation graph? (iterative vs. recursive solutions)

Example 1: vector sum

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

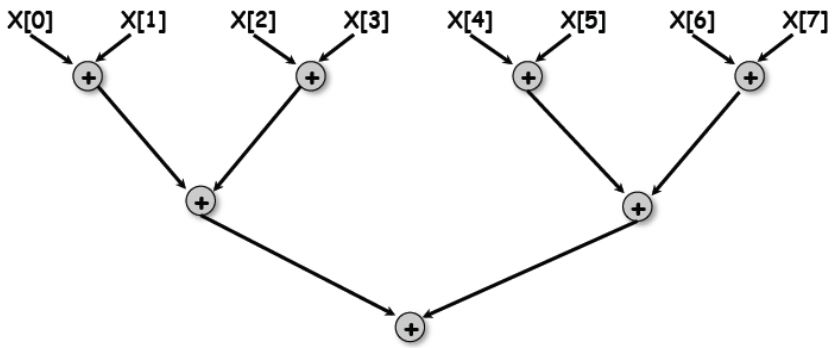
Sequential **recursive** algorithm:

```
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 2: database query processing

Consider the following database with 10 records:

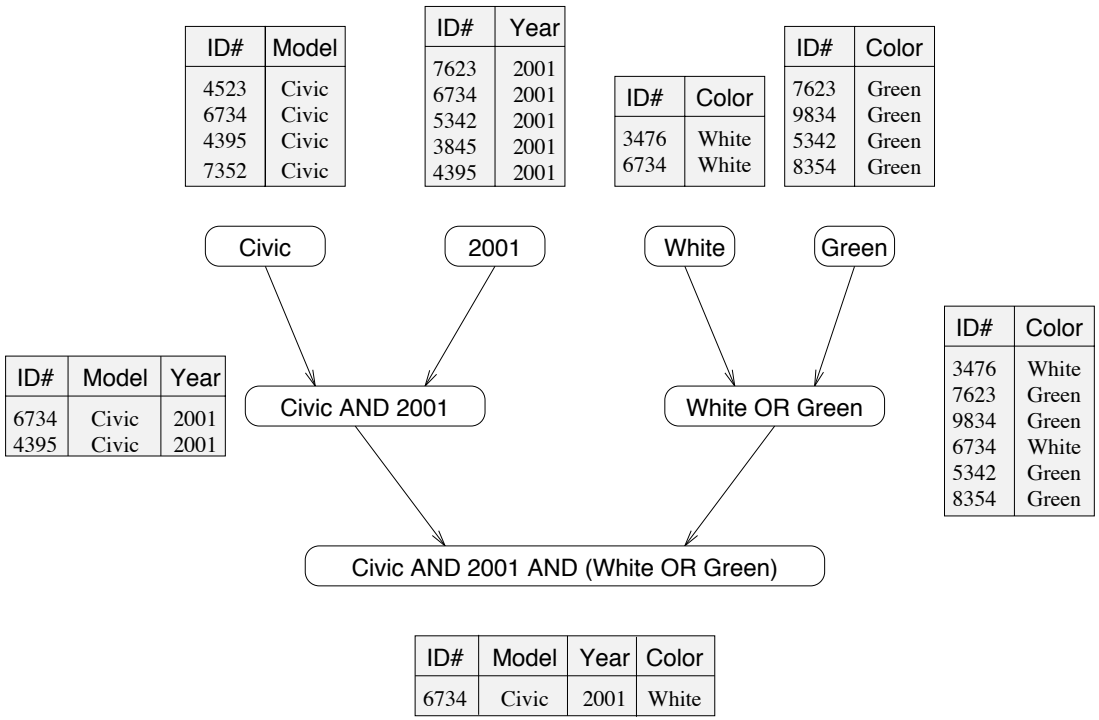
ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000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

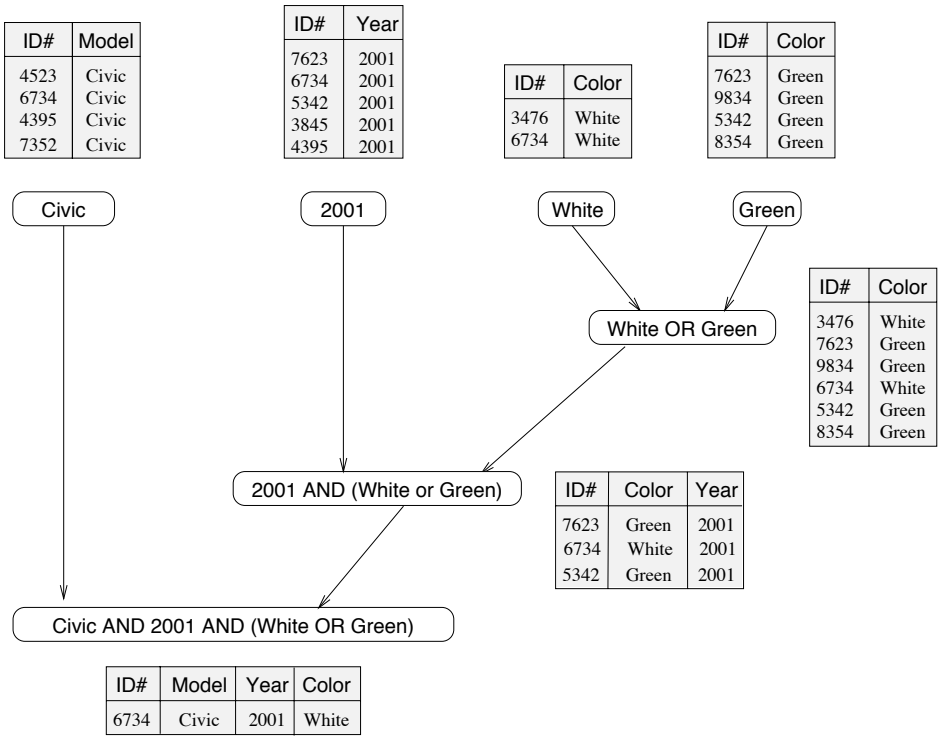
Example 2: database query processing

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 2: database query processing

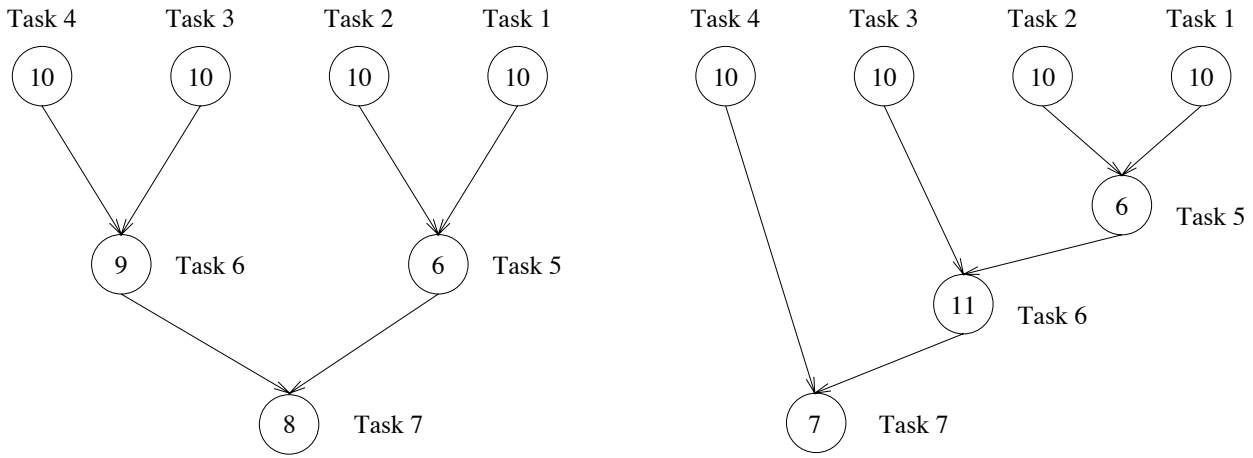
Note that the same problem can be decomposed into tasks in other ways as well.



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

Example 2: database query processing

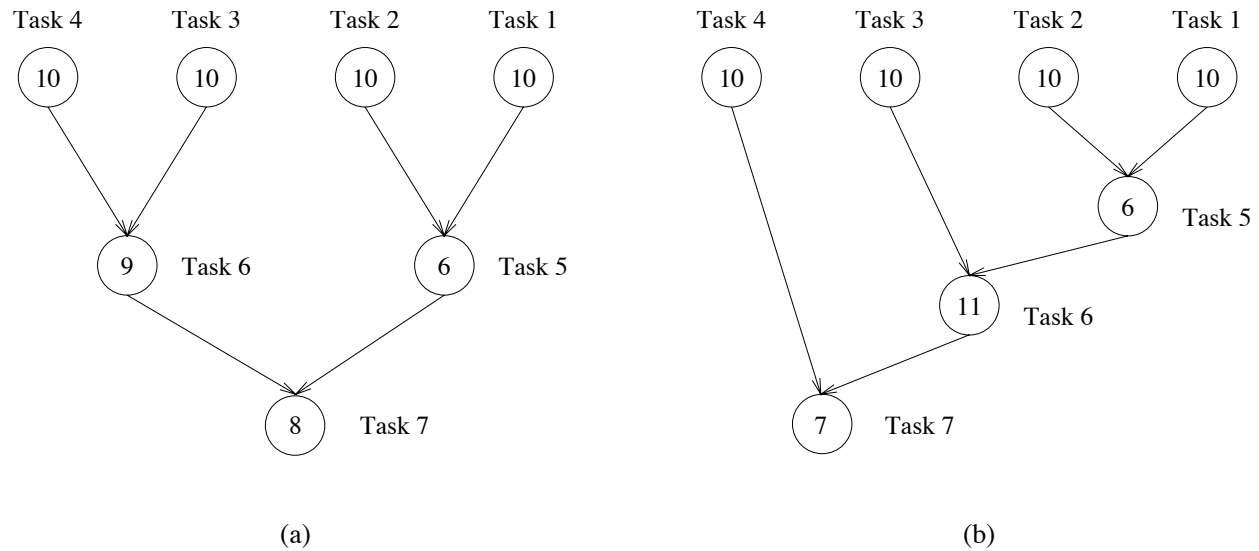
Consider the task dependency graphs of the two database query decompositions[2]:



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 2: database query processing



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

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

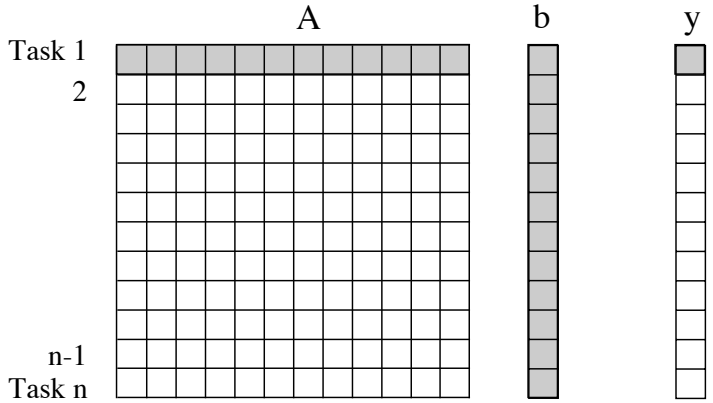
Question: How many processors P_{min} are needed in each case to achieve this $Parallelism$?

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

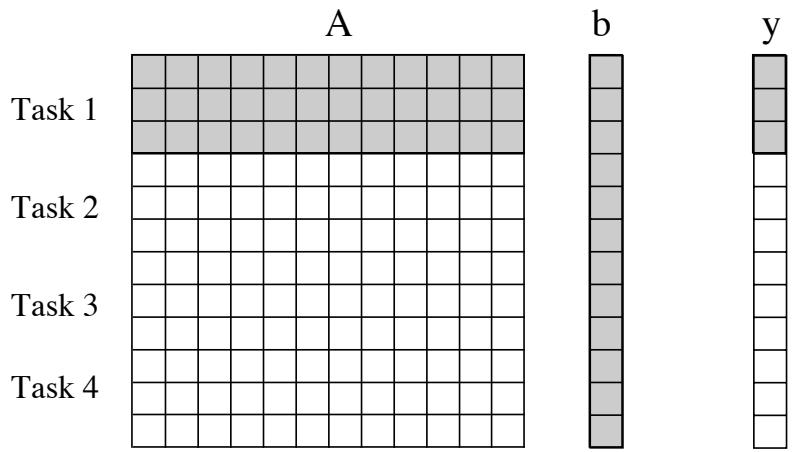
Example: matrix-vector product (n by n matrix):



- 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: creation of tasks, task synchronization, exchange of data between tasks, ...)
  - The granularity may determine performance bounds

Example 3: stencil computation using Jacobi solver

Stencil algorithm that computes each element of matrix `utmp` using 4 neighbor elements of matrix `u`, both matrices with $n \times n$ elements

```
void compute(int n, double *u, double *utmp) {
int i, j;
double tmp;

for (i = 1; i < n-1; i++) {
    for (j = 1; j < n-1; j++) {
        tmp = u[n*(i+1) + j] + u[n*(i-1) + j] +  // elements u[i+1][j] and u[i-1][j]
              u[n*i + (j+1)] + u[n*i + (j-1)] -  // elements u[i][j+1] and u[i][j-1]
              4 * u[n*i + j];                    // element u[i][j]
        utmp[n*i + j] = tmp/4;                   // element utmp[i][j]
    }
}}
```

Example 3: stencil computation using Jacobi solver

What tasks can be? Assume: 1) the innermost loop body takes t_{body} time units; and 2) n is very large, so that $n - 2 \simeq n$

Task is ... (granularity)	**Num. tasks**	**Task cost**	**T_1**	**T_∞**	**Parallelism**
All iterations of i and j loops	1	$n^2 \cdot t_{body}$	$n^2 \cdot t_{body}$	$n^2 \cdot t_{body}$	1
Each iteration of i loop	n	$n \cdot t_{body}$	$n^2 \cdot t_{body}$	$n \cdot t_{body}$	n
Each iteration of j loop	n^2	t_{body}	$n^2 \cdot t_{body}$	t_{body}	n^2
r consecutive iterations of i loop	$n \div r$	$n \cdot r \cdot t_{body}$	$n^2 \cdot t_{body}$	$n \cdot r \cdot t_{body}$	$n \div r$
c consecutive iterations of j loop	$n^2 \div c$	$c \cdot t_{body}$	$n^2 \cdot t_{body}$	$c \cdot t_{body}$	$n^2 \div c$
A block of r x c iterations of i and j, respectively	$n^2 \div (r \cdot c)$	$r \cdot c \cdot t_{body}$	$n^2 \cdot t_{body}$	$r \cdot c \cdot t_{body}$	$n^2 \div (r \cdot c)$

Finer grain task decomposition $\rightarrow$ higher parallellism, but ...

Example 3: stencil computation using Jacobi solver

... what if each task creation takes t_{create}?

Task is ... (granularity)	**Num. tasks**	**Task cost**	**Task creation ovh**
All iterations of i and j loops	1	$n^2 \cdot t_{body}$	t_{create}
Each iteration of i loop	n	$n \cdot t_{body}$	$n \cdot t_{create}$
Each iteration of j loop	n^2	t_{body}	$n^2 \cdot t_{create}$
r consecutive iterations of i loop	$n \div r$	$n \cdot r \cdot t_{body}$	$(n \div r) \cdot t_{create}$
c consecutive iterations of j loop	$n^2 \div c$	$c \cdot t_{body}$	$(n^2 \div c) \cdot t_{create}$
A block of r x c iterations of i and j, respectively	$n^2 \div (r \cdot c)$	$r \cdot c \cdot t_{body}$	$(n^2 \div (r \cdot c)) \cdot t_{create}$

Trade-off between task granularity and task creation overhead

Example 4: stencil computation using Gauss-Seidel solver

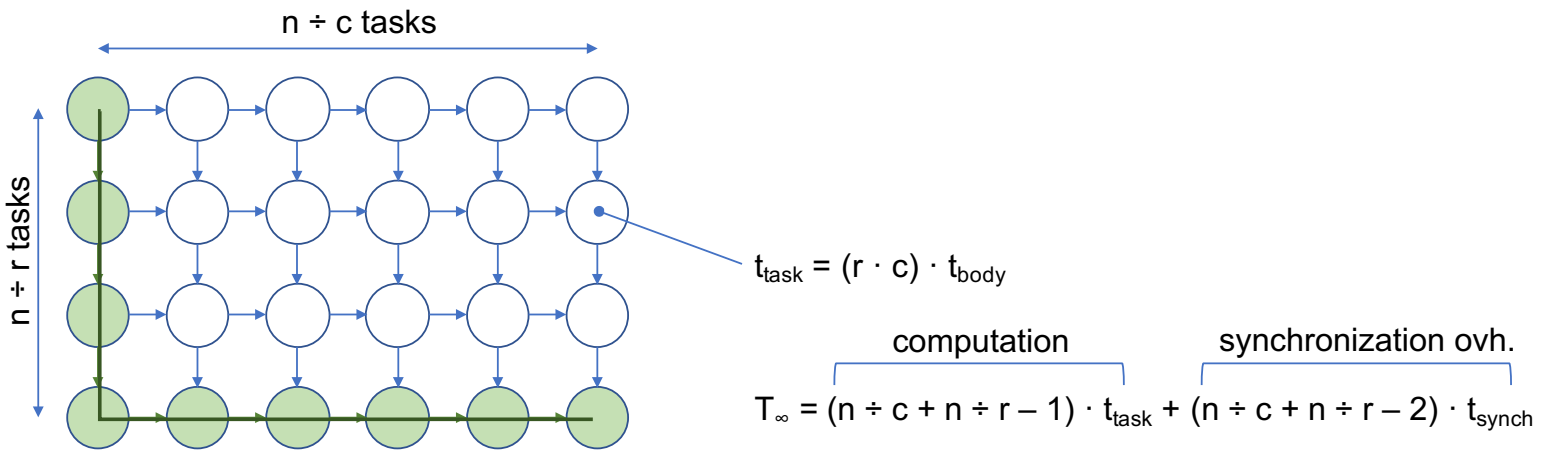
Stencil algorithm that **updates** (in place) each element of matrix u using its 4 neighbors, matrix size $n \times n$ elements.

```
void compute(int n, double *u, double *utmp) {
int i, j;
double tmp;

for (i = 1; i < n-1; i++) {
    for (j = 1; j < n-1; j++) {
        tmp = u[n*(i+1) + j] + u[n*(i-1) + j] +  // elements u[i+1][j] and u[i-1][j]
              u[n*i + (j+1)] + u[n*i + (j-1)] -  // elements u[i][j+1] and u[i][j-1]
              4 * u[n*i + j];                    // element u[i][j]
        u[n*i + j] = tmp/4;                      // element u[i][j]
    }
}}
```

Example 4: stencil computation using Gauss-Seidel solver

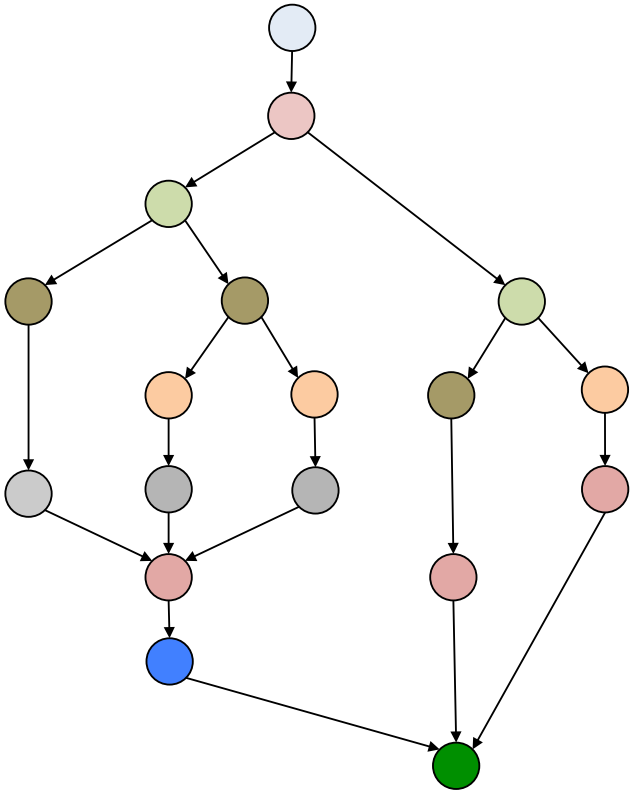
Assuming: 1) each task computes a block of $r \times c$ iterations of the `i` and `j` loops, respectively; and 2) each task synchronization takes t_{synch}



Again, trade-off between task granularity and task synchronisation overhead

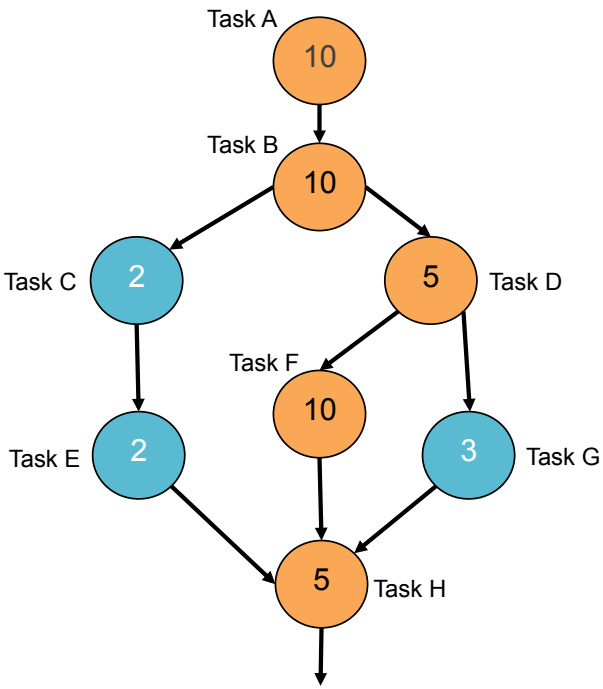
Outline

Speedup

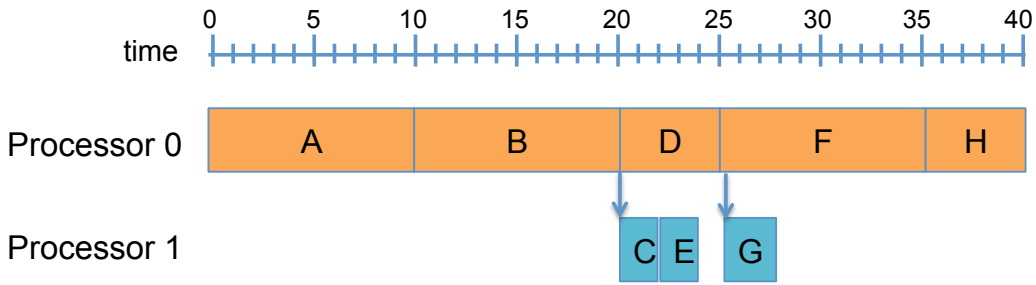


- T_p = execution time on P processors (depends on the schedule of the graph nodes on the processors)
- Lower bounds
  - $T_p \geq T_1/P$
  - $T_p \geq T_\infty$
- $Speedup$ on P processors: $S_p = T_1/T_p$

Speedup: simple example



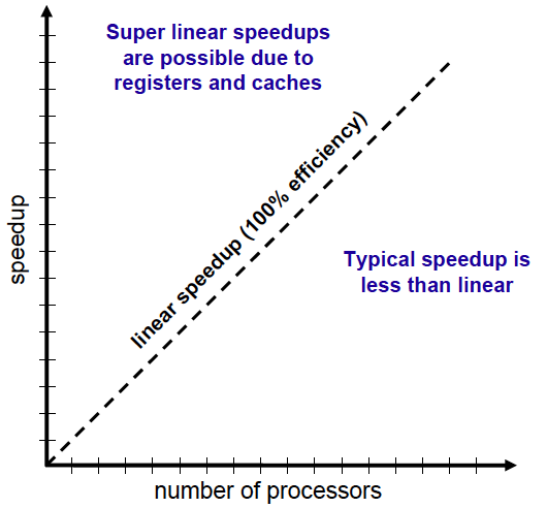
- **Task scheduling**: how could tasks be assigned to two processors?



- In this case: $T_2 = 40$, $S_2 = 47/40 = 1.175$

Speedup vs. efficiency

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

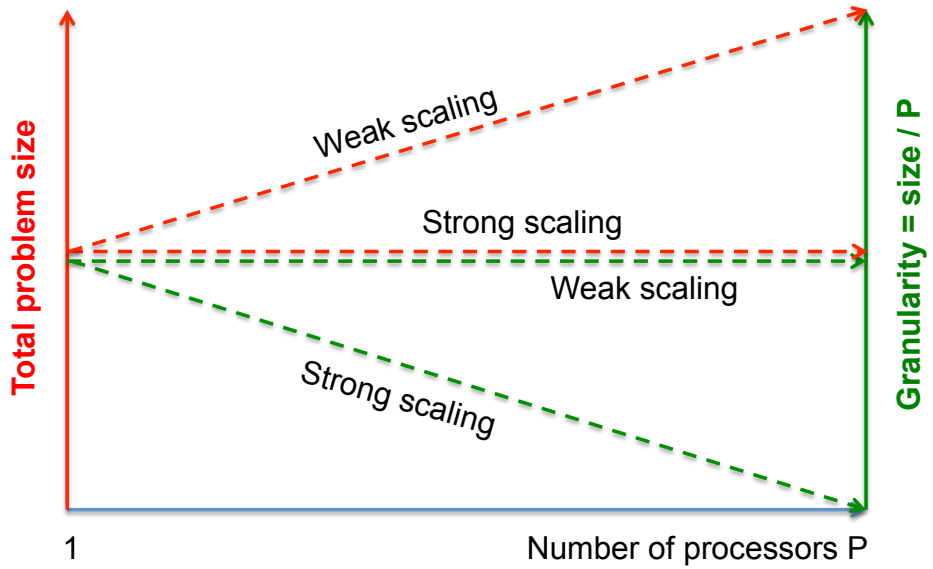


- Efficiency Eff_p: it is a measure of the fraction of time for which a processing element is usefully employed
  - $Eff_p = T_1/(T_p \times P)$
  - Also, $Eff_p = S_p/P$

Strong vs. weak scalability

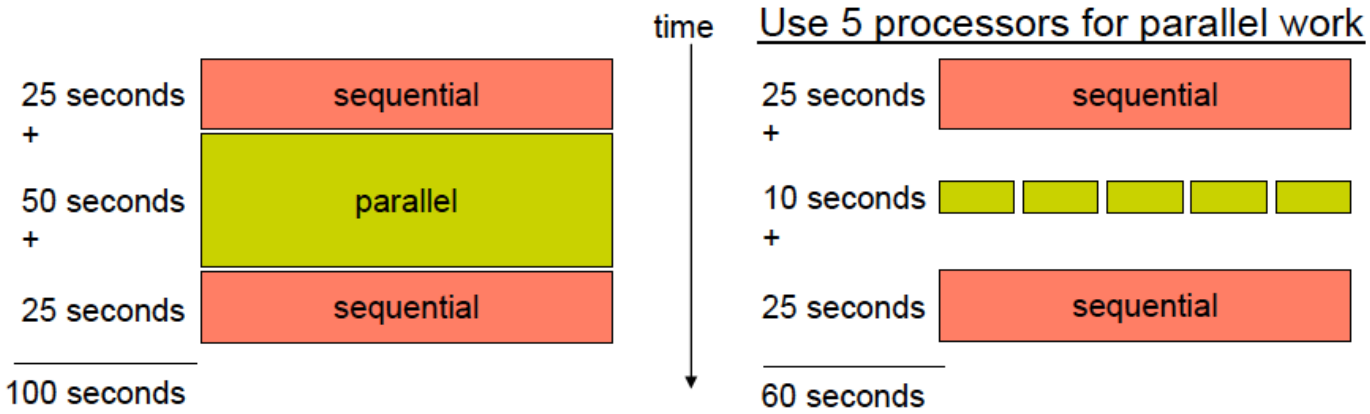
Two usual scenarios to evaluate the scalability of one application:

- Increase the number of processors P with constant problem size (strong scaling $\rightarrow$ reduce the execution time)
- Increase the number of processors P with problem size proportional to P (weak scaling $\rightarrow$ solve larger problem)



Amdahl's law

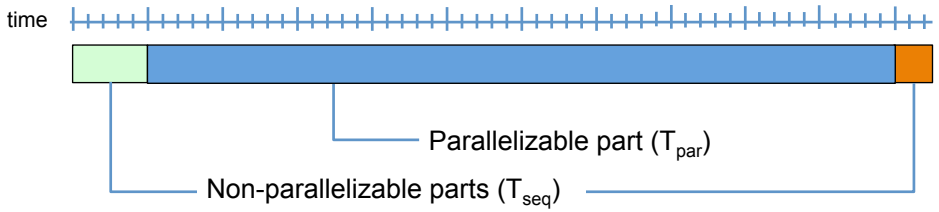
The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used



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

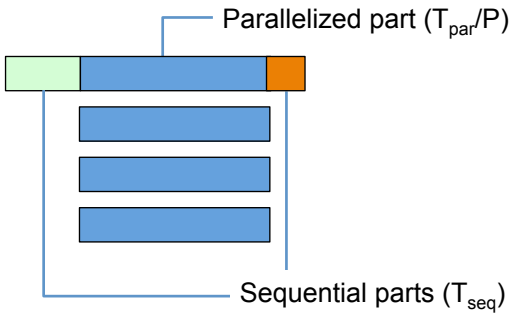
Amdahl's law

Assume the following simplified case, where the parallel fraction φ is the fraction of time the program runs in parallel



$$T_1 = T_{seq} + T_{par}$$
$$\varphi = T_{par}/T_1$$

$$T_1 = (1 - \varphi) \times T_1 + \varphi \times T_1$$



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

Amdahl's law

From where we can compute the speed–up S_P that can be achieved as

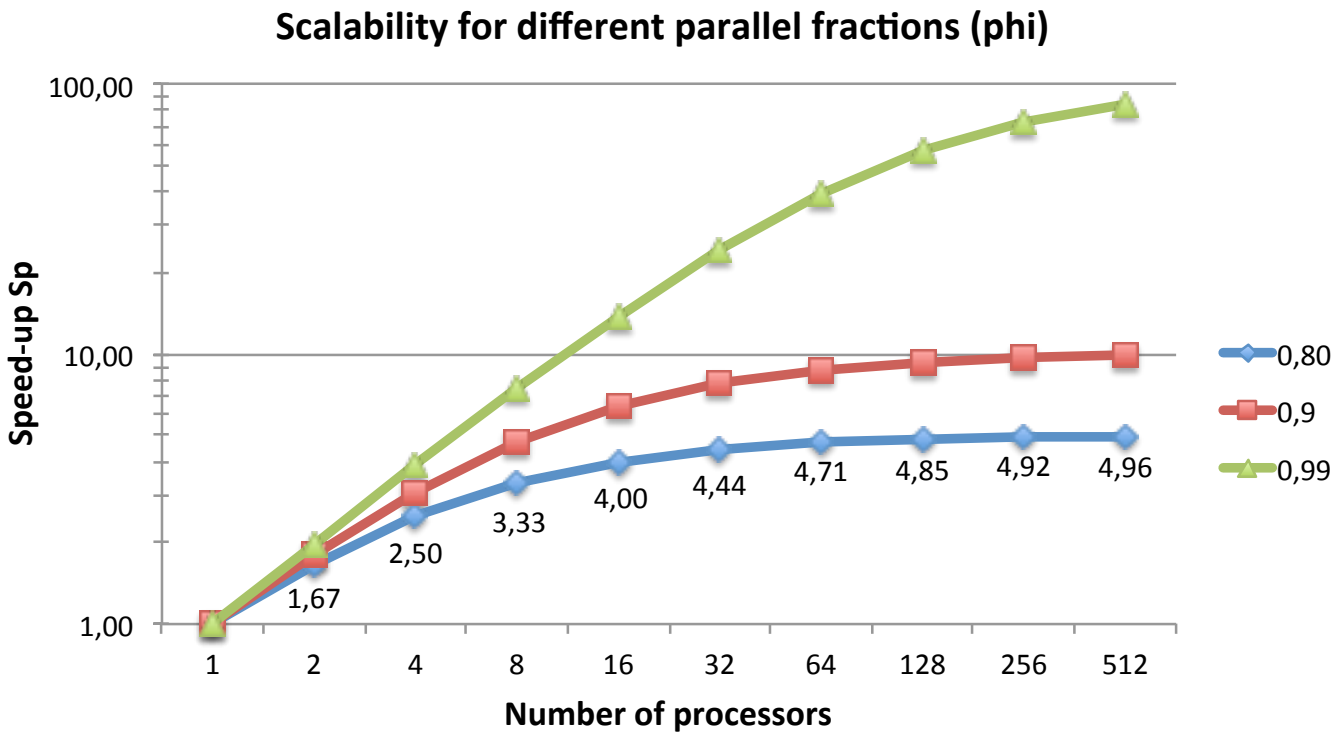
$$S_p = \frac{T_1}{T_p} = \frac{T_1}{(1-\varphi) \times T_1 + (\varphi \times T_1/P)}$$

$$S_p = \frac{1}{((1-\varphi) + \varphi/P)}$$

Two particular cases:

$$\varphi = 0 \rightarrow S_p = 1$$
$$\varphi = 1 \rightarrow S_p = P$$

Amdahl's law

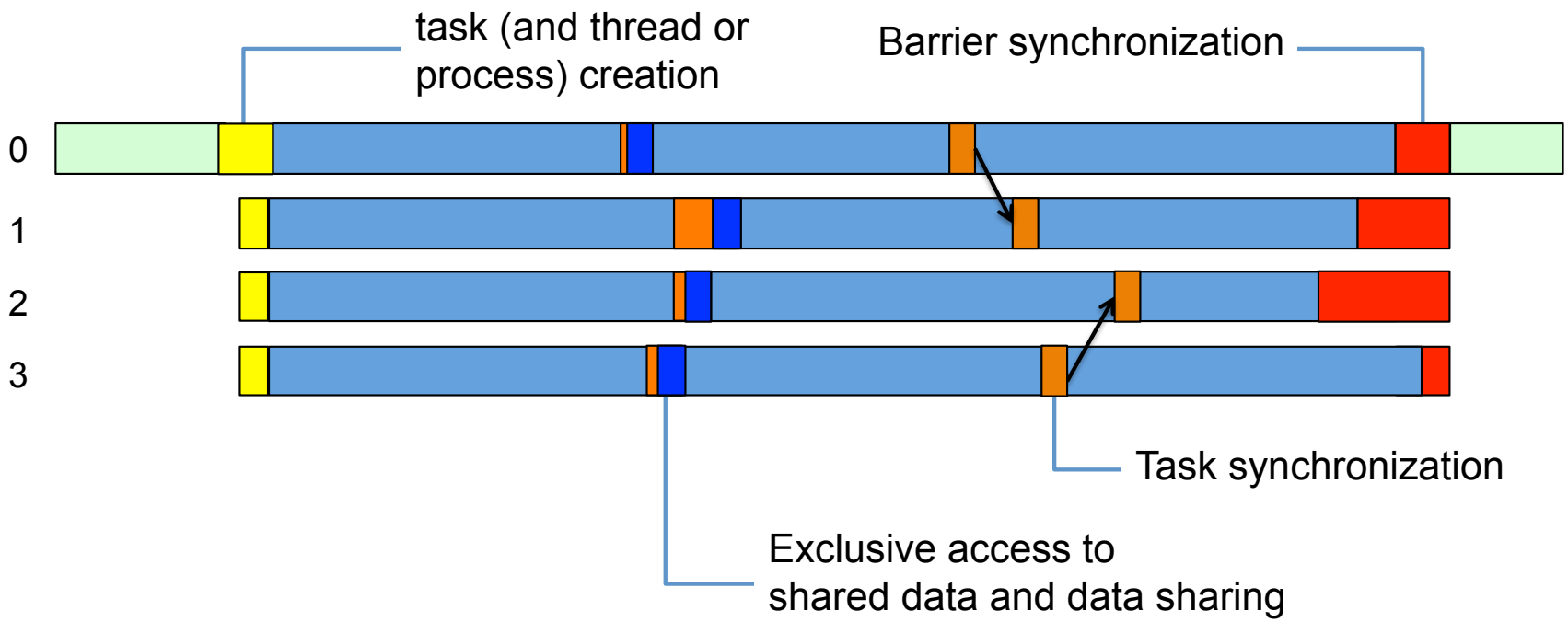


When $P \to \infty$ the expression of the speed–up becomes

$$S_p \to \frac{1}{(1-\varphi)}$$

Sources of overhead

Parallel computing is not for 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)

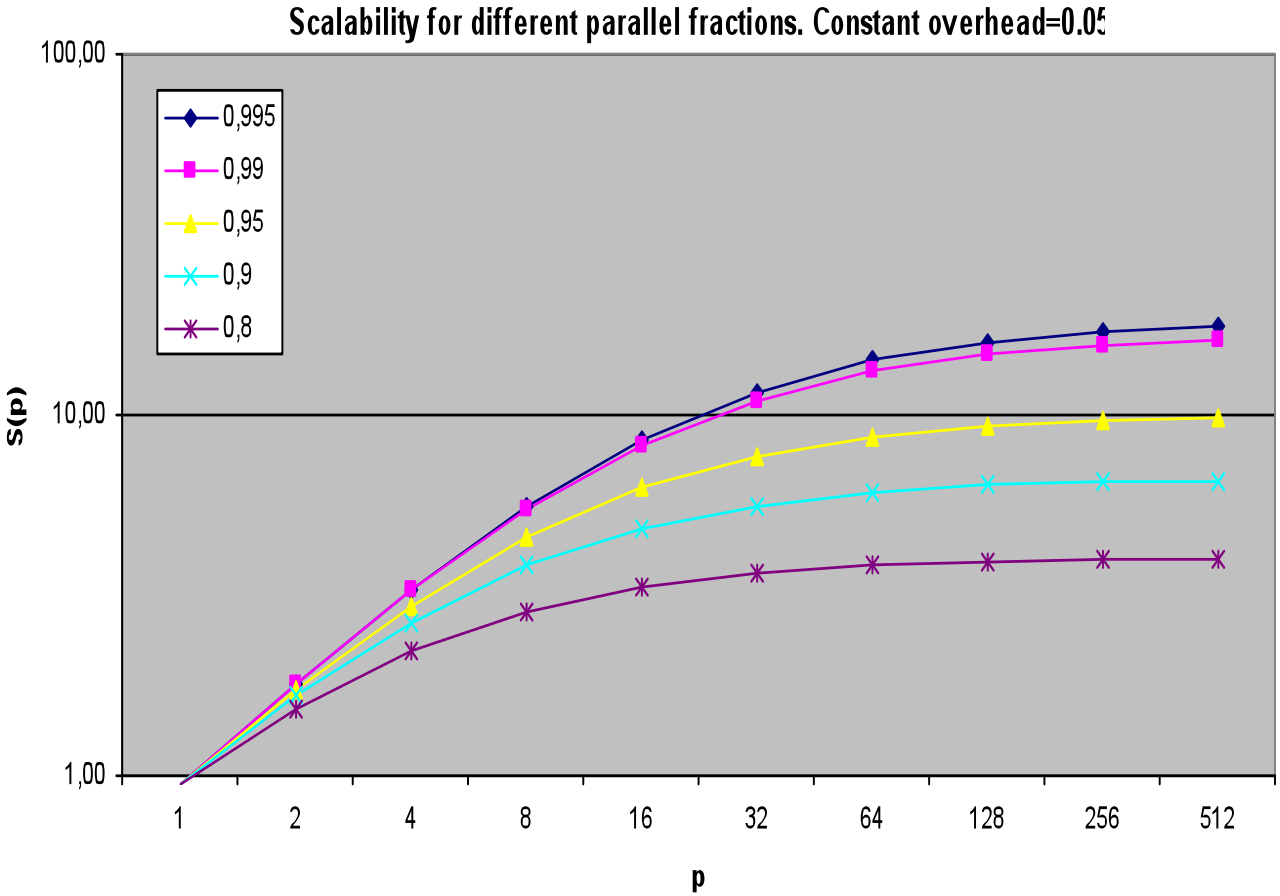


Other sources of overhead

- **Data sharing:** can be explicit via messages, or implicit via a memory hierarchy (caches)
- **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)

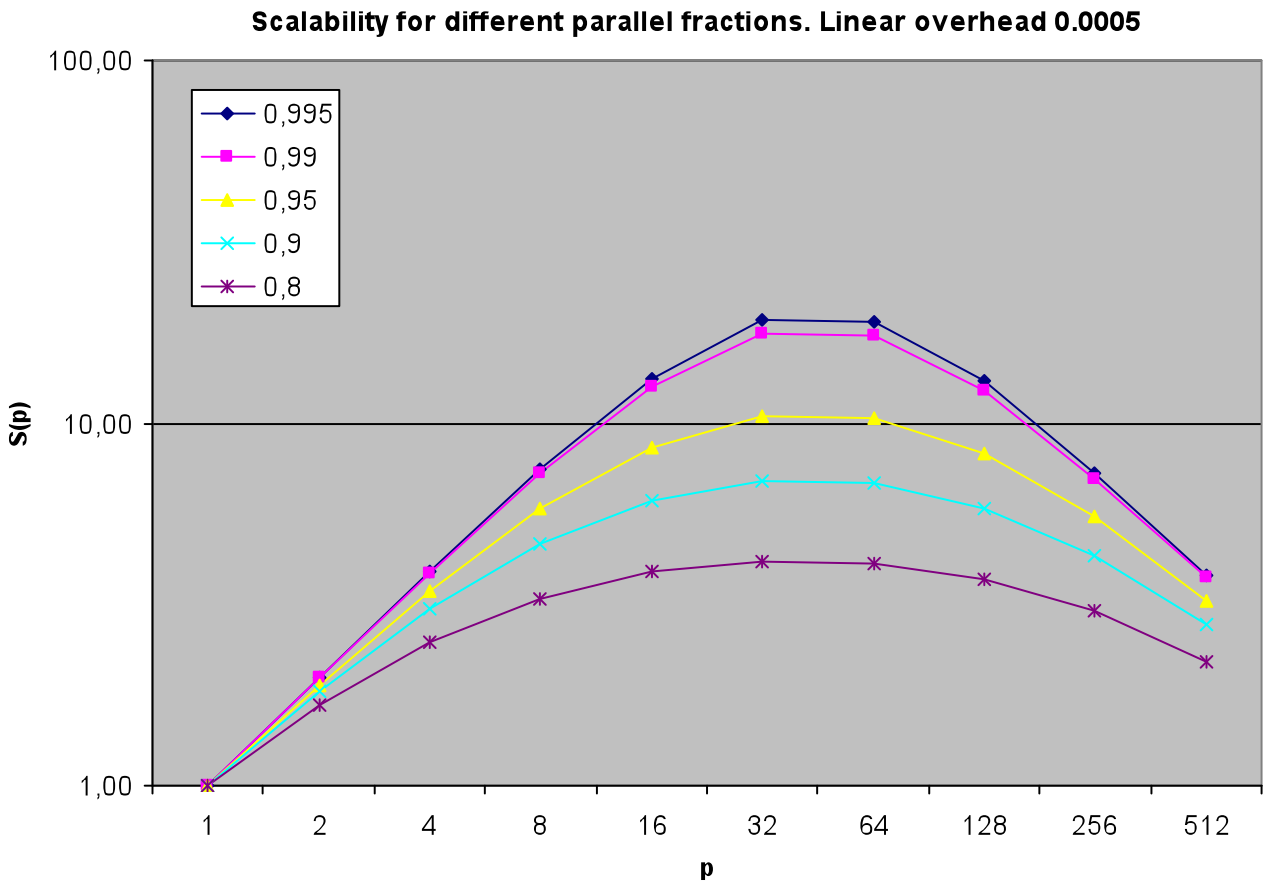
Amdahl's law (constant overhead)

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



Amdahl's law (linear overhead)

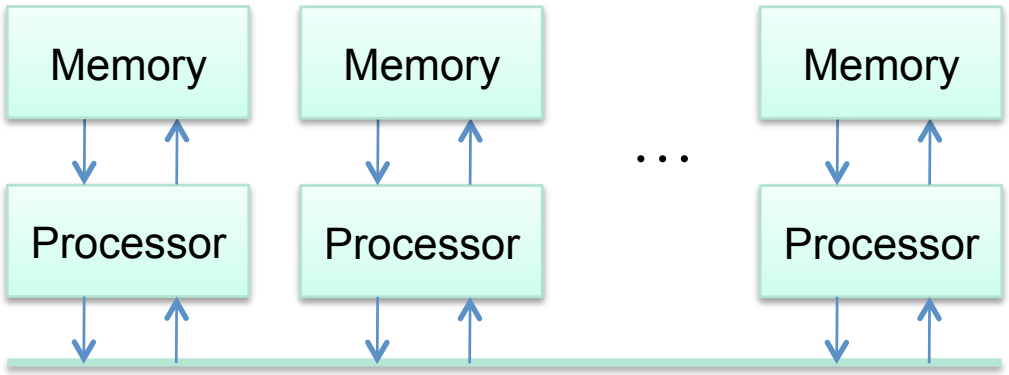
$$T_p = (1 - \varphi) \times T_1 + \varphi \times T_1/p + overhead(p)$$



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 in preparing the remote access (t_s)
  - Transfer: time spent in transferring the message (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

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

How to model data sharing overhead?

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

Back to example 3: Jacobi solver

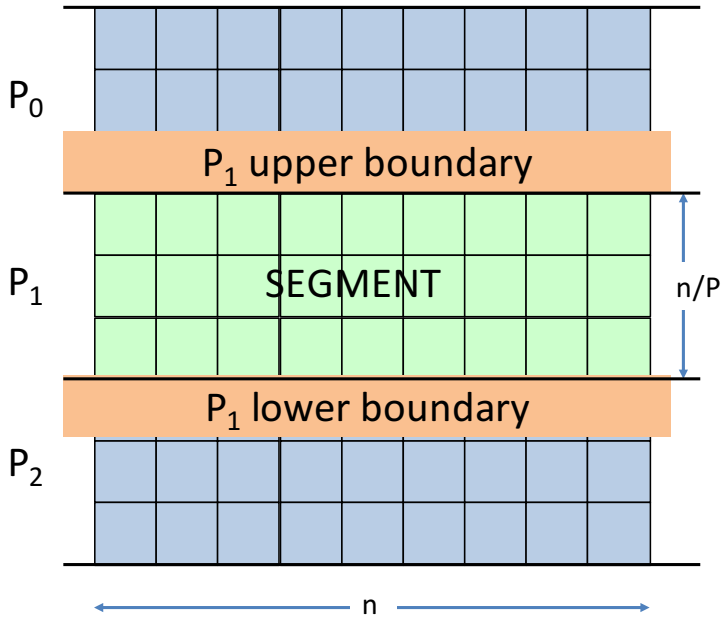
Stencil algorithm: each element of matrix `utmp` computed using 4 neighbor elements of matrix `u`, both matrices with $n \times n$ elements

```
void compute(int n, double *u, double *utmp) {
int i, j;
double tmp;

for (i = 1; i < n-1; i++) {
    for (j = 1; j < n-1; j++) {
        tmp = u[n*(i+1) + j] + u[n*(i-1) + j] +  // elements u[i+1][j] and u[i-1][j]
              u[n*i + (j+1)] + u[n*i + (j-1)] -  // elements u[i][j+1] and u[i][j-1]
              4 * u[n*i + j];                    // element u[i][j]
        utmp[n*i + j] = tmp/4;                   // element utmp[i][j]
    }
}}
```

Task definition: $n \div P$ consecutive iterations of i loop, being P the number of processors

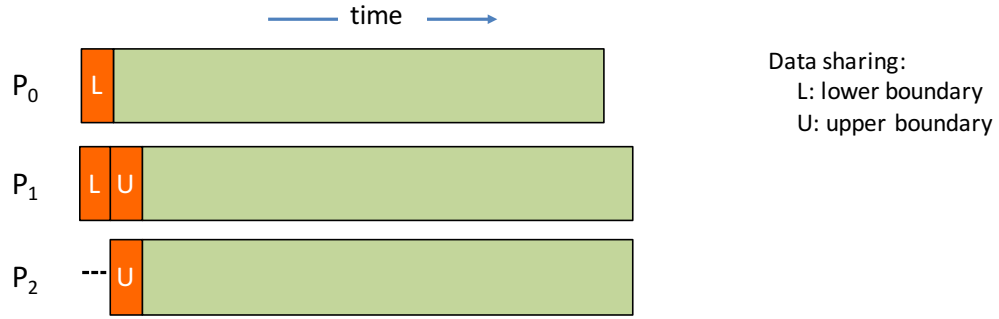
Example 3: data decomposition and movement



- Row decomposition, each processor with n^2/P elements of `u` matrix and n^2/P elements of `utmp` matrix (segment)
- Upper and lower boundaries, each with n elements
- No need to gather `utmp` in one of the processors at the end of the computation

Example 3: parallel execution timeline

- 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

Example 3: parallel execution time and speedup

With the same assumptions as before:

- The sequential execution time is $T_1 = n^2 \times t_{body}$
- The parallel execution time, considering both computation and data movement, is:

$$T_P = \frac{n^2}{P} \times t_{body} + 2 \times (t_s + n \times t_w)$$

- The corresponding expression for the speedup is

$$S_P = \frac{T_1}{T_P} = \frac{n^2 \times t_{body}}{\frac{n^2}{P} \times t_{body} + 2 \times (t_s + n \times t_w)}$$

Back to example 4: Gauss-Seidel solver

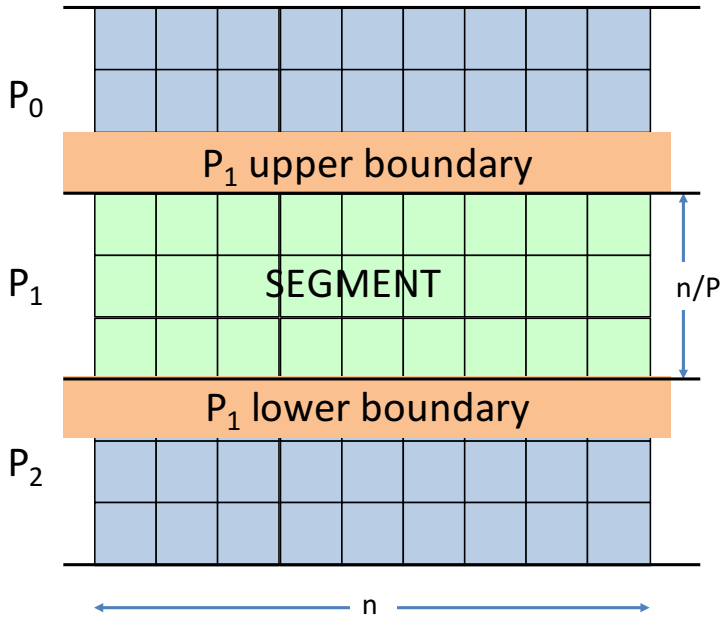
Stencil algorithm that **updates** (in place) each element of matrix u using its 4 neighbours, matrix size $n \times n$ elements.

```
void compute(int n, double *u, double *utmp) {
int i, j;
double tmp;

for (i = 1; i < n-1; i++) {
    for (j = 1; j < n-1; j++) {
        tmp = u[n*(i+1) + j] + u[n*(i-1) + j] +  // elements u[i+1][j] and u[i-1][j]
              u[n*i + (j+1)] + u[n*i + (j-1)] -  // elements u[i][j+1] and u[i][j-1]
              4 * u[n*i + j];                    // element u[i][j]
        u[n*i + j] = tmp/4;                      // element u[i][j]
    }
}}
```

Task definition 1: $n \div P$ consecutive iterations of i loop, being P the number of processors

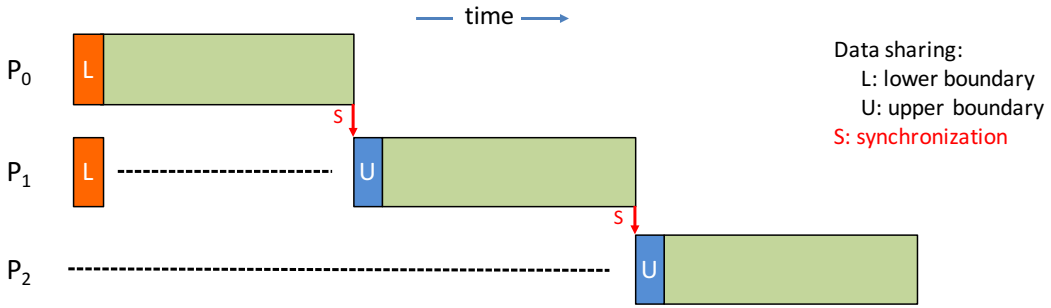
Example 4: data decomposition and movement



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

Example 4: parallel execution timeline

- Parallelization strategy:
  1. Access to lower boundary (if needed)
  2. Wait for upper boundary (dependence with previous processor, if any)
  3. Access to upper boundary (if needed)
  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)?

Example 4: Gauss-Seidel solver

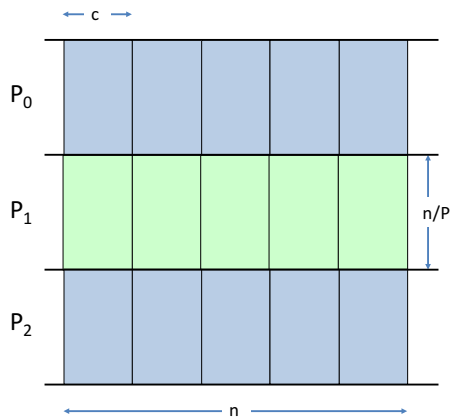
Stencil algorithm that **updates** (in place) each element of matrix u using its 4 neighbours, matrix size $n \times n$ elements.

```
void compute(int n, double *u, double *utmp) {
int i, j;
double tmp;

for (i = 1; i < n-1; i++) {
    for (j = 1; j < n-1; j++) {
        tmp = u[n*(i+1) + j] + u[n*(i-1) + j] +  // elements u[i+1][j] and u[i-1][j]
              u[n*i + (j+1)] + u[n*i + (j-1)] -  // elements u[i][j+1] and u[i][j-1]
              4 * u[n*i + j];                    // element u[i][j]
        u[n*i + j] = tmp/4;                      // element u[i][j]
    }
}}
```

Task definition 2: block of $n \div P$ by c consecutive iterations of i and j loops, respectively; P is the number of processors

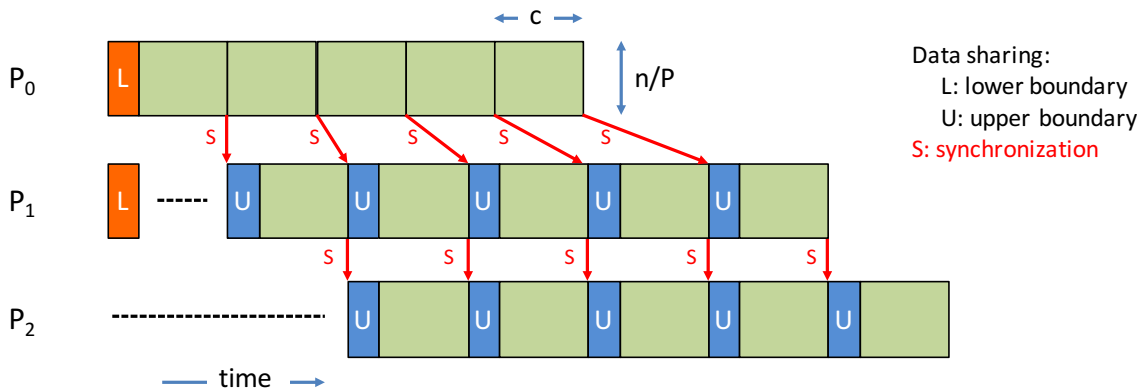
Example 4: blocking parallelization



- Data decomposition:
  - Row distribution, each processor with n^2/P elements
  - Tasks compute segments of $n \div c$ rows by c columns
- Parallelization strategy:

  Once: Access to n elements for lower boundary (if needed)
  For each block (task):
  - Wait for termination of task computing the same block in previous processor, if any (dependence)
  - Access to c elements for upper boundary (if needed)
  - Apply stencil algorithm to the block

Example 4: parallel execution timeline with blocking



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 c?

Example 4: parallel execution time and speedup

Assuming that t_{body} is the computation time for the innermost loop body and n is very large, so that $n - 2 \simeq n$

$$T_P = (\frac{n}{c} + P - 1) \times (\frac{n}{P} \times c) \times t_{body} \quad +$$

$$(t_s + n \times t_w) + ((\frac{n}{c} + P - 2) \times (t_s + c \times t_w))$$

The corresponding expression for the speedup would be

$$S_P = \frac{T_1}{T_P} = \frac{n^2 \times t_{body}}{T_P}$$

Example 4: optimum blocking factor

Assuming $P >> 2$:

$$T_P \simeq (\frac{n}{c} + P)(\frac{n}{P} \times c)t_{body} + (t_s + n \times t_w) + (\frac{n}{c} + P)(t_s + c \times t_w)$$

The optimum block size c_{opt} is obtained applying the derivative to T_P and equal it to zero

$$\frac{\partial T_P}{\partial c} = n \times t_{body} - t_s \frac{n}{c^2} + P \times t_w = 0$$

$$c_{opt} = \sqrt{\frac{n \times t_s}{n \times t_{body} + P \times t_w}} = \sqrt{\frac{t_s}{t_{body} + t_w \frac{P}{n}}}$$

If $n >> P$ then

$$c_{opt} \simeq \sqrt{\frac{t_s}{t_{body}}}$$

Parallelism (PAR)

Understanding parallelism

Eduard Ayguadé, José Ramón Herrero and Gladys Utrera

Computer Architecture Department
Universitat Politècnica de Catalunya

Course 2019/20 (Fall semester)