Parallelism and Parallel Performance

CPSC 424/524
Lecture #4
September 17, 2018

Topics

- Types of Parallelism
  - Instruction Level Parallelism
  - Multicore/Multi-Node
  - Flynn's Taxonomy
- Parallel Performance
  - Speedup, Efficiency, Scalability
  - Amdahl's Law
  - Gustafson's Law

Types of Parallelism

- Parallelism can occur at multiple levels
  - Instruction level parallelism
    - Multiple functional units
    - Pipelining
  - Multicore parallelism
    - Separate cpus in a single chip or node
    - May or may not be in lockstep
    - May or may not share data
  - Multinode parallelism
    - Completely independent processors
    - Communication requires an interconnection network
    - Software may hide the communication from the programmer

Flynn's Taxonomy

SISD Single instruction stream Single data stream	SIMD/SPMD Single instruction stream Multiple data stream
MISD Multiple instruction stream Single data stream	MIMD/MPMD Multiple instruction stream Multiple data stream

classic von Neumann

Vector processors (SSE, MMX, AVX); GPUs

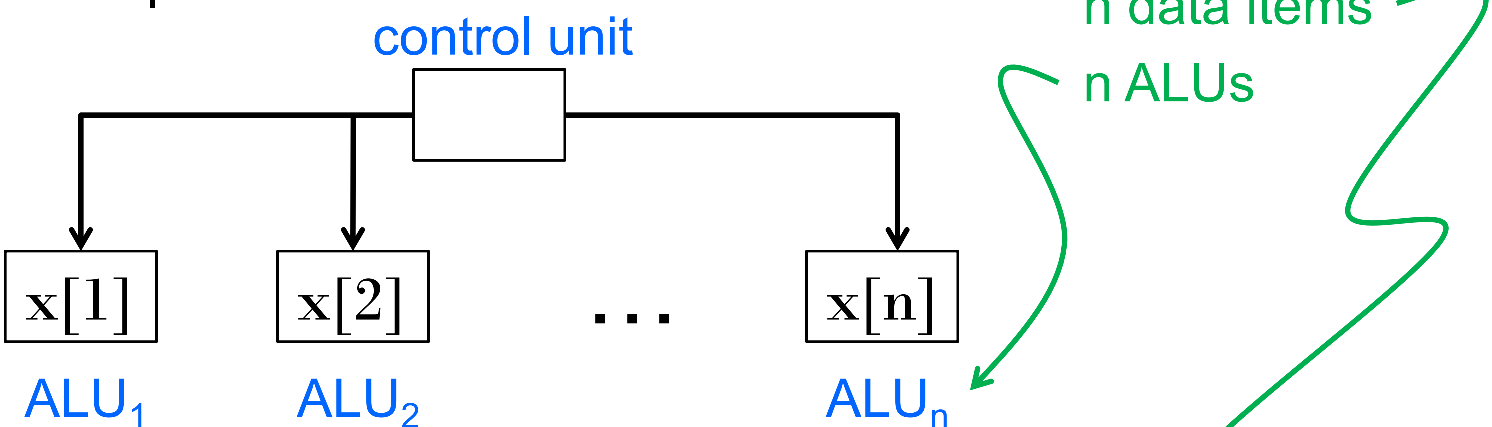
pipelining

Multicore/Multi-node parallel machines

SIMD: Data Parallelism

- Parallelism achieved by dividing data among functional units such as multiple arithmetic logic units (ALUs) that run in lockstep
- Same instruction applied at once to multiple data (vector entries)
- Sometimes combined with pipelining

Simple Example:



$$\text{for } (i = 0;\ i < n;\ i{+}{+})$$
$$x[i] \mathrel{+}= y[i];$$

Classical SIMD drawbacks

- All units must execute the same instruction, or remain idle
- In classic design, units must also operate synchronously
- The units have no instruction storage (may matter if each unit handles multiple vector entries)
- Efficient for large “data parallel” problems, but not for complex parallel problems, such as those with lots of program logic

- These drawbacks are addressed in modern SIMD-like processors:
  - Vector Processors (e.g. Intel SSE or AVX extensions)
    - Operate on vectors of data rather than individual scalars
    - Have pipelined vector functional units with instruction storage & “chaining”
  - GPUs
    - Not strictly SIMD; (e.g., may cache instructions)
    - Relax the requirement for synchronous operations
    - Incorporate pipelining and vector operations in some cases

MIMD

- Supports multiple simultaneous instruction streams operating on multiple data streams
- Typically consists of a collection of fully independent processing units or cores, each of which has its own control unit and its own functional units

Theoretical Speedup and Efficiency

<u>Speedup</u> is simply the ratio between serial and parallel time

$$S(p) = \frac{\text{Execution time using one processor (best sequential algorithm)}}{\text{Execution time using a multiprocessor with } p \text{ processors}} = \frac{t_s}{t_p}$$

where t_s is "best possible" execution time on one processor of your parallel machine, and t_p is execution time on multiple processors (ignoring overhead, such as communication). *For the moment, consider t_s to be constant.*

$S(p)$ characterizes performance gain from a multiprocessor.

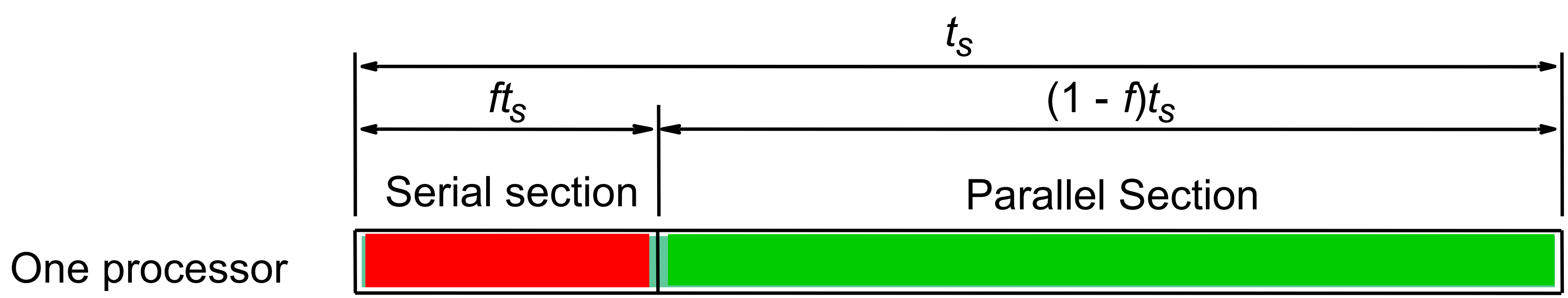
<u>Parallel Efficiency</u> is the Speedup per processor:

$$E(p) = \frac{S(p)}{p} = \frac{t_s}{p \cdot t_p} \leq 1 \quad \text{(in general) (Why?)}$$

Parallel vs. Serial Performance

Suppose we have some parallelizable problem to solve and that the "best" serial (i.e., sequential) method requires time t_s on a single processor of your parallel machine.

Now, suppose we determine that some fraction f of the program (the "Serial Section") must be done serially, but that the remainder (the "Parallel Section") can be parallelized.



What can we hope to achieve using parallel computation?

Driving Analogy

Suppose you want to drive from A to B, a distance of N miles. Consider a “serial car” that travels at exactly 30 mph. The serial implementation uses only this car.

Now suppose that after driving 30 miles (the serial section), you can change to any transportation device of your choosing (the “parallel car”) for the remainder of the trip (the parallel section).

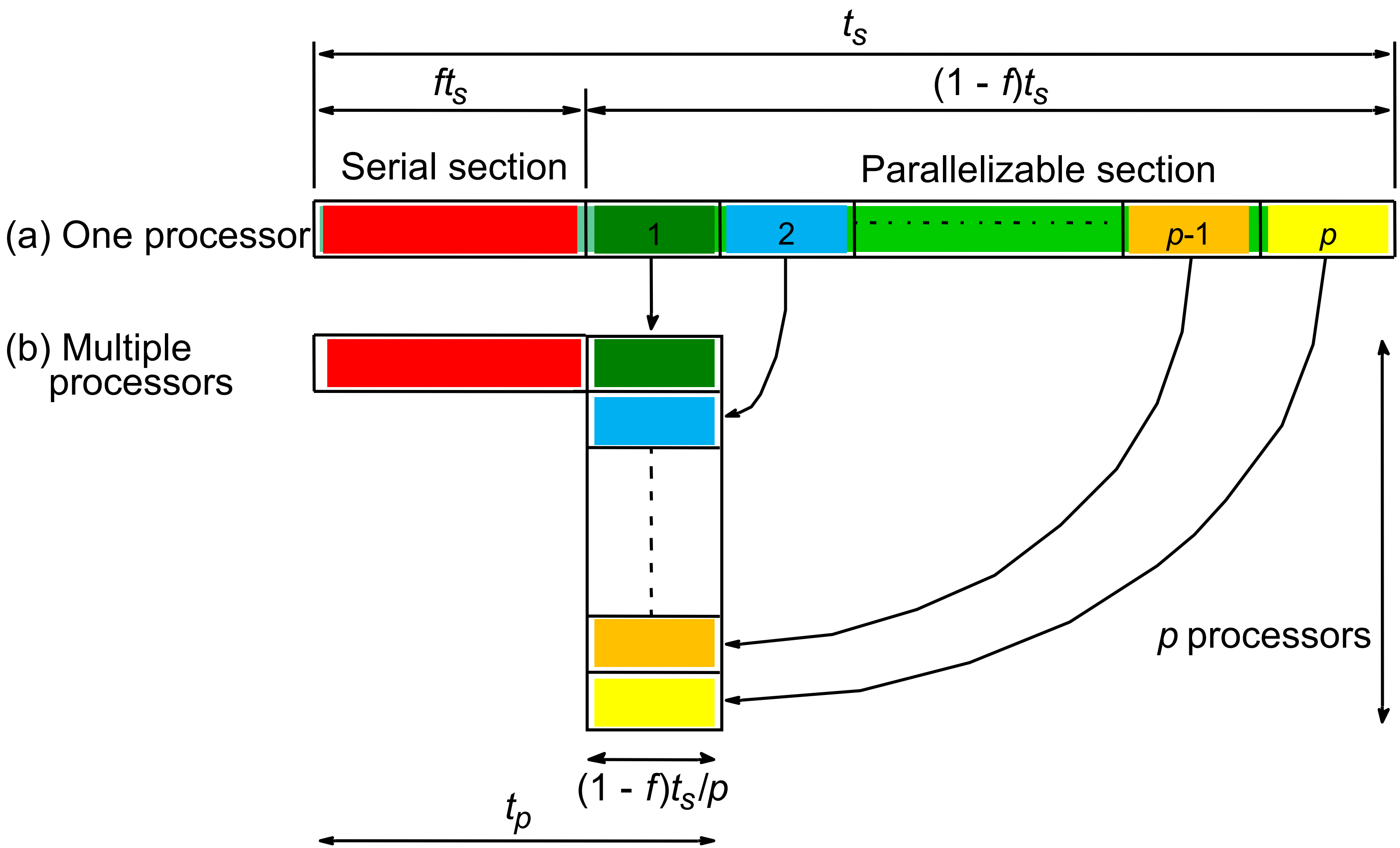
Example: Suppose N=120. What are the serial time and best possible parallel time/speedup?

$$t_s = 4$$

$$t_p = 1 + [\text{time to go final 90 miles}]$$

$$S(p) = \frac{t_s}{t_p} \leq 4$$

Parallel Performance



Speedup and Efficiency

Speedup factor is given by:

$$S(p) = \frac{t_s}{f t_s + (1 - f) t_s / p} = \frac{p}{1 + (p - 1) f}$$

Parallel Efficiency is given by:

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

Amdahl's Law

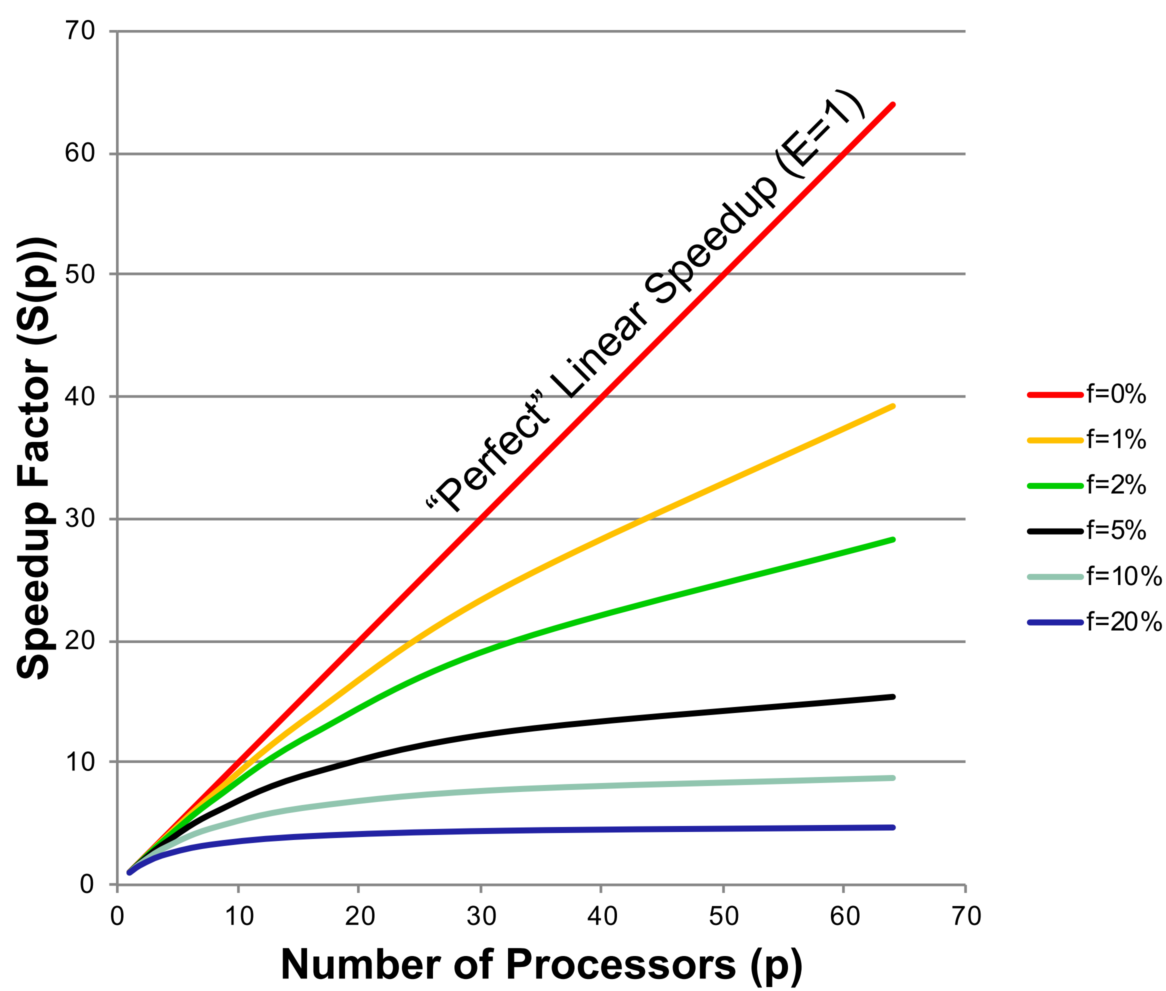
Amdahl's law assumes that the serial fraction f is fixed, independent of p and the problem size. Then, in the best ("most parallel") case:

$$S(p)_{max} = \lim_{p \to \infty} \frac{p}{1 + (p-1)f} = \frac{1}{f}$$

Bad News:

$$E(p)_{max} = \lim_{p \to \infty} \frac{S(p)}{p} = \lim_{p \to \infty} \frac{1}{pf} = 0$$

Speedup versus number of processors

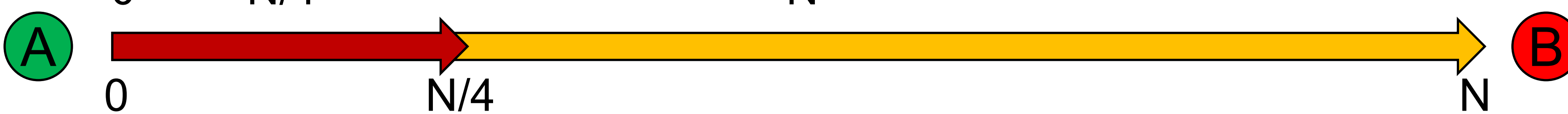


Driving Analogy Redux

Consider traveling from A to B using two cars: the 30 mph “serial car” and the arbitrarily-fast “parallel car.” In the serial case, you only use the serial car.

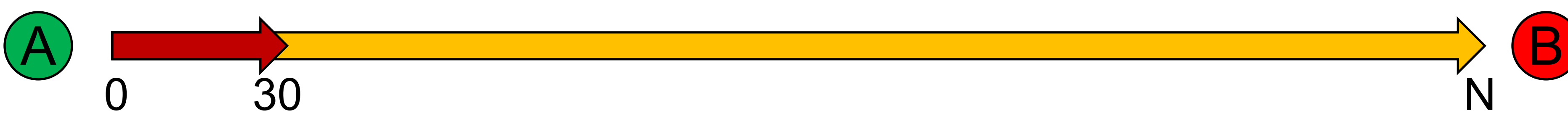
Amdahl’s Law: Suppose $f = \frac{1}{4}$.You must go 1/4 of the serial time (or of the total distance) in the serial car before you can switch vehicles.

What’s the maximum speedup? Does the answer depend on *N*?



Gustafson’s Law: You must go 1 hour (30 miles) in the serial car, regardless of N, before you can switch vehicles.

Now does the speedup depend on N? What’s the maximum speedup?



Scaled Speedup (Gustafson's Law)

- Amdahl's Law: Serial fraction f is fixed, independent of problem size N.
- Gustafson's Law: Serial time $K_s(N,p)$ satisfies $K_s / t_p \to 0$ as N or N & p grow. [Want to have "constant" parallel work per processor as p grows.]
- For a given N :

$$t_p = K_s + (t_s - K_s) / p$$ (For perfect parallelism)

$$t_s = K_s + p \cdot (t_p - K_s)$$ (Rearrange the equation)

Scaled Speedup

$$S_s(p) = \frac{t_s}{t_p} = \frac{K_s}{t_p} + p \cdot \left(1 - \frac{K_s}{t_p}\right)$$

- What happens as N increases with fixed p?
- What happens as p and N increase together at a linked rate?
- What happens if p grows "too quickly" relative to N? (E.g., $p \to \infty$ w/fixed N)

Scalability

- In general, a parallel algorithm/program is scalable if p can increase while keeping parallel efficiency nearly constant
- Strong Scalability
  - Problem size stays fixed
  - p increases
- Weak Scalability
  - Problem size and p both grow (possibly at a linked rate)

Maximum Absolute Speedup

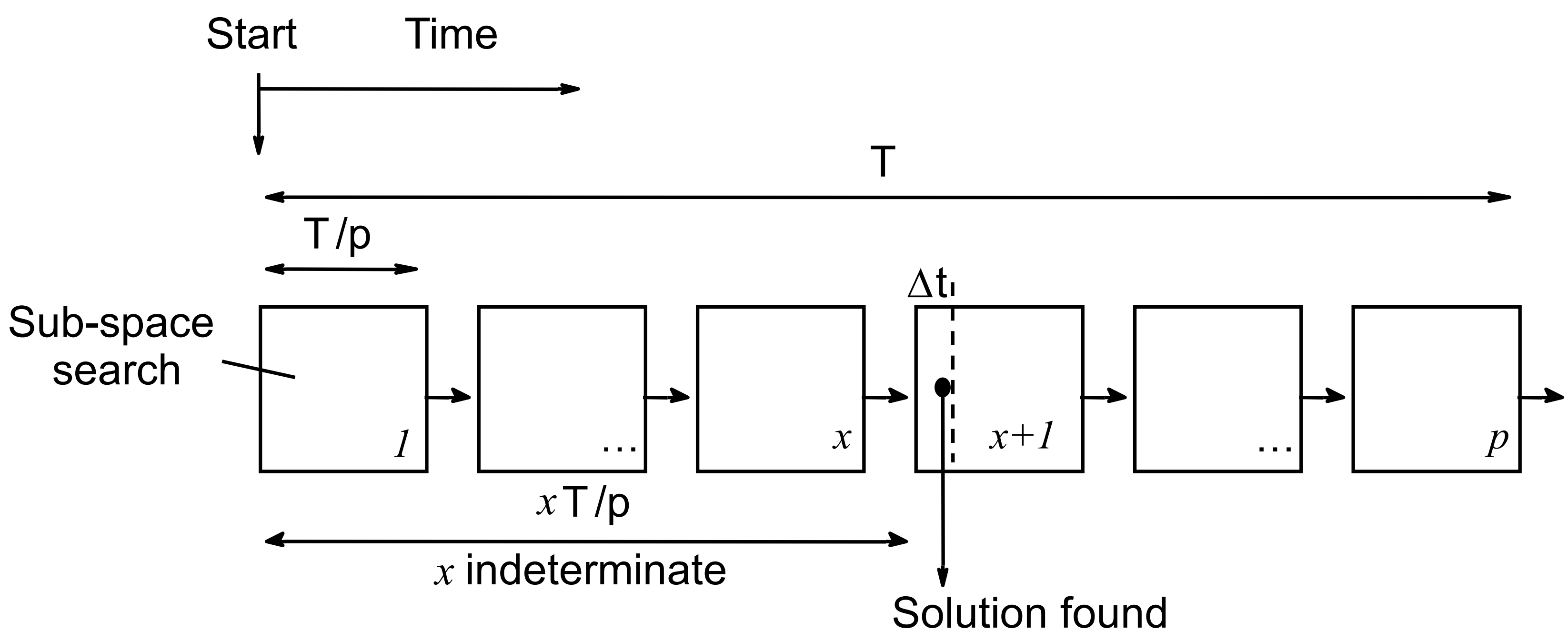
Maximum ***potential*** absolute speedup:
Usually p with p processors (linear speedup) *if the entire computation can be parallelized*.

Possible to get "superlinear" speedup (greater than p), but usually for a specific reason such as:

- Nondeterministic algorithm
- Extra memory in multiprocessor system

Superlinear Speedup Example: Searching

Suppose you have *p* subspaces to search, and that searching *all* of them serially would take time *T*:



$$t_s = x \times T/p + \Delta t$$

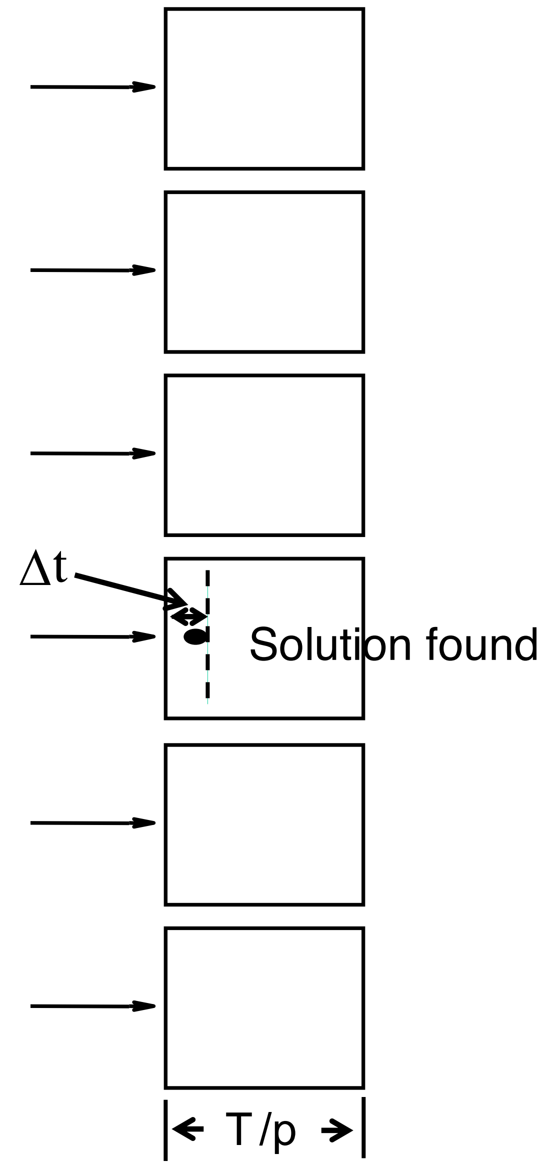
Superlinear Speedup Example: Searching (cont.)

Now search the sub-spaces in parallel using p processors.

The solution is found in time Δt, so the speedup is:

$$S(p) = \frac{\left(x \times \frac{T}{p}\right) + \Delta t}{\Delta t}$$

$$\lim_{\Delta t \to 0} S(p) = \infty$$



Superlinear Speedup Example: Searching (cont.)

Least advantage for parallel version when solution found in first sub-space search of the sequential search, i.e.

$$S(p) = \frac{\Delta t}{\Delta t} = 1$$

Actual speed-up depends upon which subspace holds solution but could be extremely large.

Practical Considerations

In the real world, parallel overhead must be considered.

$$t_p = t_{comp} + t_{overhead}$$

So far, we've only considered the computation time. A major theme of this course will be understanding the overhead time and finding ways to either reduce it or hide it behind the computation time.