“Performance & Evaluation”

Outline

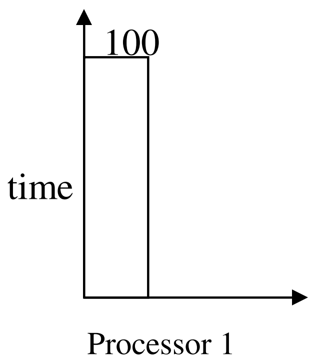
- Performance metrics
  - Speedup
  - Efficiency
  - Scalability
- Examples
- Reading: Kumar – ch 5; Foster – ch 2

Speedup

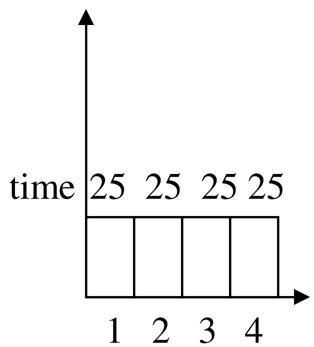
- T_s =time for the *best* serial algorithm
- T_p=time for parallel algorithm using p processors

$$S_p = \frac{T_s}{T_p}$$

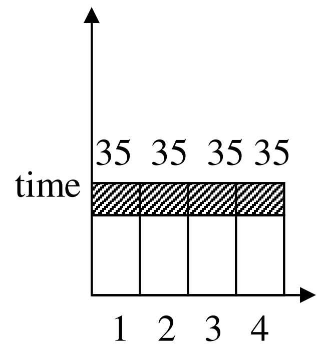
Example



(a)



(b)



(c)

$$S_p = \frac{100}{25} = 4.0,$$

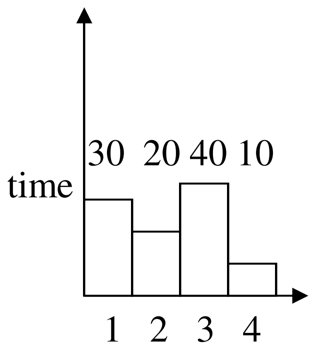
perfect parallelization

$$S_p = \frac{100}{35} = 2.85,$$

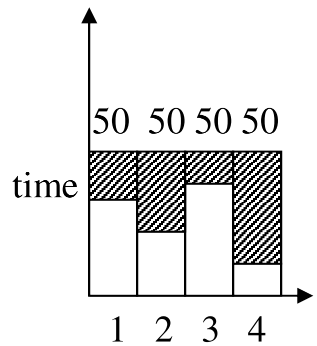
perfect load balancing

but synch cost is 10

Example (cont.)



(d)



(e)

$$S_p = \frac{100}{40} = 2.5,$$

no synch

but load imbalance

$$S_p = \frac{100}{50} = 2.0,$$

load imbalance

and synch cost

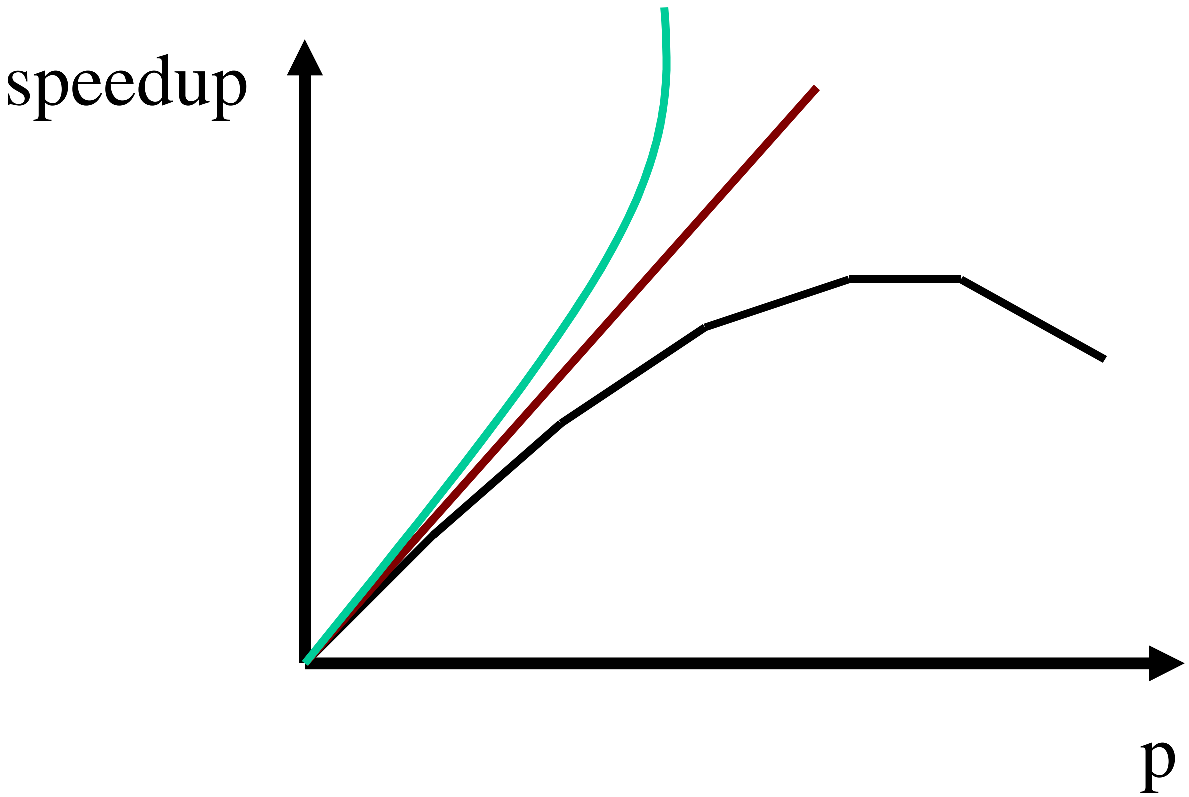
What Is "Good" Speedup?

- *Linear* speedup:

- *Super*linear speedup

- *Sub-linear* speedup:

Speedup



Sources of Parallel Overheads

- Interprocessor communication
  - Data movement costs
- Load imbalance
- Synchronization
- Extra computation
  - Computation not performed by serial version, e.g., partially-replicated computation to reduce communication
- Contention
  - Memory
  - Network

Amdahl's Law

- The performance improvement that can be gained by a parallel implementation is limited by ?

- Then, parallel run time can be written as?

Fixed-Sized Speedup Amdahl's Law

- Fixed-Size Speedup (Amdahl's law)
  - Emphasis on turnaround time
  - Problem size, ***W***, is fixed

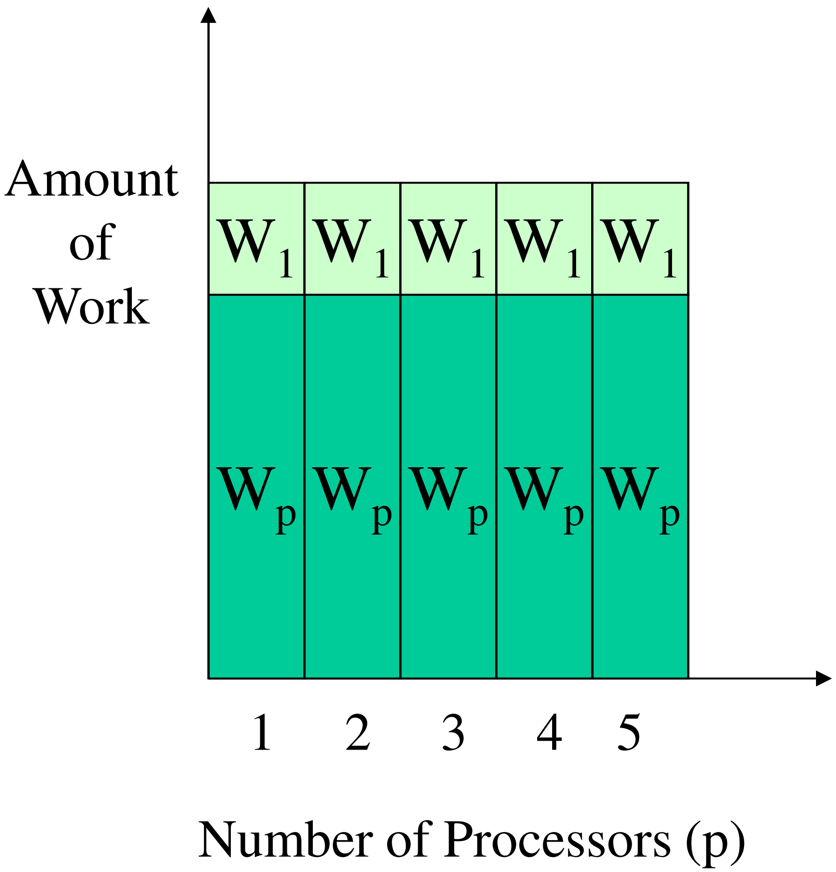
$$S_p = \frac{\text{Uniprocessor Execution Time}}{\text{Parallel Execution Time}}$$

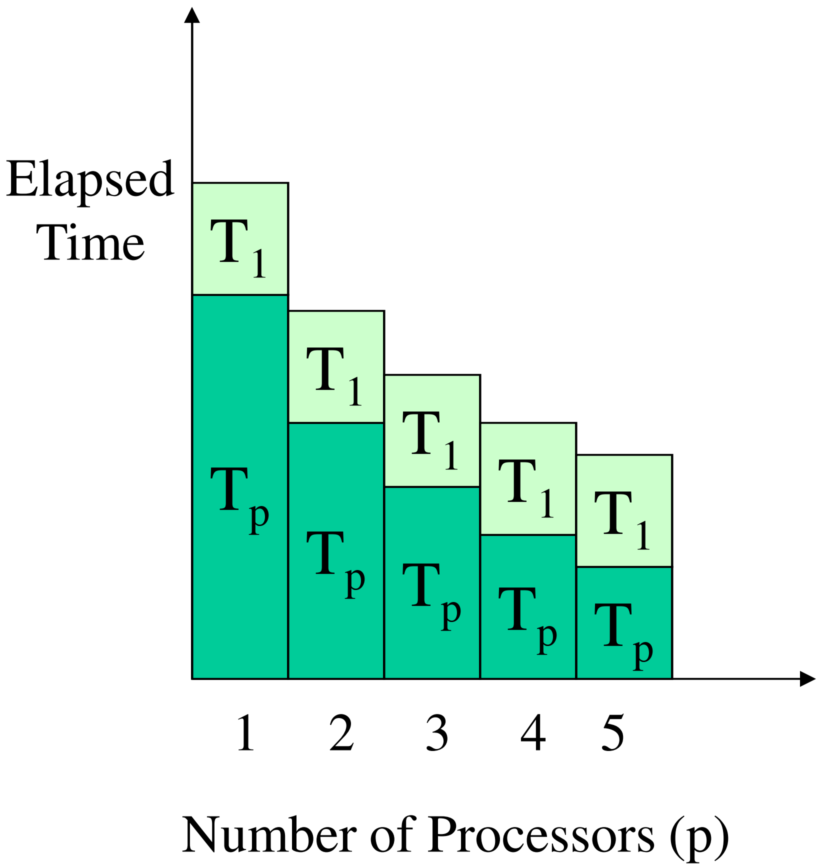
$$S_p = \frac{\text{Uniprocessor Time of Solving } W}{\text{Parallel Time of Solving } W}$$

Amdahl's Law

- **Amdahl's law** gives a limit on speedup in terms of α

• Fixed-Size Speedup (Amdahl Law, '67)





Amdahl's Law

- The speedup that is achievable on p processors is ?
- If we assume that the serial fraction is fixed, then ?
- For example, if α=10%, then ?

Comments on Amdahl's Law

- The Amdahl's fraction α in practice depends on the problem size n and the number of processors p

- What is an effective parallel algorithm?

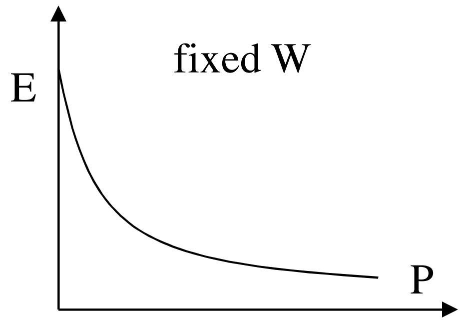
- Can we get linear speedup with an effective parallel algorithm?

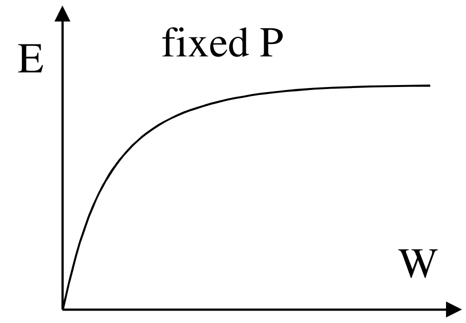
Efficiency

$$E_p = \frac{S_p}{p}$$

- Bounds:
  - Theoretically: between [0,1]
  - In practice, may be greater than 1 if superlinear speedup
- The fraction of total potential processing power that is actually used.
  - A program with linear speedup is 100% efficient

Scalability

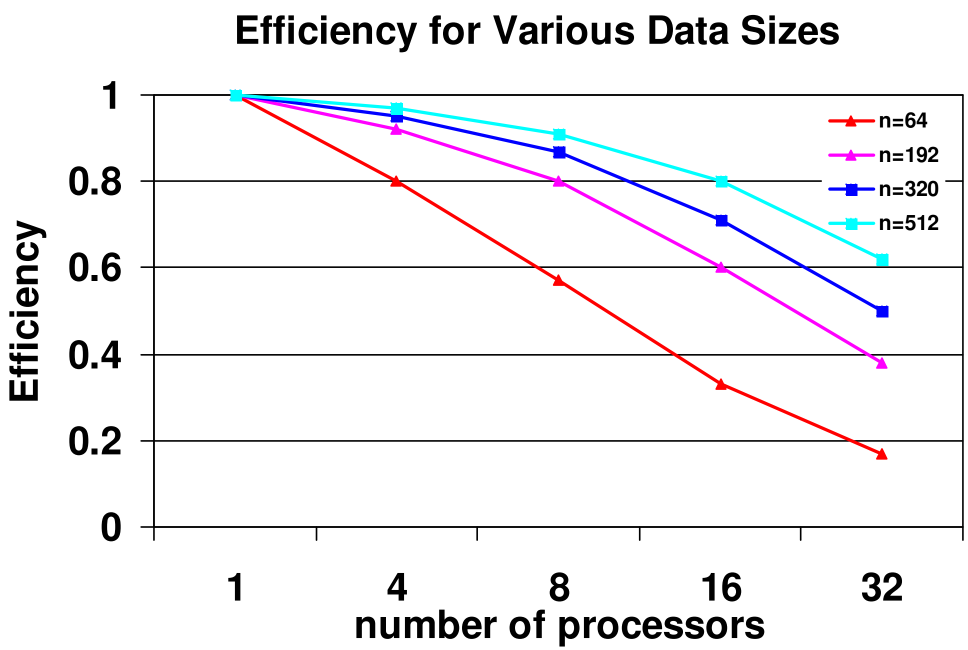




- Definition?
  - Can keep efficiency constant by
    - increasing the problem size
    - Proportionally increasing the number of processors
  - Such systems are called scalable parallel systems

Scalability

- Efficiency of adding n numbers in parallel



$E=1/(1+2plogp/n)$

  – For an efficiency of 0.80 on 4 procs, n=64
  – For an efficiency of 0.80 on 8 procs, n=192
  – For an efficiency of 0.80 on 16 procs, n=512

Isoefficiency Scalability

- Dictates the growth rate of problem size required to keep the efficiency fixed as the number of processing units increases

$$W = KT_0(W, p) \qquad (\text{ K be constant})$$

- For a desired value E

$$E = \frac{1}{1 + T_0(W, p)/W}$$

$$\frac{T_0(W, p)}{W} = \frac{1-E}{E}$$

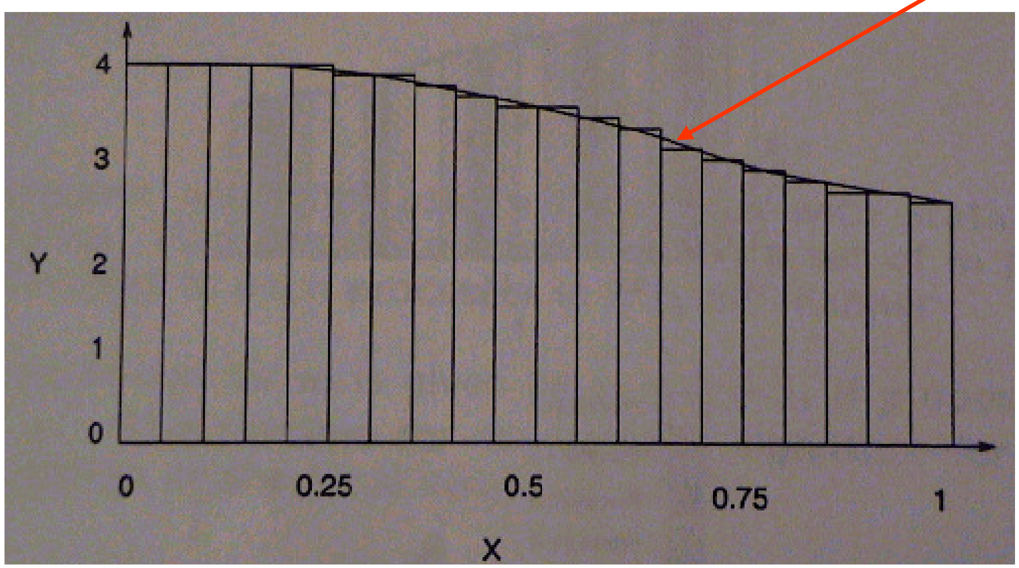
$$W = \frac{E}{1-E} T_0(W, p)$$

Compute π: Problem

- Consider parallel algorithm for computing the value of π=3.1415…through the following numerical integration

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

$$\boxed{\frac{4}{1+x^2}}$$

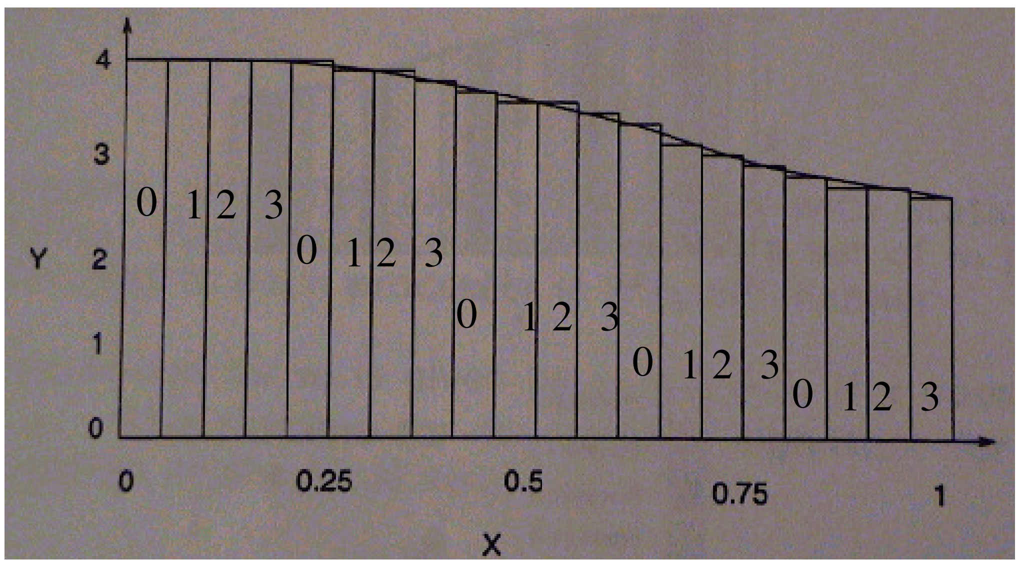


Compute π: Sequential Algorithm

```
computepi()
{
    h=1.0/n;
    sum =0.0;
    for (i=0;i<n;i++) {
        x=h*(i+0.5);
        sum=sum+4.0/(1+x*x);
    }
    pi=h*sum;
}
```

Compute π: Parallel Algorithm

- Each processor computes on a set of about n/p points which are allocated to each processor in a cyclic manner
- Finally, we assume that the local values of π are accumulated among the p processors under synchronization



Compute π: Parallel Algorithm

```
computepi()
{
    id=my_proc_id();
    nprocs=number_of_procs():
    h=1.0/n;
    sum=0.0;
    for(i=id;i<n;i=i+nprocs) {
        x=h*(i+0.5);
        sum=sum+4.0/(1+x*x);
    }
    localpi=sum*h;
    use_tree_based_combining_for_critical_section();
        pi=pi+localpi;
    end_critical_section();
}
```

Compute π: Sequential Analysis

- Assume that the computation of π is performed over n points
- For n points, the number of operations executed in the sequential algorithm is:

Compute π: Parallel Analysis

- The parallel algorithm uses p processors. Each processor computes on a set of m points which are allocated to each process in a cyclic manner

- The expression for m is given by $m \leq \frac{n}{p} + 1$ if p does not exactly divide n. The runtime for the parallel algorithm for the parallel computation of the local values of π is:

Compute π: Parallel Analysis

- The accumulation of the local values of π using a tree-based combining can be optimally performed in $\log_2(p)$ steps
- The total runtime for the parallel algorithm for the computation of π is:

- The speedup of the parallel algorithm is:

Compute π: Parallel Analysis

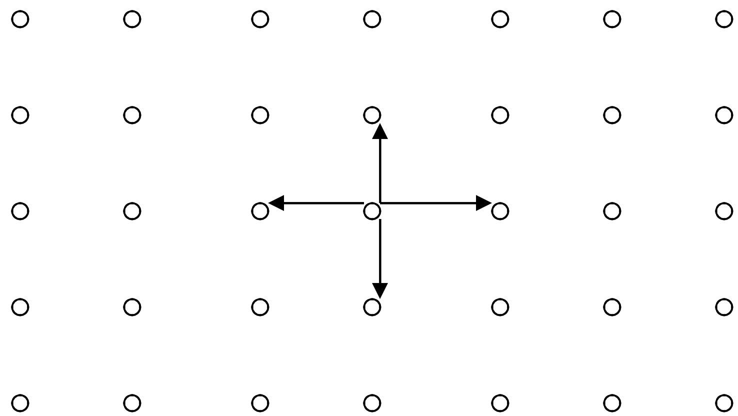
- The Amdahl's fraction α is:

- The parallel algorithm is effective or not?

Finite Differences: Problem

- Consider a finite difference iterative method applied to a 2D grid where:

$$X_{i,j}^{t+1} = \omega \cdot (X_{i,j-1}^{t} + X_{i,j+1}^{t} + X_{i-1,j}^{t} + X_{i+1,j}^{t}) + (1-\omega) \cdot X_{i,j}^{t}$$

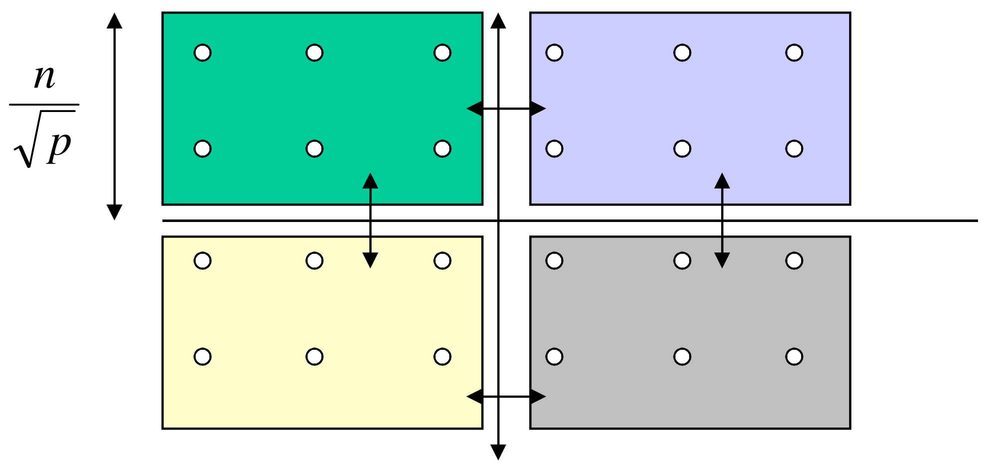


Finite Differences: Serial Algorithm

```
finitediff()
{
    for (t=0;t<T;t++) {
        for (i=0;i<n;i++) {
            for (j=0;j<n;j++) {
                x[i,j]=w_1*(x[i,j-1]+x[i,j+1]+x[i-1,j]+x[i+1,j]+w_2*x[i,j];
            }
        }
    }
}
```

Finite Differences: Parallel Algorithm

- Each processor computes on a sub-grid of $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ points
- Synch between processors after every iteration ensures correct values being used for subsequent iterations



Finite Differences: Parallel Algorithm

```
finitediff()
{
    row_id=my_processor_row_id();
    col_id=my_processor_col_id();
    p=numbre_of_processors();
    sp=sqrt(p);
    rows=cols=ceil(n/sp);
    row_start=row_id*rows;
    col_start=col_id*cols;
    for (t=0;t<T;t++) {
            for (i=row_start;i<min(row_start+rows,n);i++) {
                for (j=col_start;j<min(col_start+cols,n);j++) {
                    x[i,j]=w_1*(x[i,j-1]+x[i,j+1]+x[i-1,j]+x[i+1,j]+w_2*x[i,j];
                }
                barrier();
            }
    }
}
```

Finite Differences: Analysis

- For an n*n grid and T iterations, the number of operations executed in the sequential algorithm is:

Finite Differences: Analysis

- The parallel algorithm uses p processors. Each processor computes on an m*m sub-grid allocated to each processor in a blockwise manner

- The expression for m is given by $m \le \left\lceil \frac{n}{\sqrt{p}} \right\rceil$ The runtime for the parallel algorithm is:

Finite Differences: Analysis

- The barrier synch needed for each iteration can be optimally performed in log(p) steps
- The total runtime for the parallel algorithm for the computation is:

- The speedup of the parallel algorithm is:

Finite Differences: Analysis

- The Amdahl's fraction is:

- The parallel algorithm is effective or not?

Fixed-Time Speedup Gustafson Law

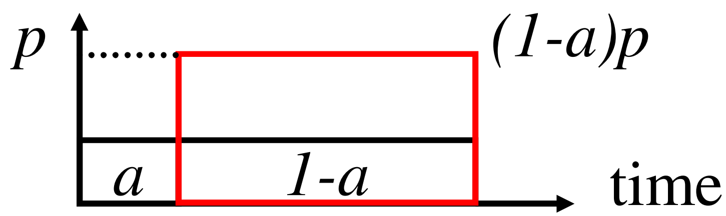
- Fixed-Time Speedup (Gustafson, '88)
  - Emphasis on work finished in a fixed time
  - Problem size is scaled from W to W'
  - W': Work finished within the fixed time with parallel processing

$$S'_p = \frac{\text{Uniprocessor Time of Solving } W'}{\text{Parallel Time of Solving } W'}$$

$$= \frac{\text{Uniprocessor Time of Solving } W'}{\text{Uniprocessor Time of Solving } W}$$

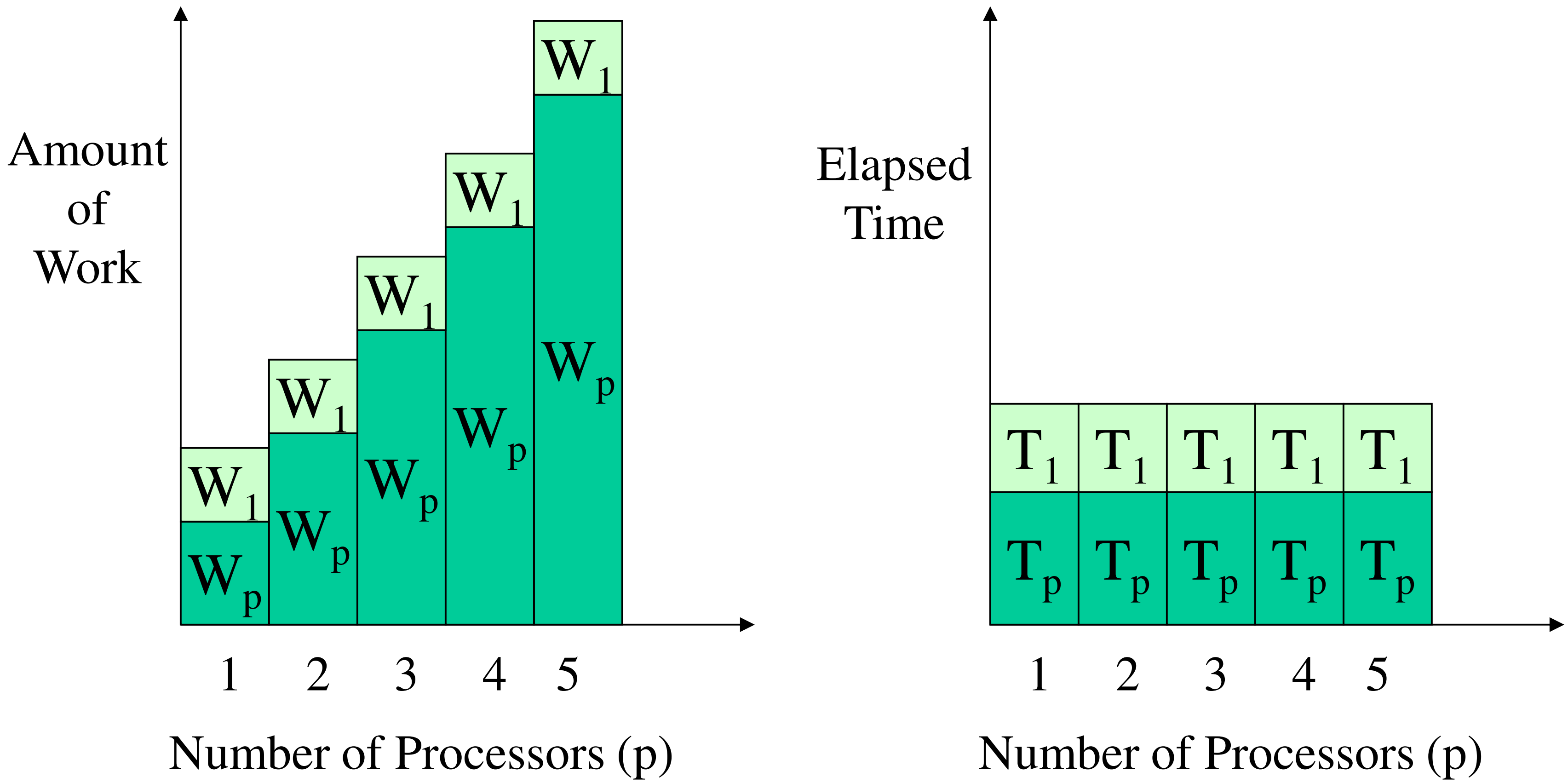
$$= \frac{W'}{W}$$

Gustafson's Law (Without Overhead)



$$Speedup_{FT} = \frac{Work(p)}{Work(1)} = \frac{\alpha W + (1-\alpha)pW}{W} = \alpha + (1-\alpha)p$$

- Fixed-Time Speedup (Gustafson)



Converting α's between Amdahl's and Gustafon's laws

$$\frac{p}{\alpha_A(p-1)+1} = \alpha_G + (1-\alpha_G)p$$

$$\alpha_A = \frac{1}{1+\frac{(1-\alpha_G).p}{\alpha_G}}$$