Parallelism and Performance

Ideal parallel world:

- Sequential runs in T_s
- P processors run in $T_p = \frac{T_s}{P}$

Today: Why that usually does't happen

- **Measuring Performance**
- **Obstacle: Non-Parallelism**
- **Obstacle: Overhead**

Measuring Performance

Latency: time to complete a task

This is normally what we want to reduce through parallelism

Measuring Performance

Speedup: ratio of latencies = $\frac{T_s}{T_p}$

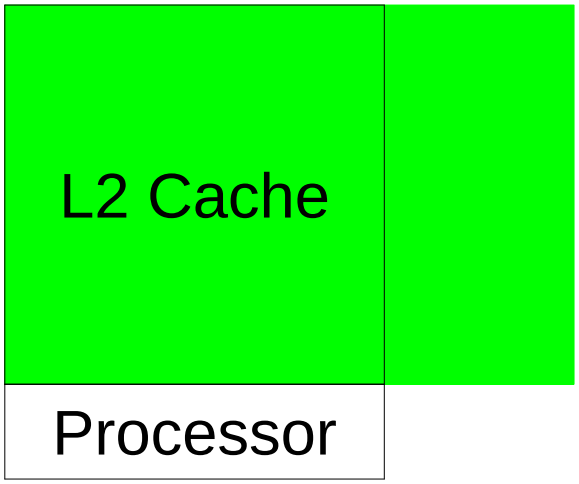
- ***Linear speedup***: speedup approximates *P*
- ***Sublinear speedup***: speedup less than *P*
- ***Superlinear speedup***: speedup more than *P* **!**

Superlinear speedup happens when the algorithm or machine changes

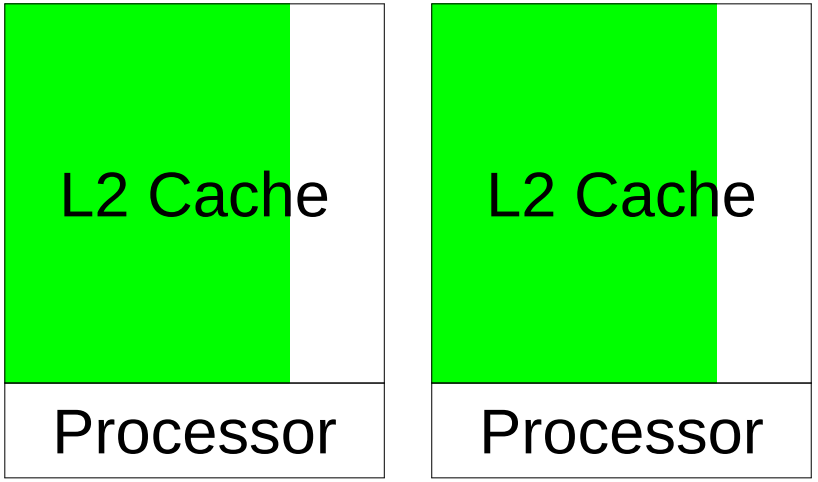
Superlinear Speedup

Machine change:

Sequential



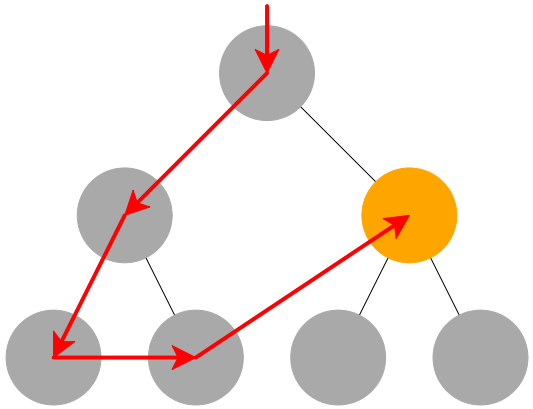
Parallel



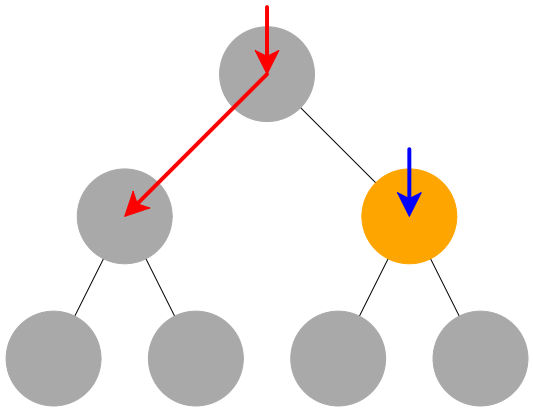
Superlinear Speedup

Algorithm change:

Sequential



Parallel



Measuring Performance

Throughput: $\frac{work}{T_p}$

Higher throughput doesn't imply lower latency

Measuring Performance

Efficiency: effective use of processors = $\frac{Speedup}{P}$

Measuring Performance

FLOPS: floating-point operations per second

IOPS: integer operations per second

Measuring Performance

Performance measurement **don't**s:

- use different machines
- disable compiler optimizations
- equate “sequential” with a single parallel process
- ignore cold start
- ignore devices

Do measure multiple *P* and multiple problem sizes

- Measuring Performance
- **Obstacle: Non-Parallelism**
- Obstacle: Overhead

Inherent Non-Parallelism

Amdahl's Law

$\frac{1}{S}$ of program is inherently sequential $\Rightarrow$

Speedup < *S*

- 50% sequential $\Rightarrow$ maximum speedup of 2
- 90% sequential $\Rightarrow$ maximum speedup of 1.1
- 10% sequential $\Rightarrow$ maximum speedup of 10

and yet lots of processors help for some computations, because it's easy and useful to scale the problem size

Dependencies

Flow Dependence: write followed by read

```
sum = a+1; /* << */
first_term = sum*scale1; /* << */
sum = sum+b;
second_term = sum*scale2;
```

This is a ***true dependence***

Dependencies

Anti Dependence: read followed by write

```
sum = a+1;
first_term = sum*scale1; /* << */
sum=b+1; /* << */
second_term=sum*scale2;
```

This is a ***false dependence***

Rewrite:

```
sum = a+1;
first_term = sum*scale1;
sum2 = b+1;
second_term = sum2*scale2;
```

Dependencies

Output Dependence: write followed by write

```
sum = a+1; /* << */
first_term = sum*scale1;
sum=b+1; /* << */
second_term=sum*scale2;
```

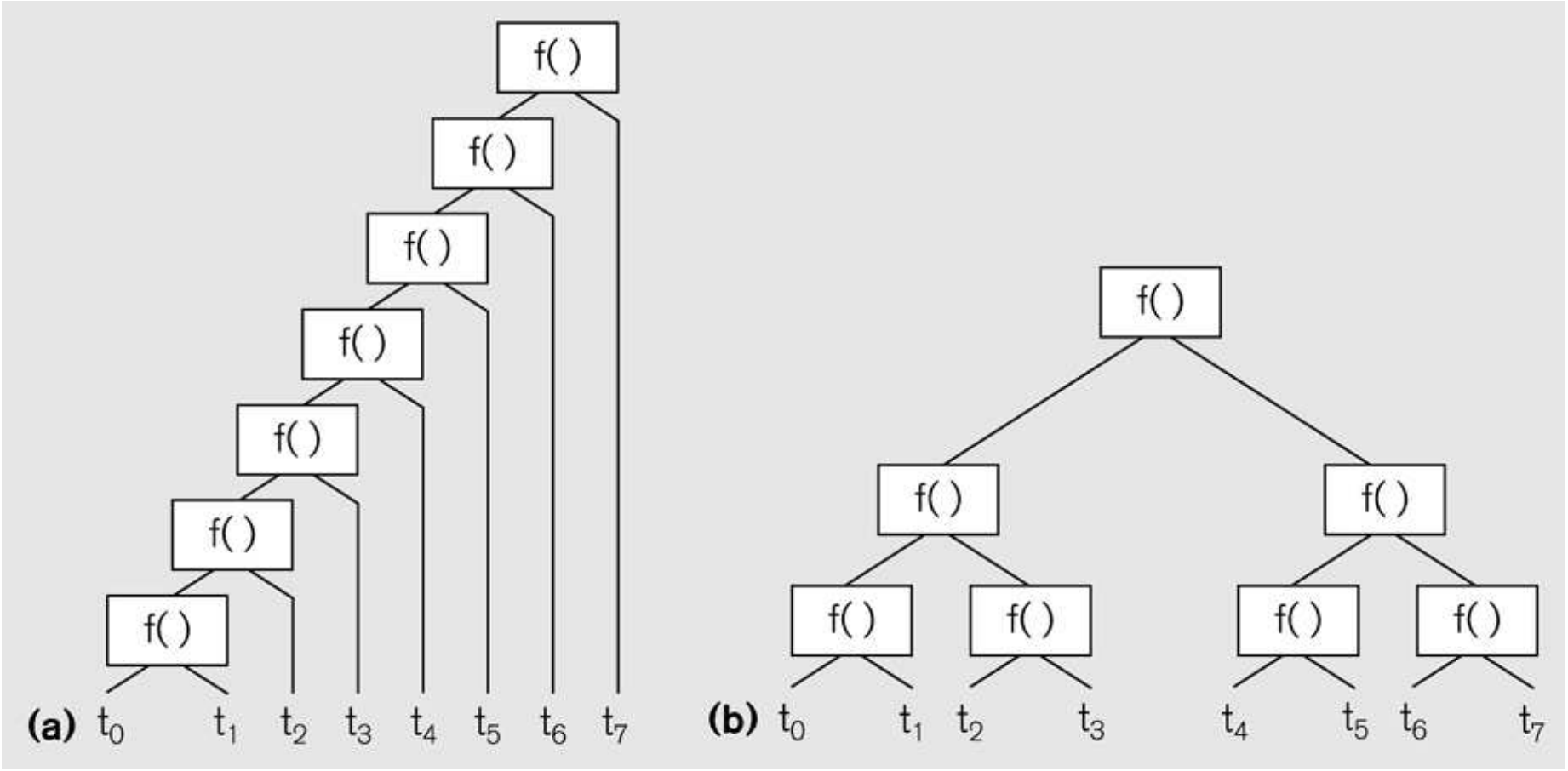
This is a ***false dependence***

Rewrite:

```
sum = a+1;
first_term = sum*scale1;
sum2 = b+1;
second_term = sum2*scale2;
```

Avoiding Dependencies

Sometimes, you can change the algorithm



Lack of Dependencies

A task that spends all its time on many mutually independent computations is
embarassingly parallel

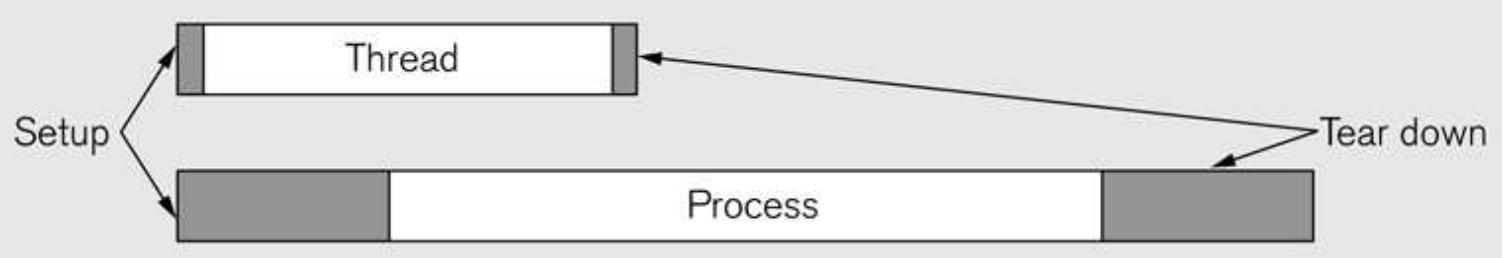
Other Non-Parallelism

Other kinds of non-parallelism:

- Memory-bound computation
- I/O-bound computation
- Load imbalance

- Measuring Performance
- Obstacle: Non-Parallelism
- Obstacle: Overhead

Overhead



Overhead

Sources of overhead:

- Communication and synchronization
- Contention
- Extra computation
- Extra memory

Overhead

Reducing communication and contention overhead:

- Larger ***granularity***, so that per-message overhead is less costly
  - Example: pass whole array section instead of individual elements
- Improve ***locality***, so that less communication is needed
  - Example: compute sums where data already resides
- Recompute instead of communicating
  - Example: recompute pseudo-random sequences instead of centralizing

Overhead

Trade-offs:

- Communication versus computation
- Memory versus parallelism
- Overhead versus parallelism