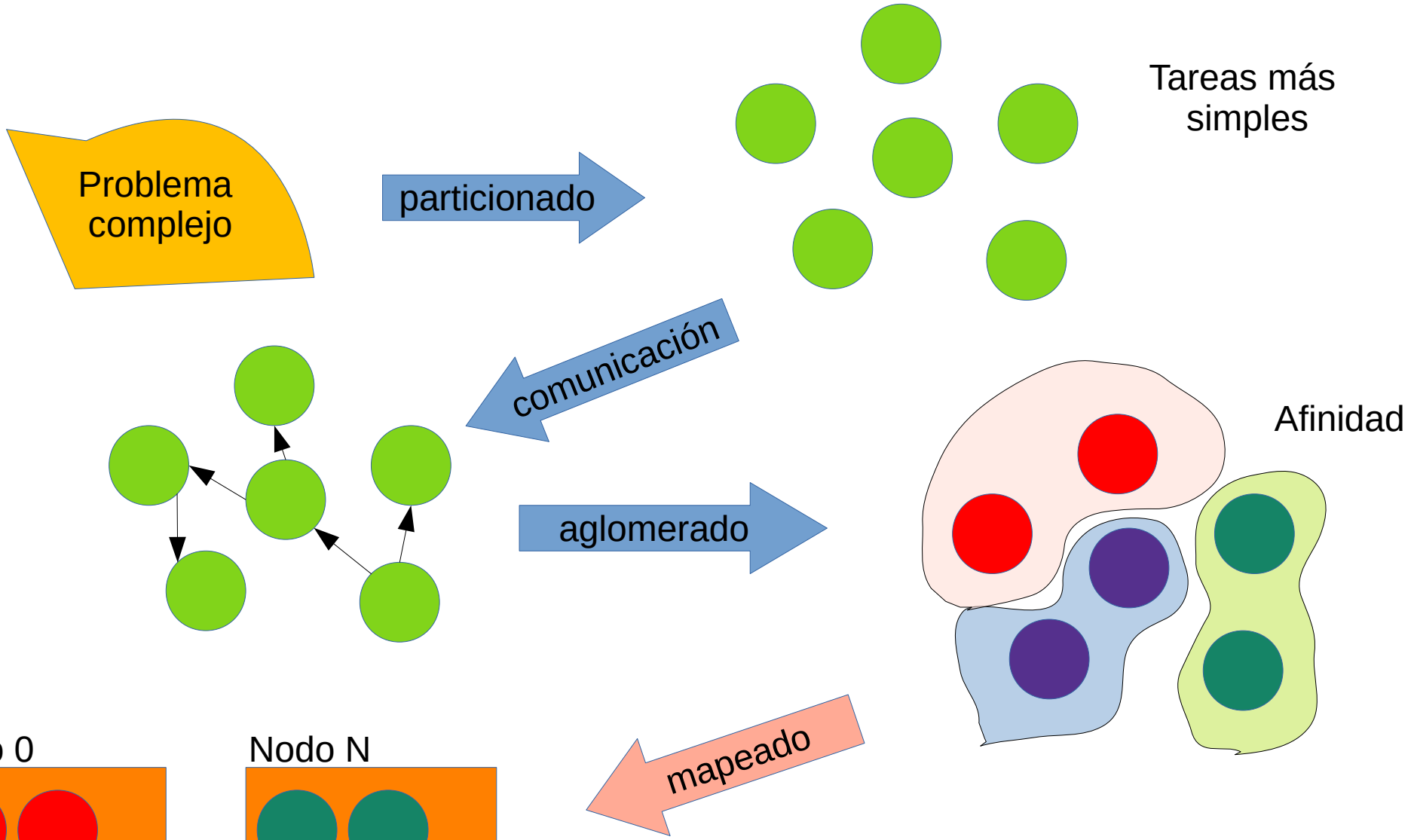
Aspectos avanzados en el uso del clúster UO (II)

Dando poder a los investigadores

Etapas del desarrollo de un programa paralelo

Etapa del desarrollo ...

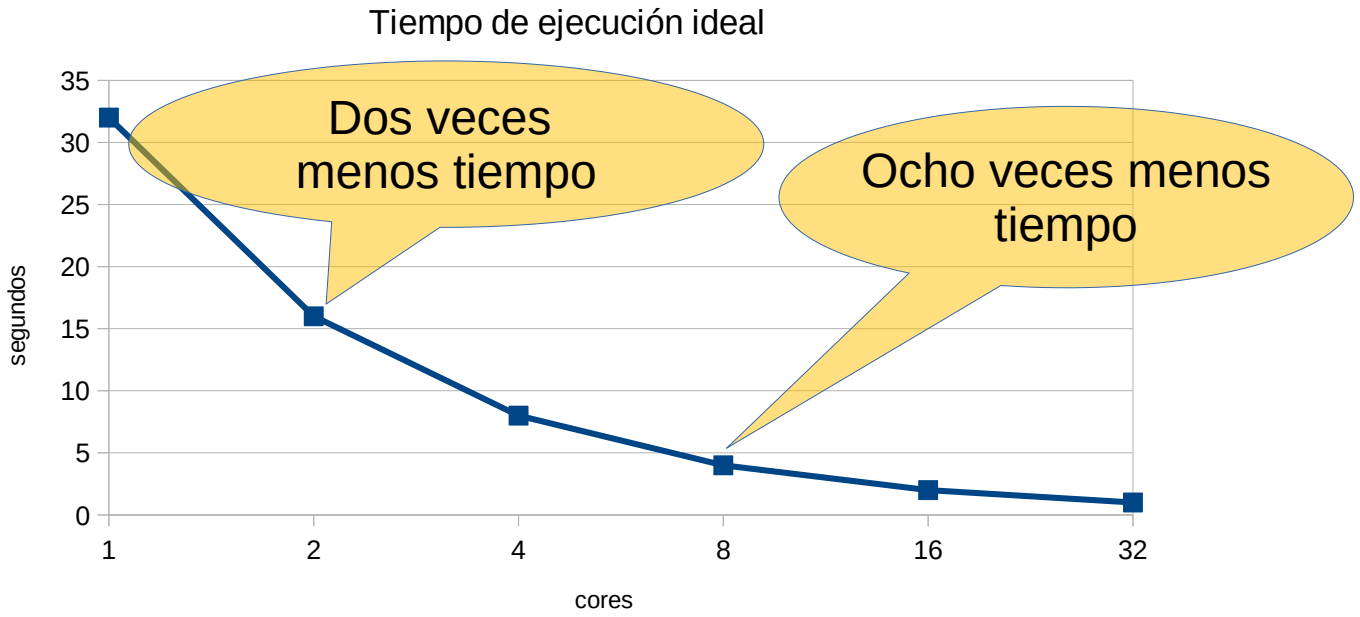


Evaluación de un programa paralelo

Parámetros a medir

- Tiempo de ejecución
  - $ time prog #En Linux
- Aceleración
  - ¿Cuántas veces va más rápido con más cores?
- Eficiencia
  - ¿Cuán eficiente es la paralelización?
  - Independiente del hardware usado

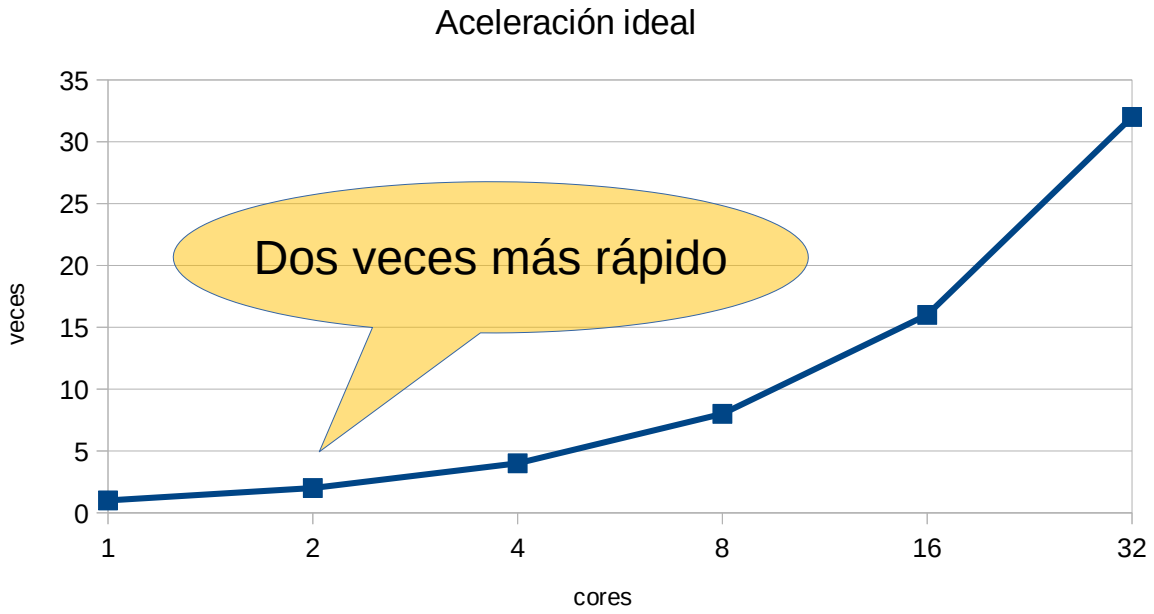
Tiempo de ejecución



$$Tp = tiempo\,medido\,para\,p\,cores$$

Aceleración

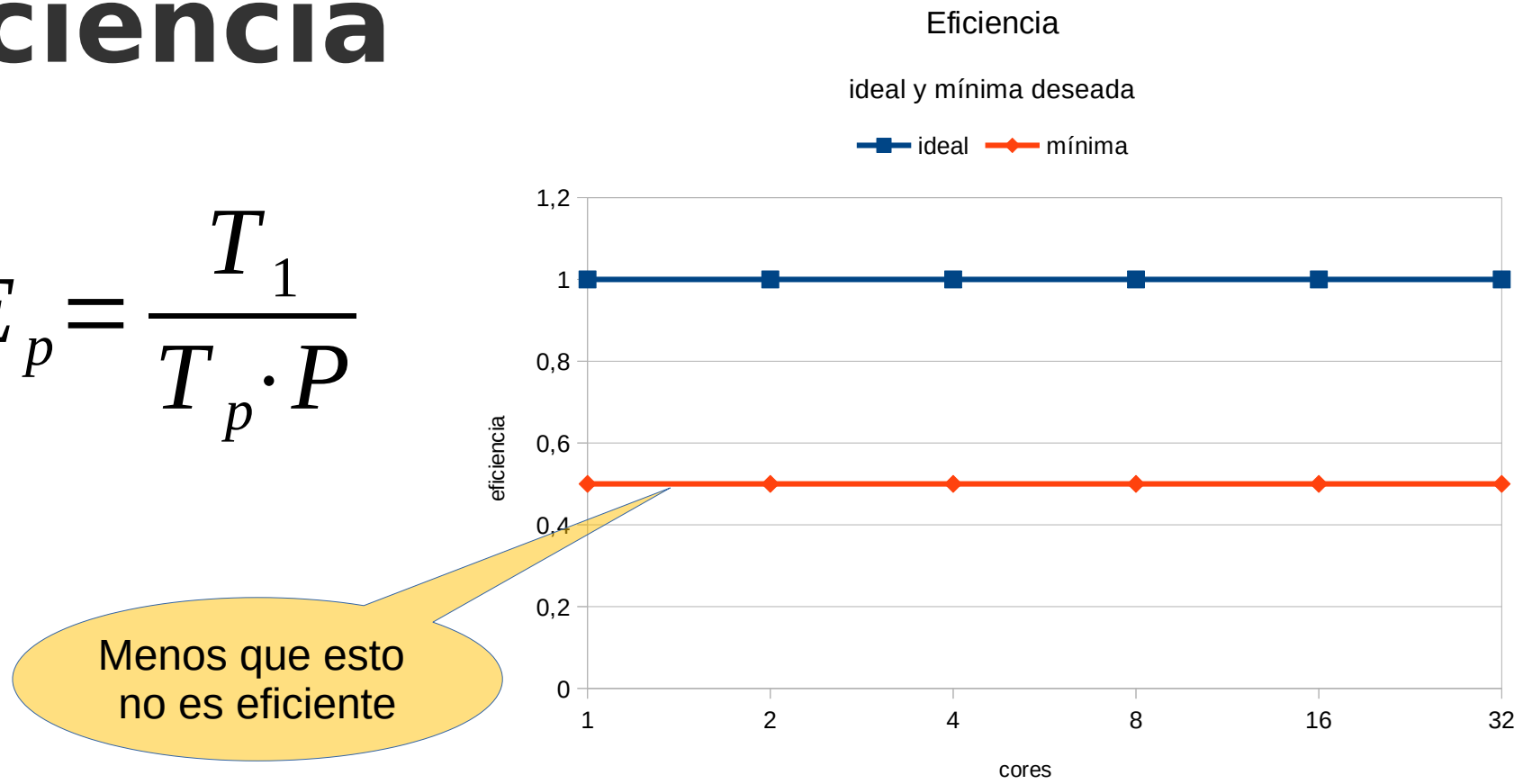
$$Ap = \frac{T_1}{T_p}$$



Ap – Aceleración para p cores
T1 – Tiempo sobre 1 core
Tp – Tiempo sobre p cores

Eficiencia

$$E_p = \frac{T_1}{T_p \cdot P}$$



Ep – Eficiencia para p cores
T1 – Tiempo sobre 1 core
Tp – Tiempo sobre p cores
P – Cores

Ley de Amdahl

$$Ap = \frac{1}{S + \frac{T_p}{P}}$$

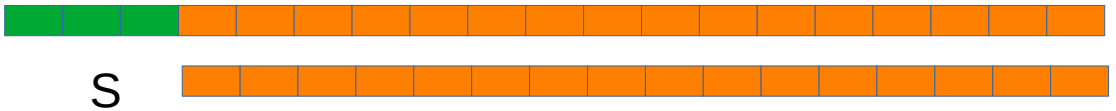
Programa paralelo con
32 unidades paralelas
3 unidades secuenciales

S Tp

Ap – Aceleración para p cores
S – Tiempo de la porción secuencial
Tp – Tiempo paralelo sobre p cores
P – Cores

Ley de Amdahl

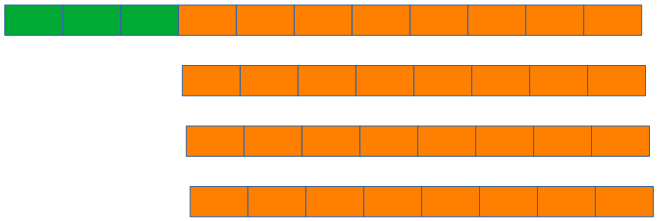
Para 2 procesadores



S

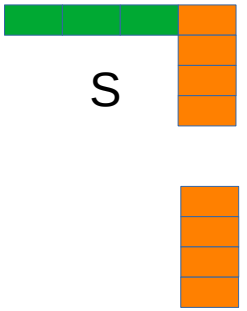
Tp

Para 4 procesadores



Tp

Para 32 procesadores

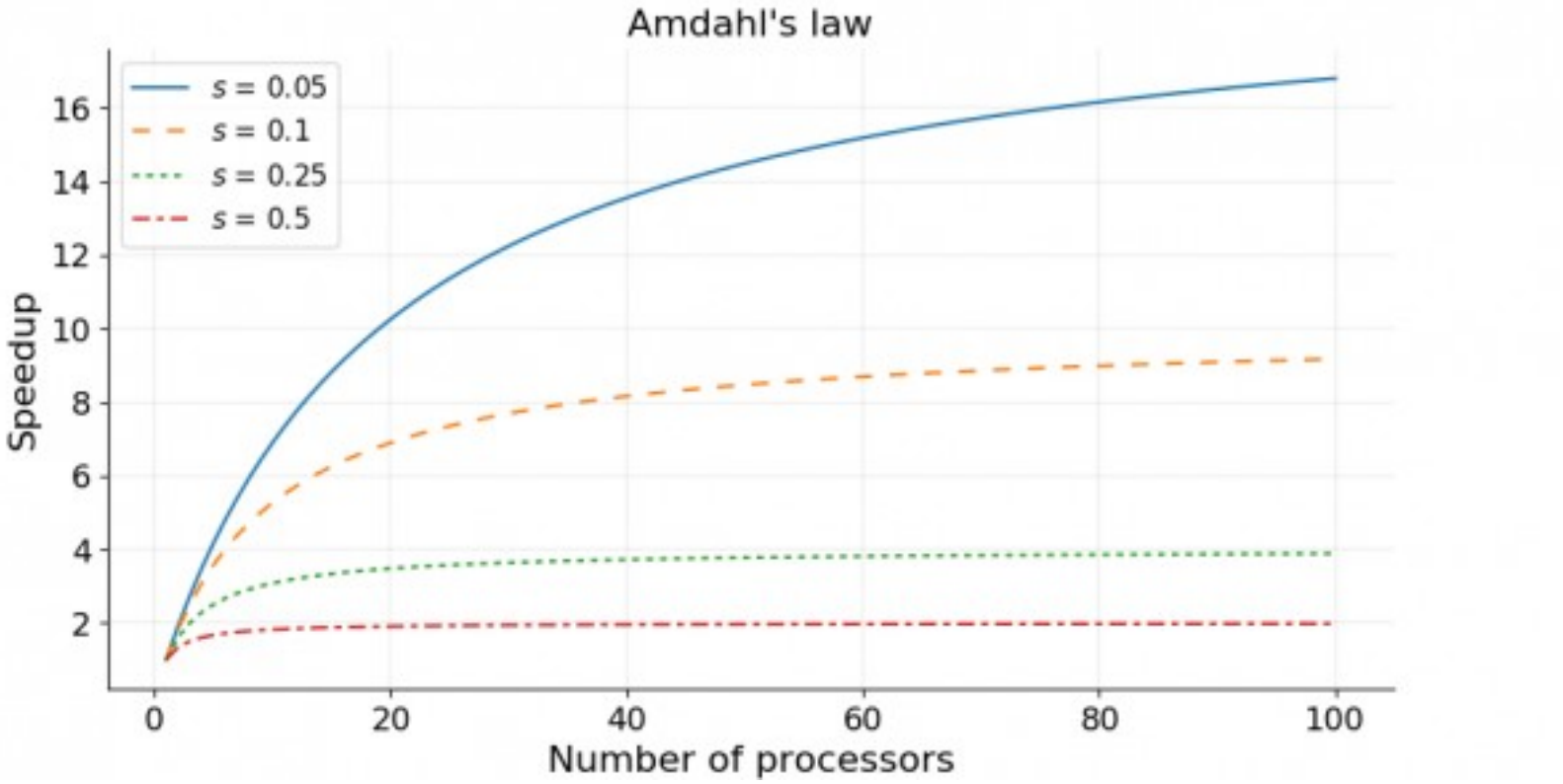


S

Más detallado

- Considere un programa que toma 20 horas sobre un core.
- Una parte que no puede ser paralelizada toma 1 hora en ejecutarse.
- El resto del código, que toma 19 horas, es paralelizado.
- El tiempo mínimo de ejecución nunca será menor de una hora.

Ley de Amdahl



https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/

Ley de Amdahl

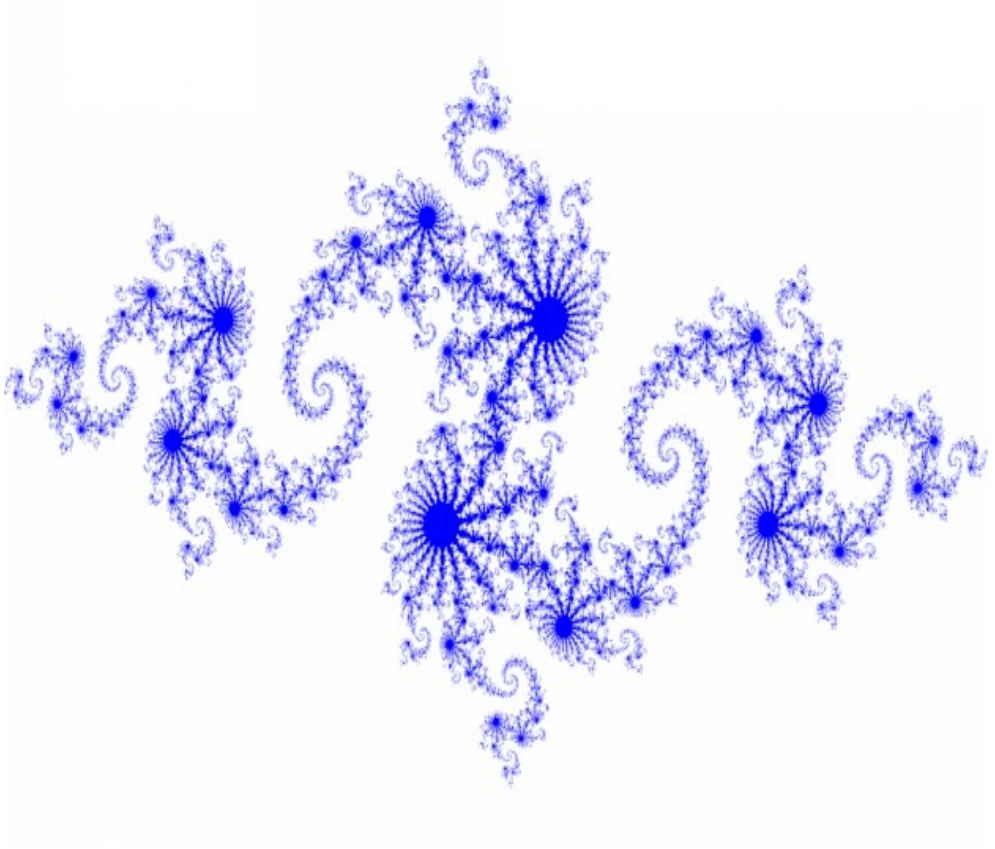
Conclusión de Amdahl

- La computación paralela con muchos procesadores solamente es útil para programas altamente paralelizables.

Implicaciones para HPC-UO

- Para un tamaño de problema determinado existe un número máximo de cores (P) para los que la eficiencia de ejecución se mantiene por encima de 0.5
- ¿Cómo calcular el valor de P?
  - Mediante el cálculo de la eficiencia
  - ¿En cuál valor de P la curva corta a la recta y = 0.5?

Ejemplo práctico
Imagen del conjunto de Julia



Código parcial ...

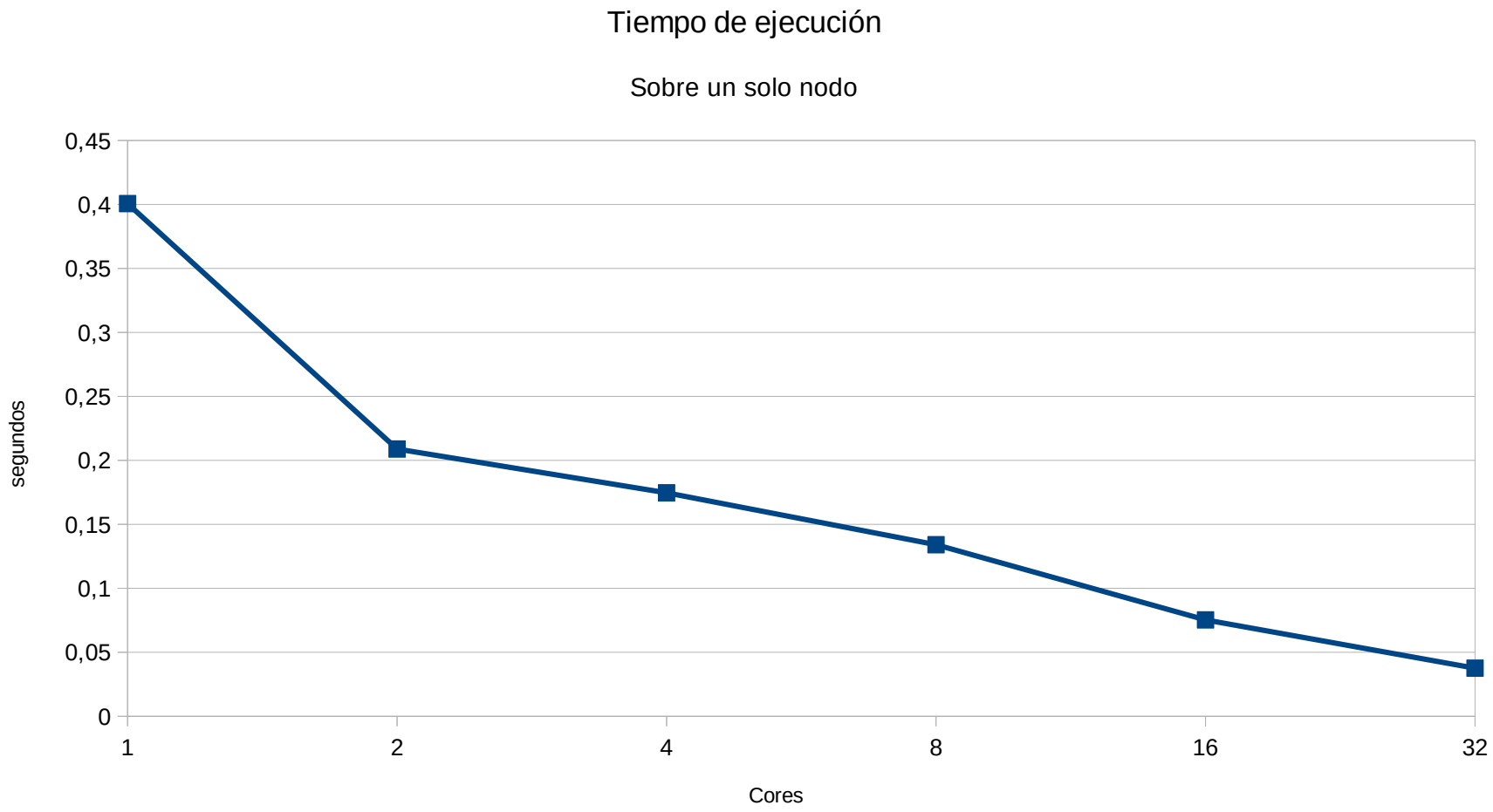
```
# pragma omp parallel for schedule(dynamic) shared ( h, w, xl, xr, yb, yt )
private ( i, j, k, juliaValue )
  for ( j = 0; j < h; j++ )
  {
    ...
  }
int main ( int argc, char* argv[] )
{
  if ( argc < 3 )
  {
    printf ( "Usage: ./julia_openmp height width\n" );
    return 0;
  }
  int h = atoi(argv[1]);
  int w = atoi(argv[2]);
  ...
}
```

Tiempo de ejecución

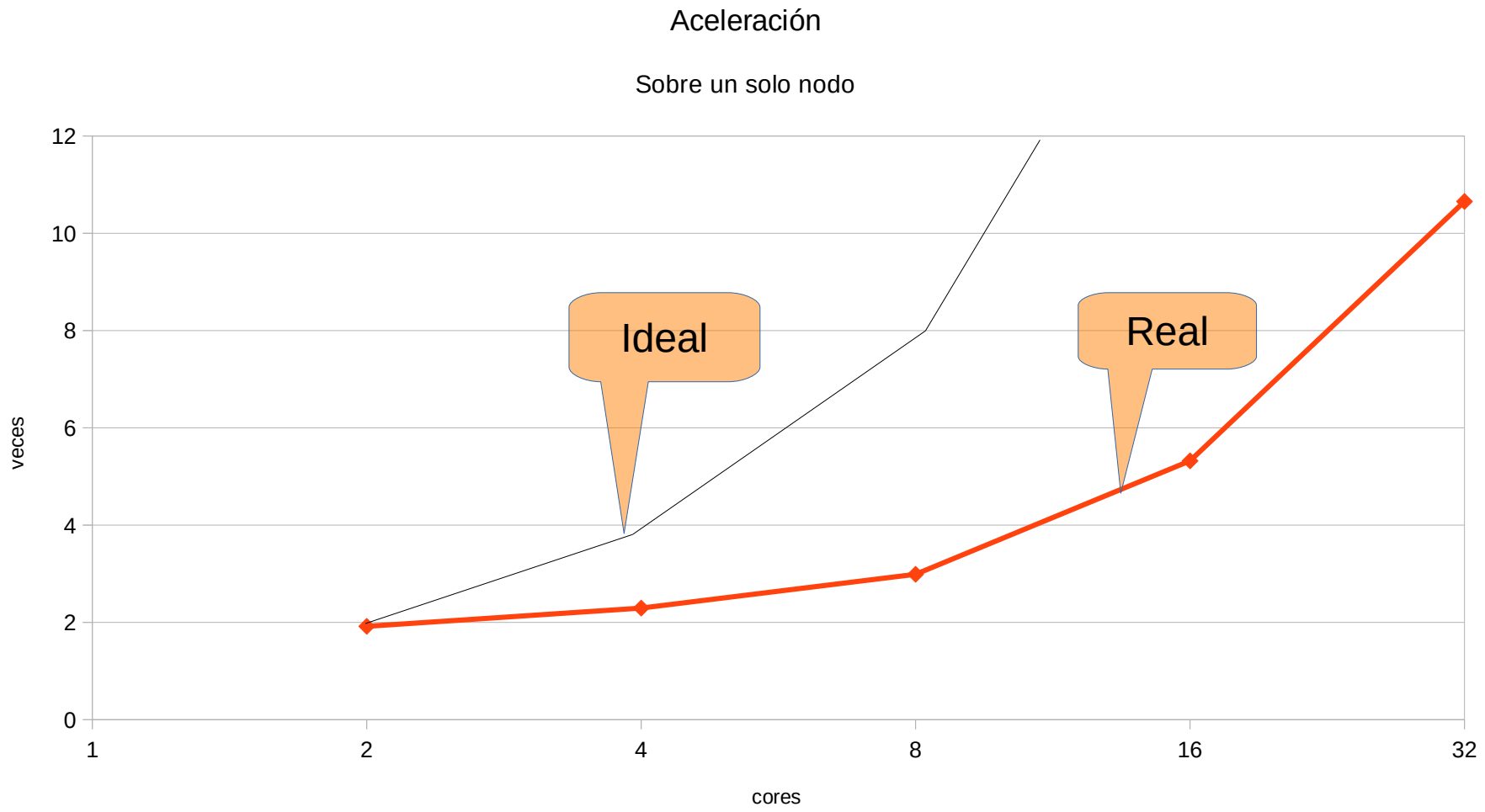
Ancho = Largo = 240,000 pixels

Hilos	Tiempo (s)
1	0.400652
2	0.208863
4	0.174637
8	0.13406
16	0.0752996
32	0.0376086

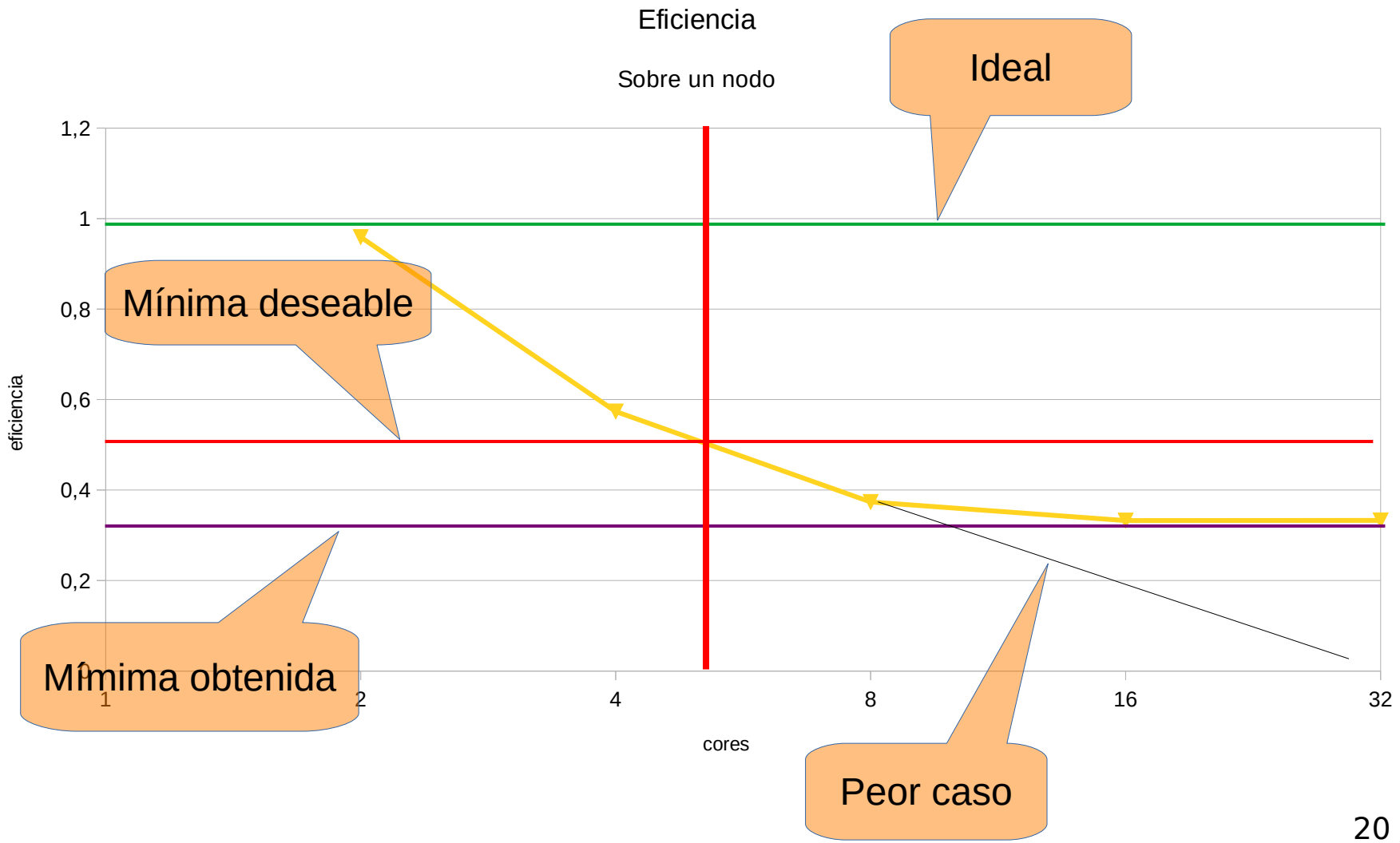
Análisis de escalabilidad



Análisis de escalabilidad



Análisis de escalabilidad



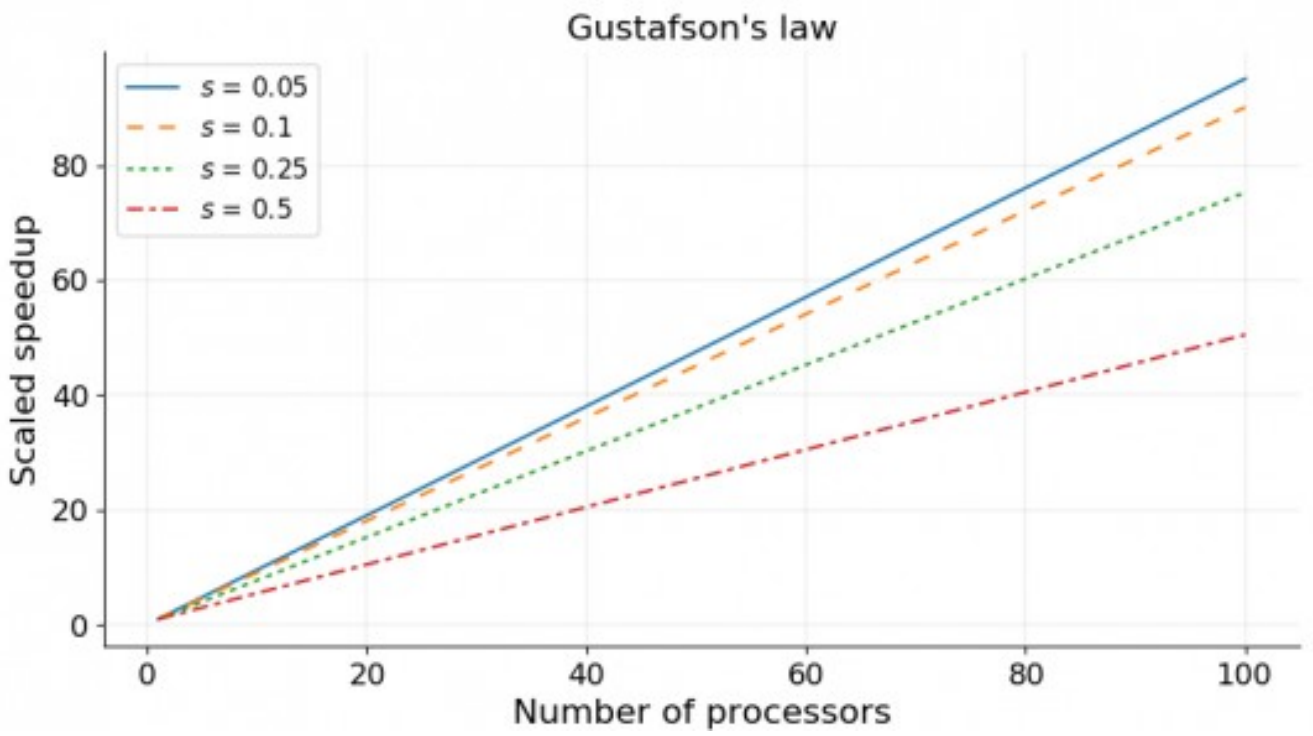
¿Conclusión?

Ley de Gustafson

Ley de Gustafson

$$Ap = S + T_p . P$$

Escalado débil



Ley de Gustafson

- La aceleración escala linealmente con el número de cores usado.
- No hay límite superior a la aceleración.
- La aceleración se calcula en base al trabajo realizado para un problema escalado.
- Para un número mayor de cores se aborda un problema de mayor tamaño.

Escalar el problema

Número de nodos y/o cores

10 — 100 — 1000

Problema pequeño

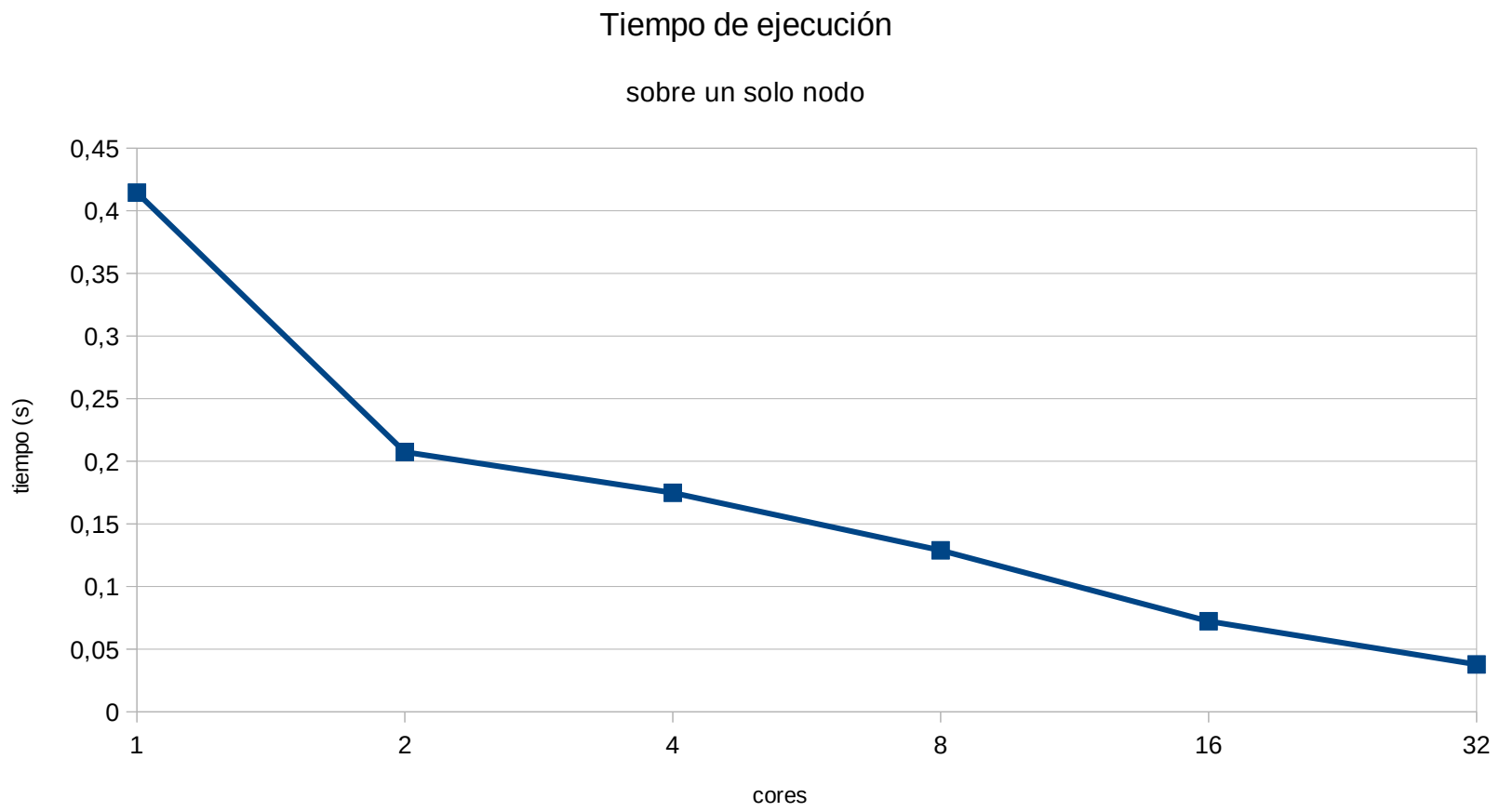
Problema mediano

Problema grande

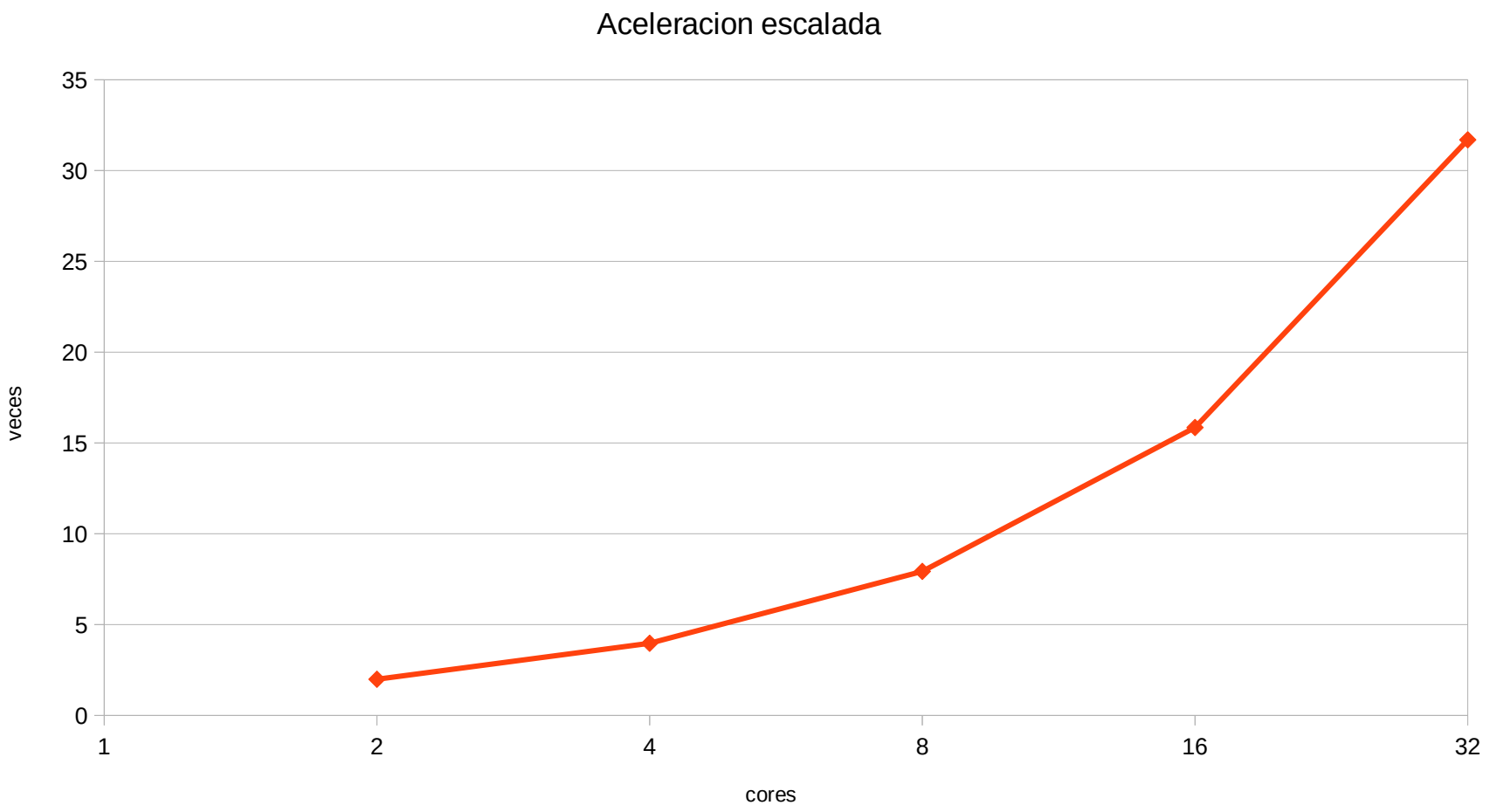
Ejemplo anterior escalado

Tamaño	Cores	Tiempo (s)
10,000	1	0.414513
20,000	2	0.2073996
40,000	4	0.174805
80,000	8	0.12888
160,000	16	0.0722656
320,000	32	0.0377899

Escalado débil



Escalado débil



$$A_p = P - \partial \cdot (P - 1)$$

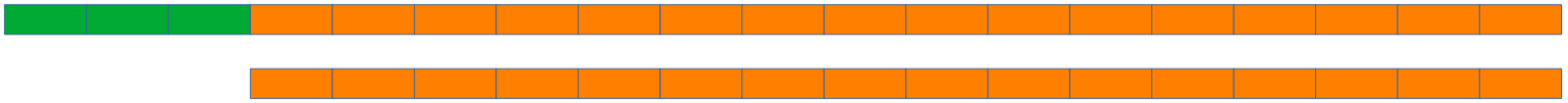
Escalado débil

Para 1 procesador

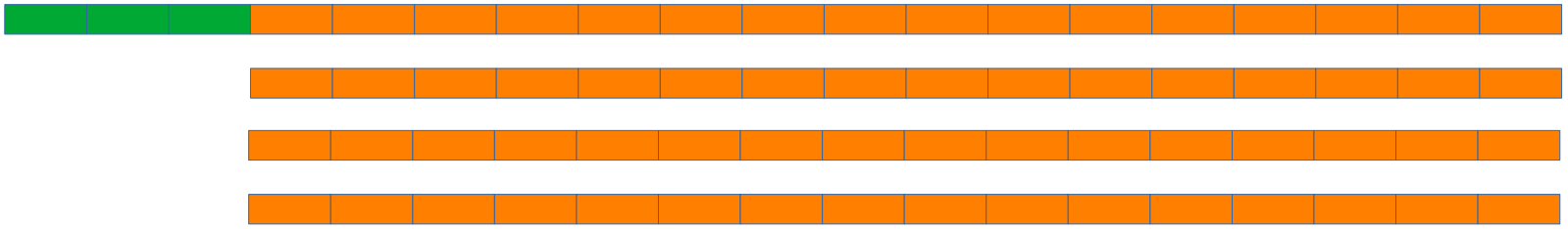
S

Tp

Para 2 procesadores



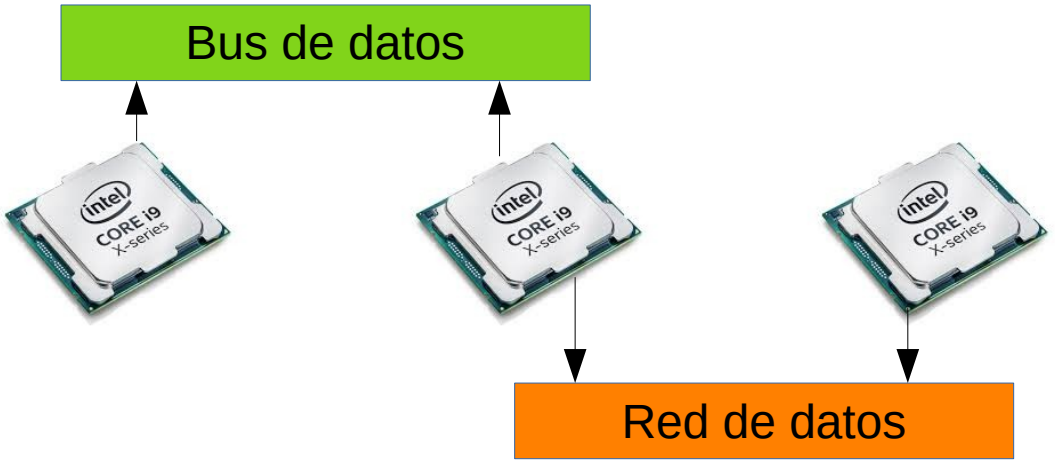
Para 4 procesadores



Efecto de las comunicaciones sobre el tiempo de ejecución

Tiempo de ejecución

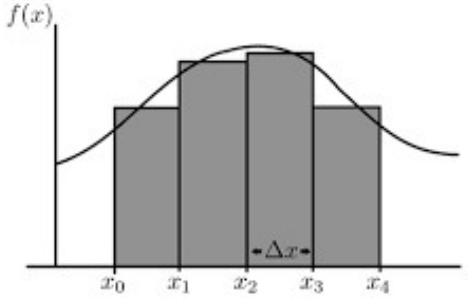
$$T_t = T_s + T_p + T_c$$

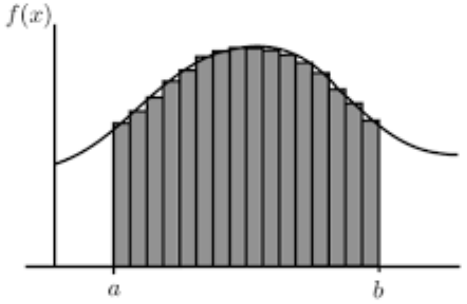


Tt = Tiempo total de ejecución
Ts = Tiempo de la porción secuencial
Tp = Tiempo de procesamiento en paralelo
Tc = Tiempo de las comunicaciones

Ejemplo de problema

Cálculo del área bajo la curva





y = 1.0 / (sin(x) + 2.0) + 1.0 / (sin(x)*cos(x) + 2.0);

Intervalo: 1 – 5 000 000
N = 100 000

Memoria compartida vs Memoria distribuida

6 tareas distribuidas de maneras distintas

Nodos	Cores	Tiempo (ms)	Esquema
1	1	2065.838	Secuencial
1	6	353.131	Compartida
6	1	382.767	Distribuida

30 tareas distribuidas de maneras distintas

1	30	73.220	Compartida
3	10	129.808	Híbrida
5	6	113.489	Híbrida
6	5	139.810	Híbrida
30	1	?	Distribuida

Efecto de la red

Nodos	Cores (++16)	Tiempo(ms)
1	1	2180.593
1	16	142.509
1	32	67.936
2	48	68.121

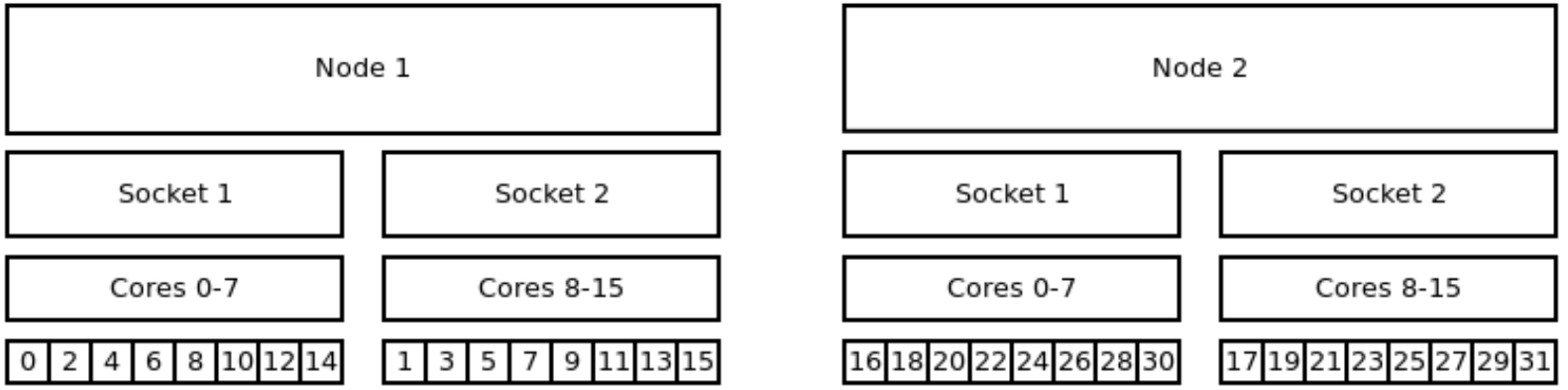
Nodos	Cores (++1)	Tiempo(ms)
1	31	74.219
1	32	67.936
2	33	80.932

Distribución de las tareas sobre los nodos (mapeado)

Ejemplos de scripts

DEFAULT BINDING AND DSITRIBUTION PATTERN

The default binding uses --cpu_bind=cores in combination with --distribution=block:cyclic. The default (as well as block:cyclic) allocation method will fill up one node each second socket of each node.




```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --tasks-per-node=16
#SBATCH --cpus-per-task=1

srun --ntasks 32 ./application
```

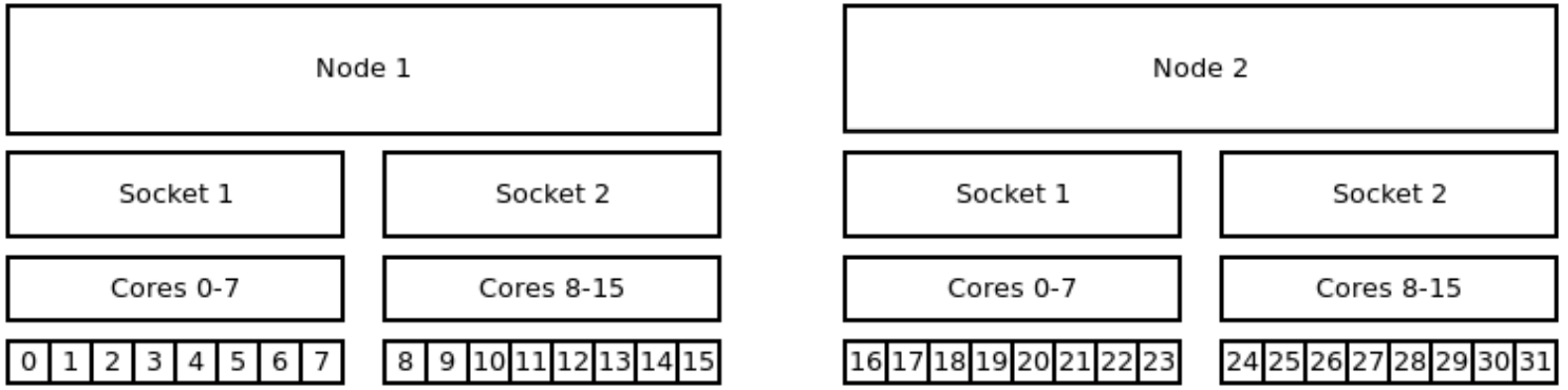
Ejemplos de scripts

CORE BOUND

Note: With this command the tasks will be bound to a core for the entire runtime of your application.

Distribution: block:block

This method allocates the tasks linearly to the cores.

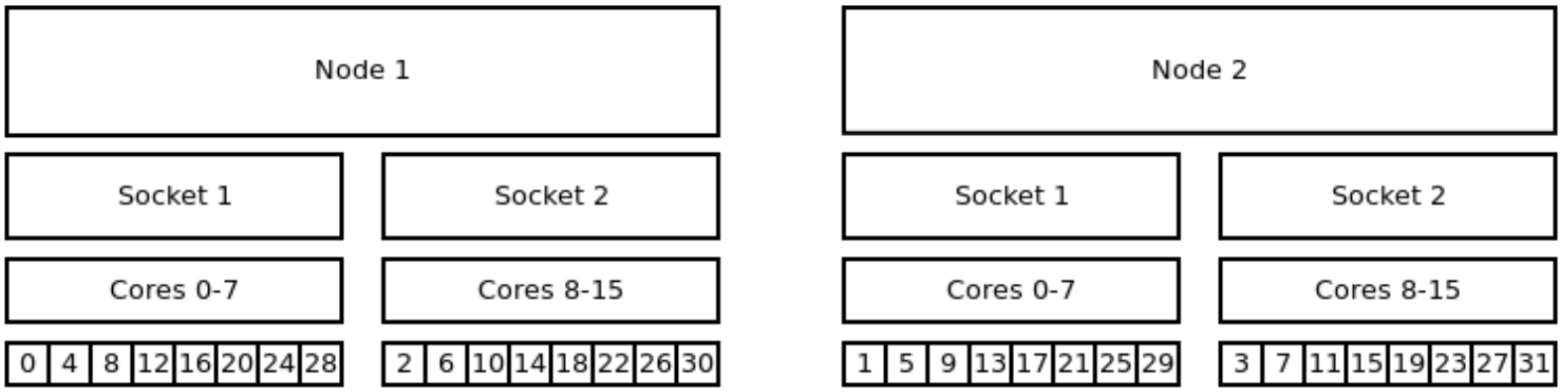



```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --tasks-per-node=16
#SBATCH --cpus-per-task=1

srun --ntasks 32 --cpu_bind=cores --distribution=block:block ./application
```

Ejemplos de scripts

Distribution: cyclic:cyclic

--distribution=cyclic:cyclic will allocate your tasks to the cores in a round robin approach. It starts with the first socket of the first node, then the first socket of the s
every node and so on.

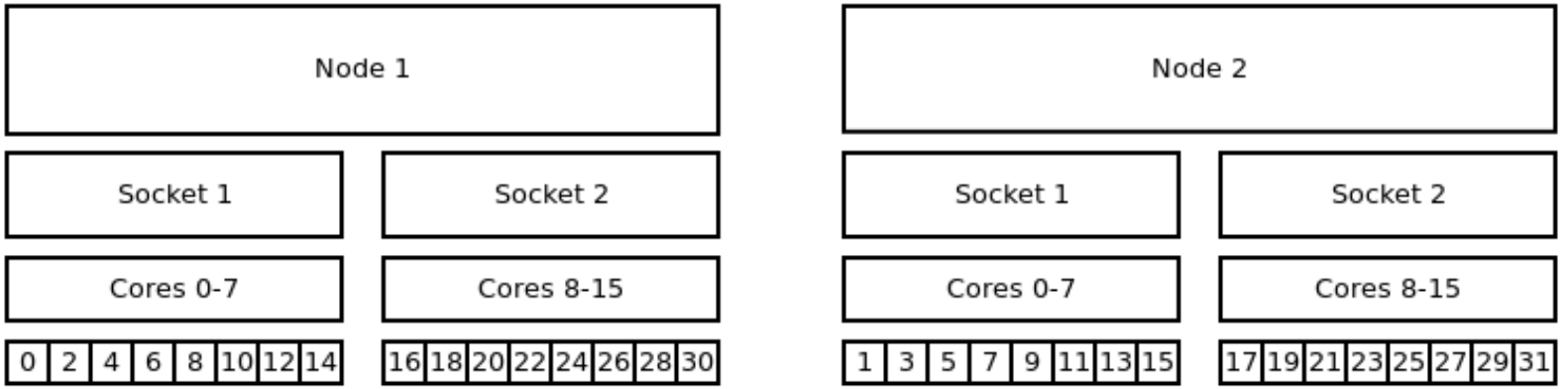



```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --tasks-per-node=16
#SBATCH --cpus-per-task=1

srun --ntasks 32 --cpu_bind=cores --distribution=cyclic:cyclic
```

Ejemplos de scripts

Distribution: cyclic:block

The cyclic:block distribution will allocate the tasks of your job in alternation on node level, starting with first node filling the sockets linearly.




```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --tasks-per-node=16
#SBATCH --cpus-per-task=1

srun --ntasks 32 --cpu_bind=cores --distribution=clyclic:block ./application
```

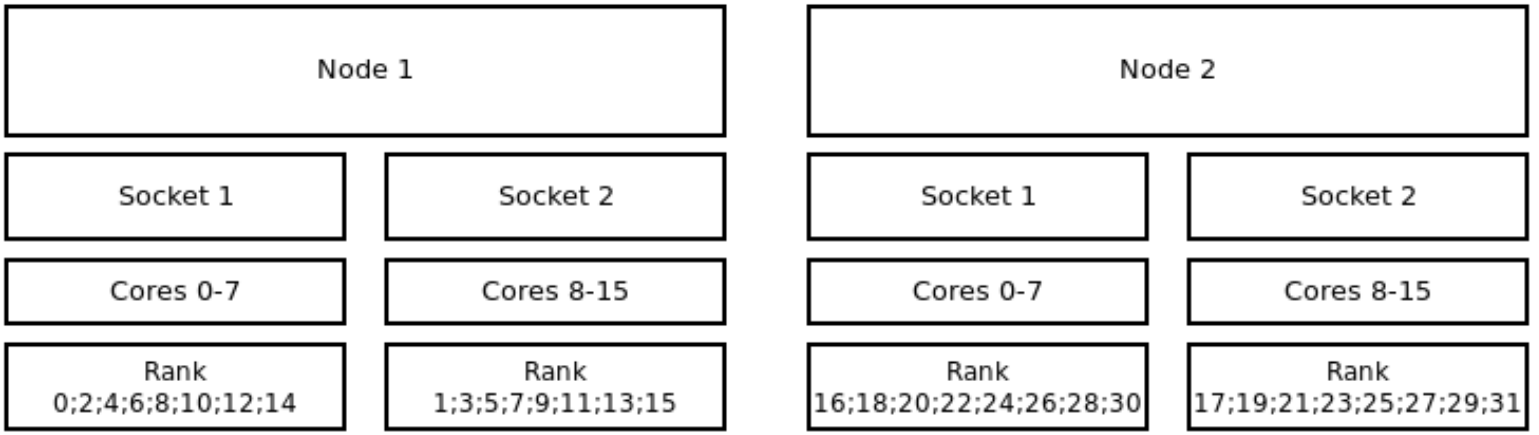
Ejemplos de scripts

SOCKET BOUND

Note: The general distribution onto the nodes and sockets stays the same. The mayor difference between socket and cpu bound lies within the ability of the ta
your application.

Default Distribution

The default distribution uses --cpu_bind=sockets with --distribution=block:cyclic. The default allocation method (as well as block:cyclic) will fill up one node aft
second socket of each node.

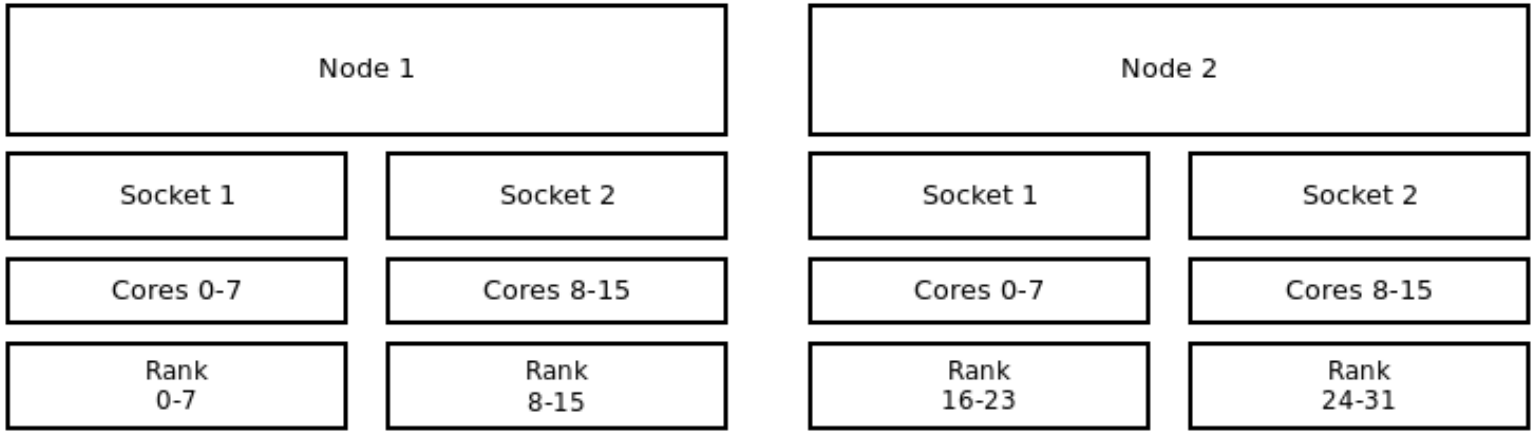



```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --tasks-per-node=16
#SBATCH --cpus-per-task=1

srun --ntasks 32 -cpu_bind=sockets ./application
```

Ejemplos de scripts

Distribution: block:block

This method allocates the tasks linearly to the cores.

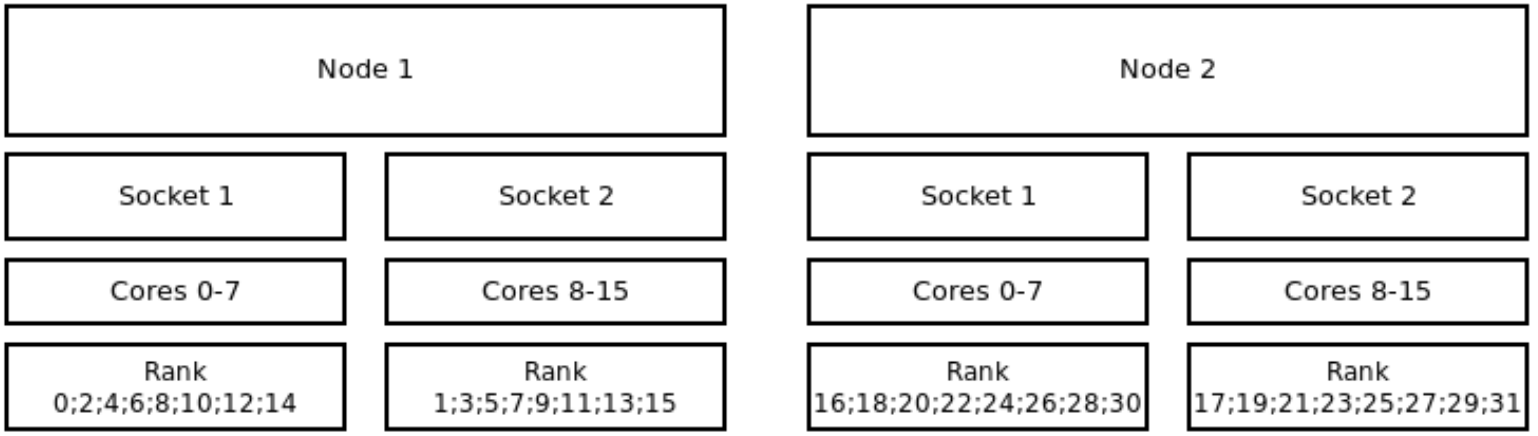



```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --tasks-per-node=16
#SBATCH --cpus-per-task=1

srun --ntasks 32 --cpu_bind=sockets --distribution=block:block ./application
```

Ejemplos de scripts

Distribution: cyclic:block

The cyclic:block distribution will allocate the tasks of your job in alternation between the first node and the second node while filling the sockets linearly.



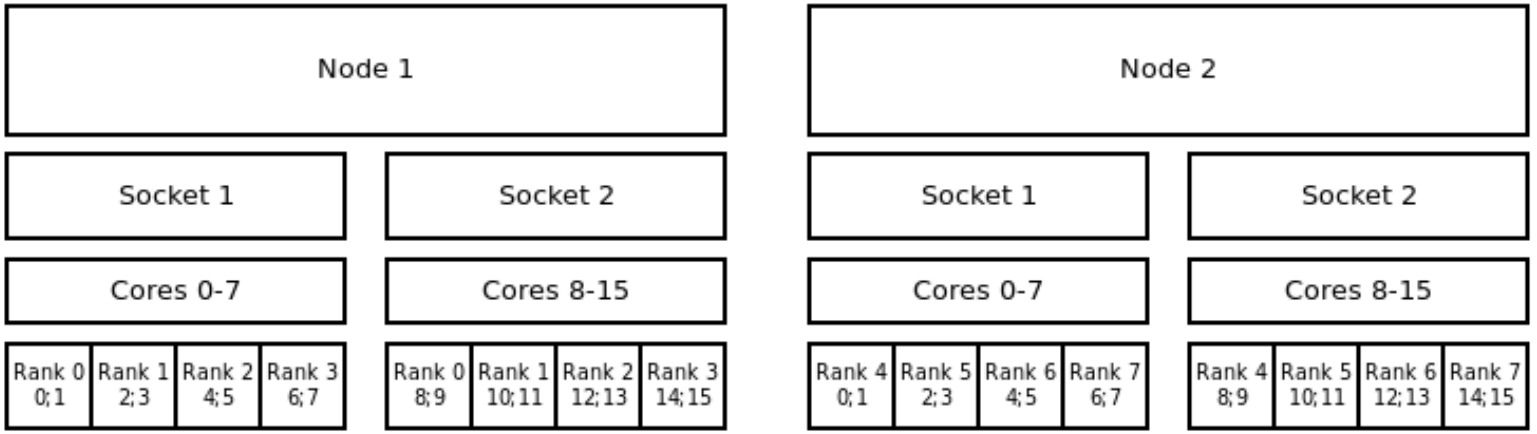

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCh --tasks-per-node=16
#SBATCH --cpus-per-task=1

srun --ntasks 32 --cpu_bind=sockets --distribution=cyclic:block ./application
```

Ejemplos de scripts

HYBRID STRATEGIES

DEFAULT BINDING AND DISTRIBUTION PATTERN

The default binding pattern of hybrid jobs will split the cores allocated to a rank between the sockets of a node. The example shows that Rank 0 has 4 cores



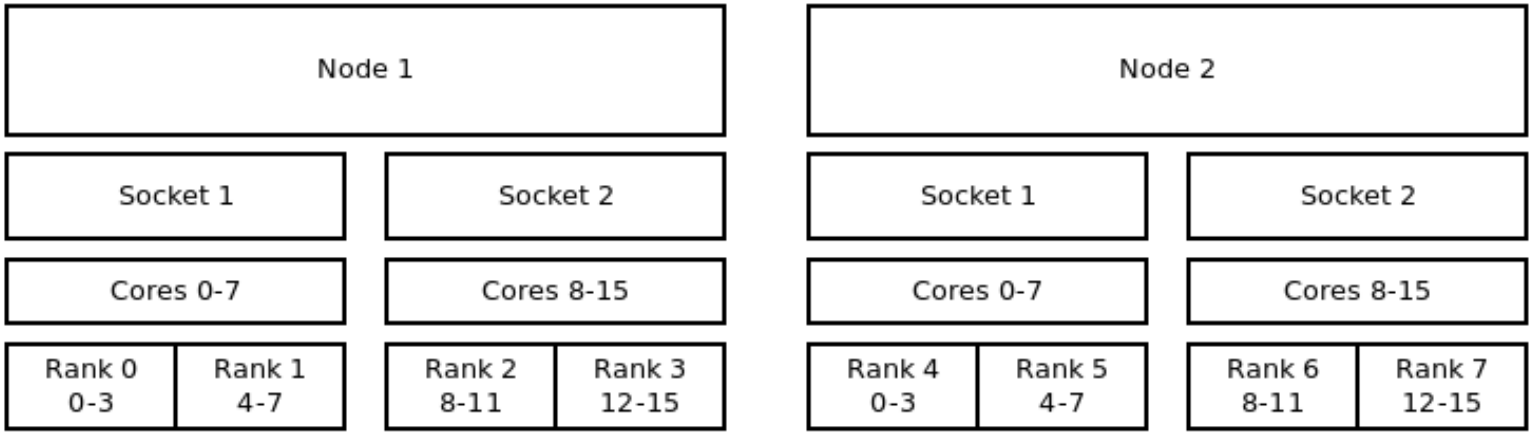

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --tasks-per-node=4
#SBATCH --cpus-per-task=4

export OMP_NUM_THREADS=4

srun --ntasks 8 --cpus-per-task $OMP_NUM_THREADS ./application
```

Ejemplos de scripts

CORE BOUND

Distribution: block:block

This method allocates the tasks linearly to the cores.

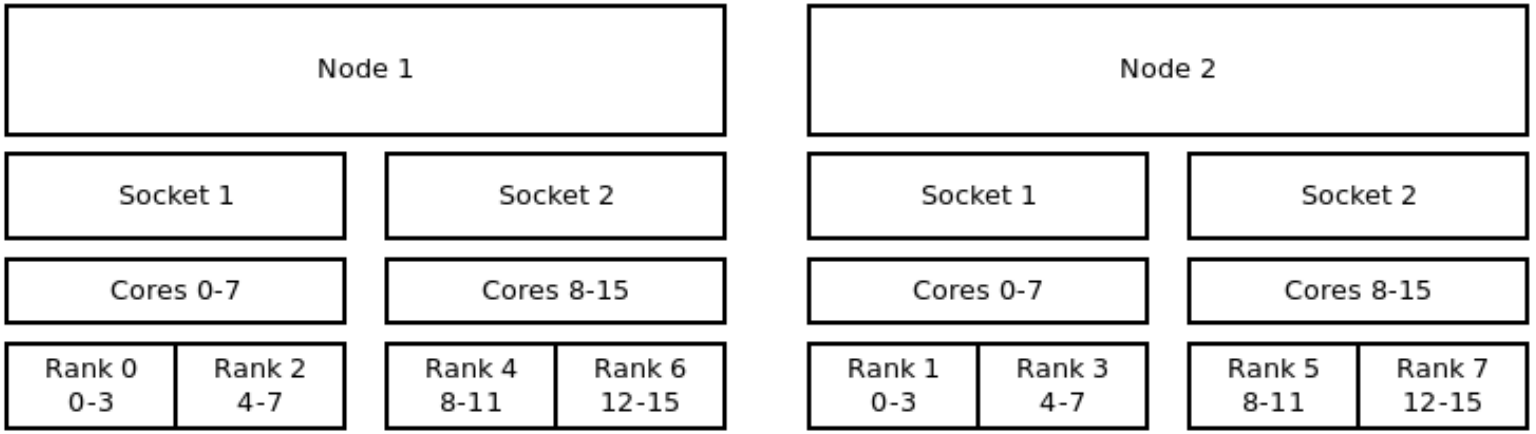



```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --tasks-per-node=4
#SBATCH --cpus-per-task=4

export OMP_NUM_THREADS=4

srun --ntasks 8 --cpus-per-task $OMP_NUM_THREADS --cpu_bind=cores --distribution=block:block ./application
```

Ejemplos de scripts

Distribution: cyclic:block

The cyclic:block distribution will allocate the tasks of your job in alternation between the first node and the second node while filling the sockets linearly.




```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --tasks-per-node=4
#SBATCH --cpus-per-task=4

export OMP_NUM_THREADS=4

srun --ntasks 8 --cpus-per-task $OMP_NUM_THREADS --cpu_bind=cores --distribution=cyclic:block ./application
```

Distribución arbitraria de tareas

- srun -n5 -m arbitrary -w nodo[0,0,0,0,1] hostname

  0, 1, 2, 3, 4

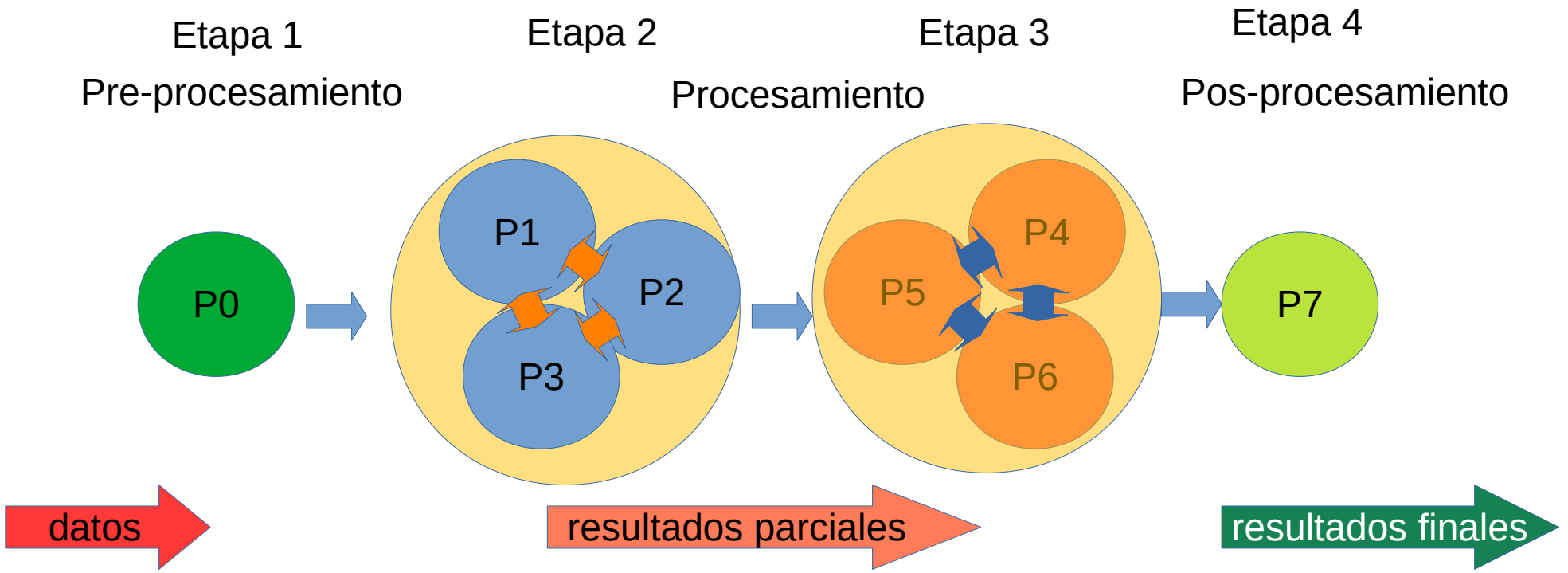
  (tareas 0-3 sobre primer nodo, la 4 sobre segundo nodo)

- srun -n5 -m arbitrary -w nodo[0,0,1,0,0] hostname

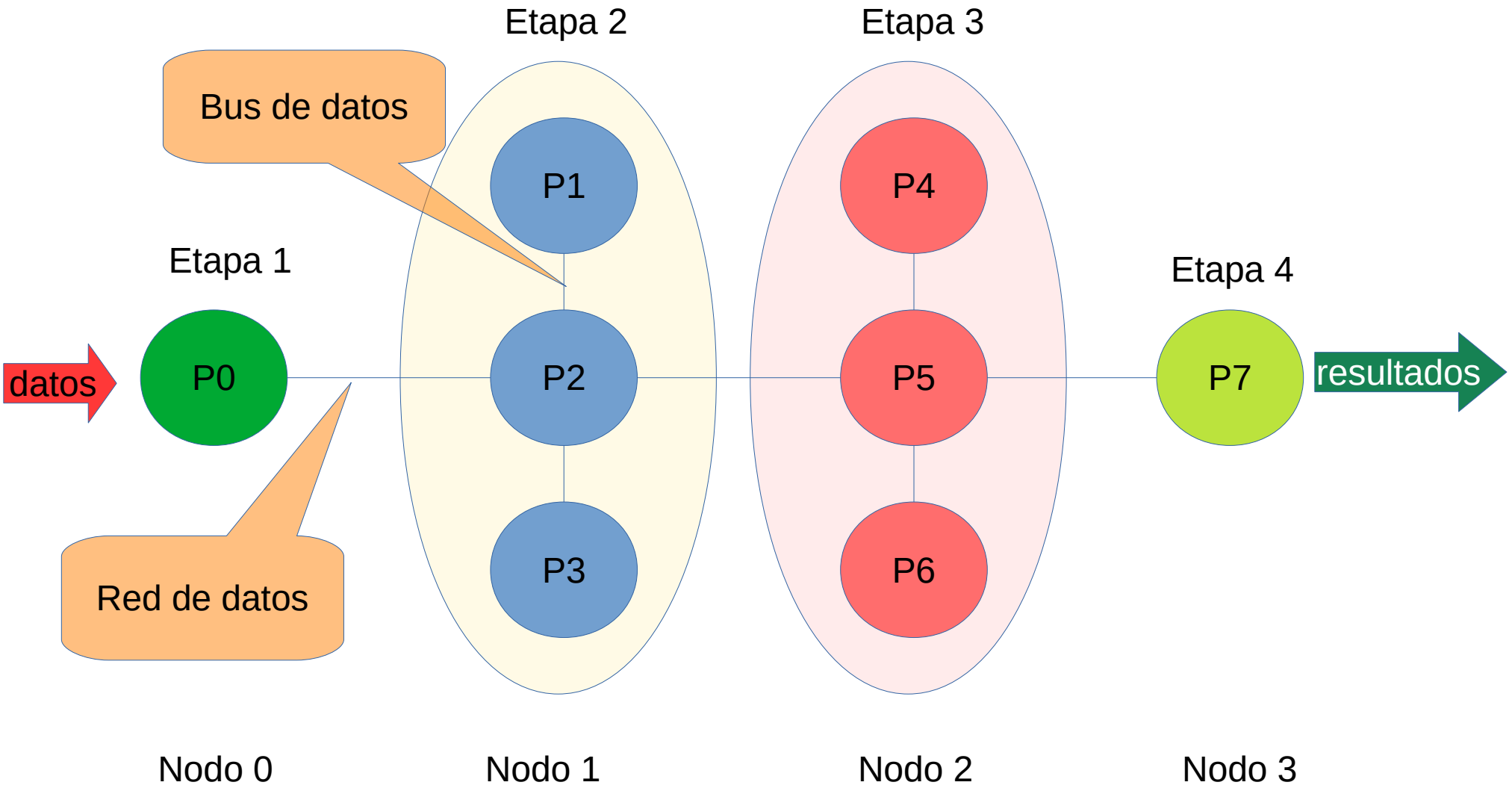
  0, 1, 2, 3, 4

  (tarea 2 sobre segundo nodo)

Ejemplo de una tubería

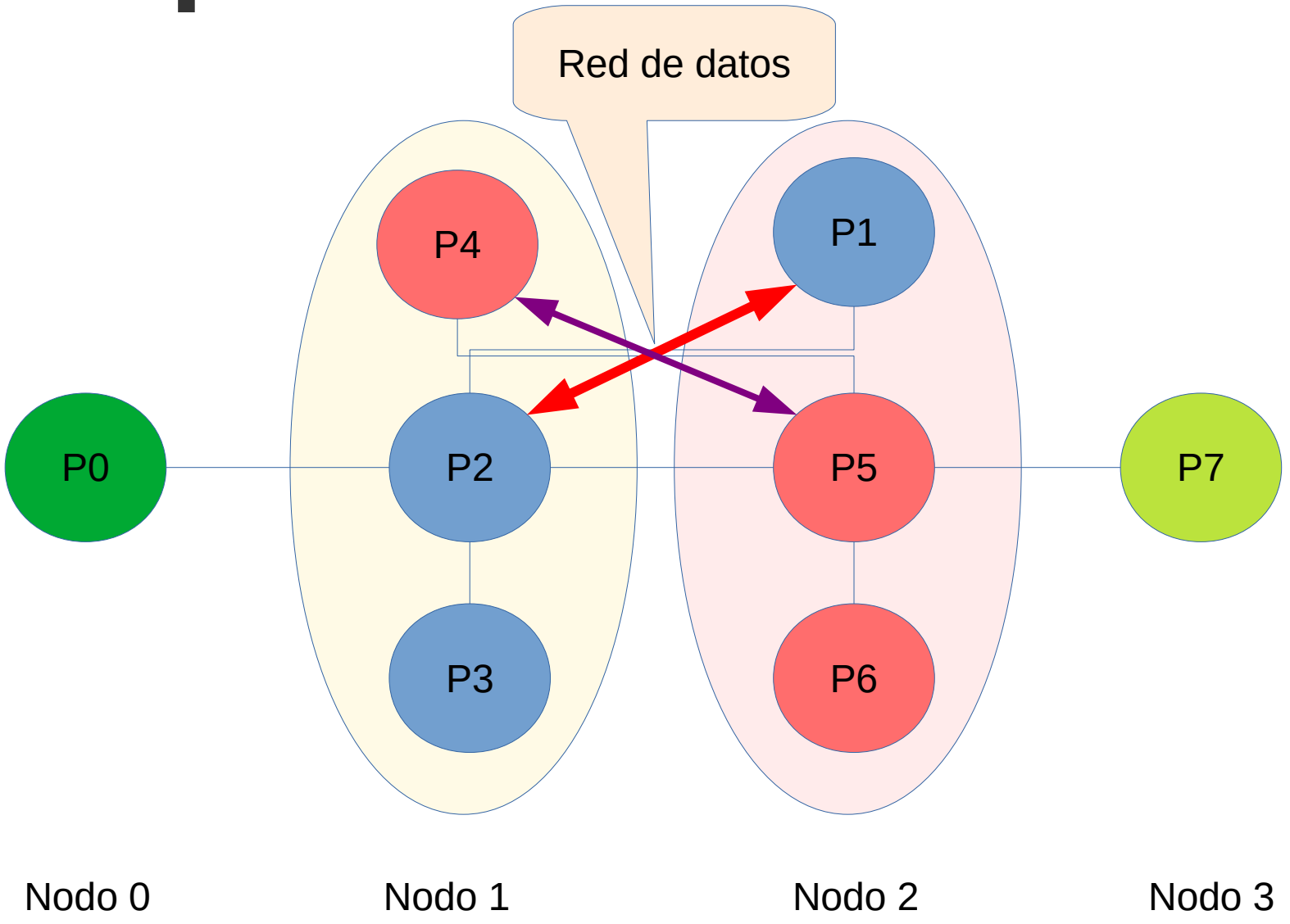


Ejemplo de una tubería



nodo[0,1,1,1,2,2,2,3]

Ejemplo de una tubería



Resumen

- Para un problema de tamaño fijo existe un máximo número de procesadores que tiene sentido usar. (ley de Amdahl)
- Si dispongo de un número mayor de procesadores debo aprovecharlos para resolver un problema de mayor tamaño. (ley de Gustafson)
- La manera en que distribuya los procesos sobre los nodos influye en el tiempo de ejecución. (mapeado)

Gracias
¿Preguntas?

Aspectos avanzados en el uso del clúster UO (II)

Dando poder a los investigadores