Electrical Engineering and Computer Science

EECS 358 - INTRODUCTION TO PARALLEL COMPUTING

Lecture 2

Introduction to Parallel Programming

Outline

- What is parallel programming
- Why parallel programming
- Identifying parallelism
- Types of parallelism
- Performance of parallel programs
- Examples of parallel programs
- Summary

What is Parallel Programming ?

- Parallel programming involves constructing or modifying a program for solving a given problem on a parallel machine starting from:
  - A serial algorithm for the problem
  - A serial program for solving the problem
  - A parallel algorithm for the problem
  - A parallel program for the problem
- Goals of parallel programming - performance, performance, performance
- Effective parallel programming requires:
  - Minimization of inter processor synchronization costs
  - Equal sharing of problem load between processors

Why Parallel Programming ?

- Alternatives to parallel programming:
  - Libraries
  - Parallelizing Compilers
- Libraries:
  - Built for special application classes such as Linear algebra
  - Reduce flexibility and do not allow for customization
  - Handle very general input data sets which preclude optimizations possible for certain data sets
- Parallelizing Compilers:
  - State-of-the-art analysis and transformation techniques cannot handle all types of programs
  - Often, it is difficult to infer the underlying algorithm from a given serial program even with detailed analysis

Data Dependence

- No Dependence (can run in parallel):

```
S1: X = K + 3;
S2: Y = Z * 5;
```

- True Dependence (cannot run in parallel):

```
S1:    X = 3;
S2:    Y = X * 4;
```

- Anti Dependence (cannot run in parallel):

```
S1:    Y = X * 4;
S2:    X = 3;
```

- Output Dependence (cannot run in parallel):

```
S1:    X = Y * 4;
S2:    X = 3;
```

Data Dependence and Parallelization

- Consider the following loop of a C program:

```
for (i=0; i < 1000; i++)
  a[i] = b[i] + c[i];
```

- If one unfolds the loops, the statements would be executed as follows:

```
  a[0] = b[0] + c[0];
  a[1] = b[1] + c[1];
  a[2] = b[2] + c[2];
        .....
  a[999] = b[999] + c[999];
```

- Each iteration can be executed in parallel

Data and Functional Parallelism

- Data parallel: C program loops where each iteration of a loop is independent and represents a simple statement and is executed on a different processor

```
for (i=0; i < 1000; i++)
  a[i] = b[i] + c[i];
```

- Functional parallel: Multiple C program loops which cannot be parallelized individually, but the different code blocks are independent and are executed on different processors

```
for (i=0; i < 10; i++)  /* block 1 */
  b[i+1] = b[i] + c[i];
 ...
for (j=0; j < 5; j++)   /* block n */
  a[j+1] = a[j] + d[j];
```

Coarse and Fine Grain Parallelism

- Grain size categorizes amount of compute work done over independent sub-tasks in parallel
- Coarse grain: implies thousands of instructions, e.g. functions or procedure calls in programs – Each group of loop iterations of C program representing complex sets of statements containing function calls executed on different processor

```
for (i=0; i < 1000; i++) a[i] = b[i] + c[i] * work(d[i]);
```

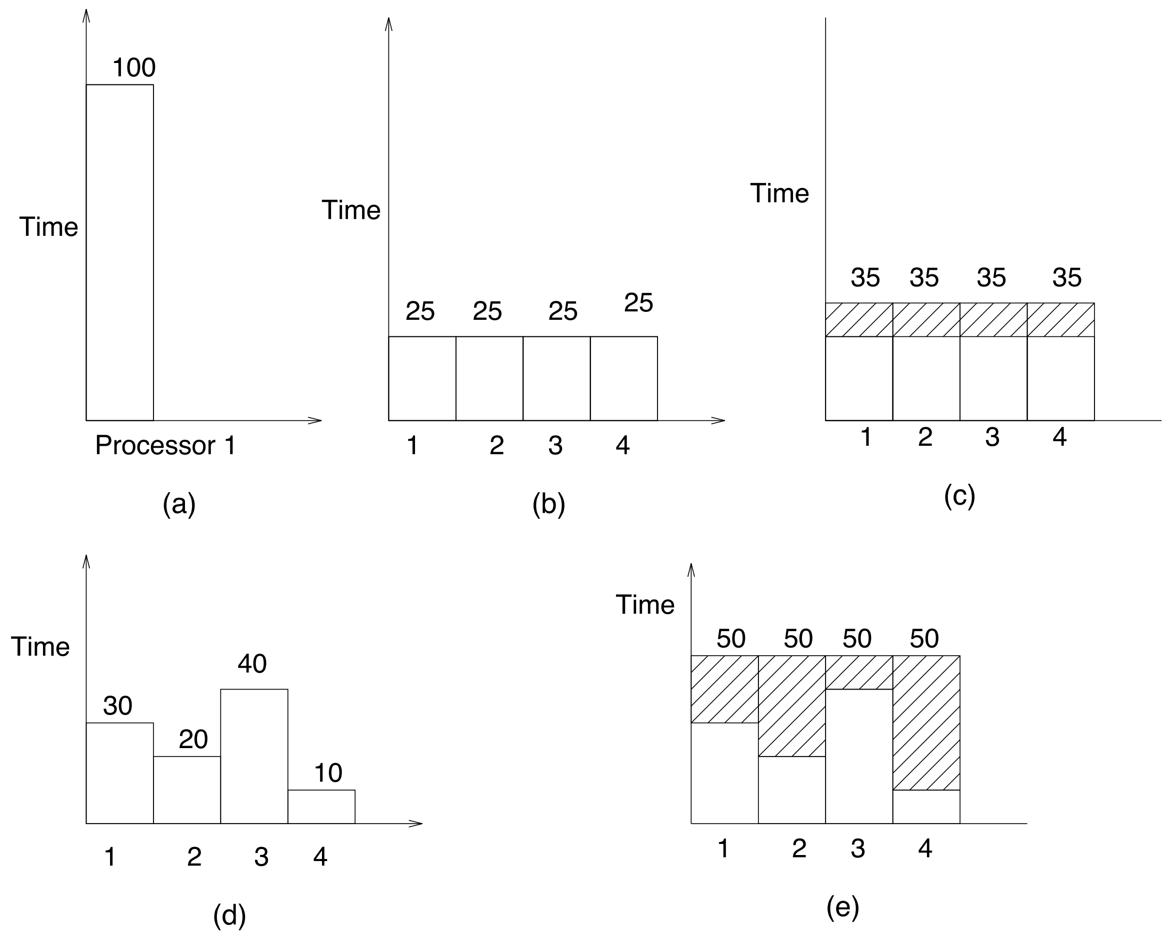
- Fine grain: implies tens of instructions, e.g. statements in programs – Each loop iteration of C program is executed on a different processor

```
for (i=0; i < 1000; i++) a[i] = b[i] + c[i];
```

Performance of Parallel Programs

- T = time for the best serial algorithm
- T_p = time for parallel algorithm using p processors
- Speedup $S_p = \frac{T}{T_p}$
- Efficiency or Utilization $E_p = \frac{S_p}{p}$
- If parallel algorithm is 100% efficient, then one observes linear speedups
- Efficiency is practically never 100% due to:
  - Cost of synchronization or communication across processors
  - Suboptimal load balance among parallel processors

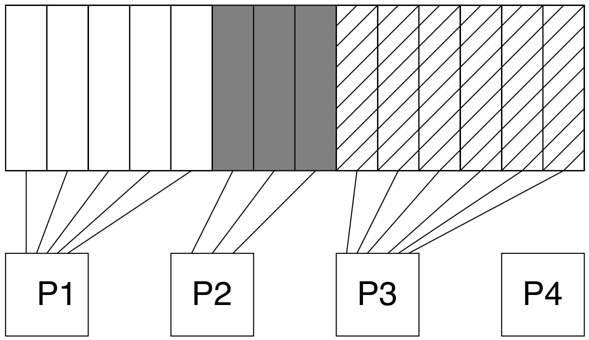
Example Timecharts



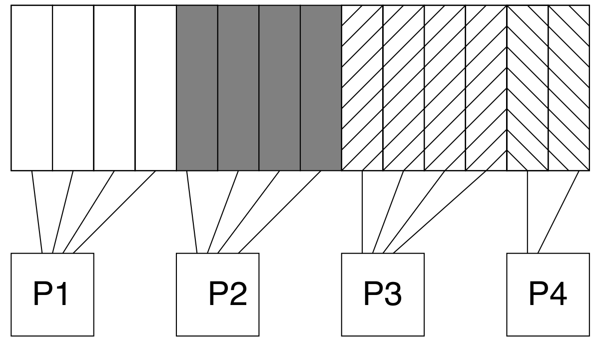
Example Timecharts

- Case (a) represents serial time 100 units
- Case (b) represents perfect parallelization time 25 units, speedup 4
- Case (c) representes perfect load balance but synch cost 10, hence speedup $= 100/35 = 2.85$
- Case (d) represents no synch but load imbalance, speedup $= 100/40 = 2.5$
- Case (e) represents load imbalance and synch cost, speedup $= 100/50 = 2$

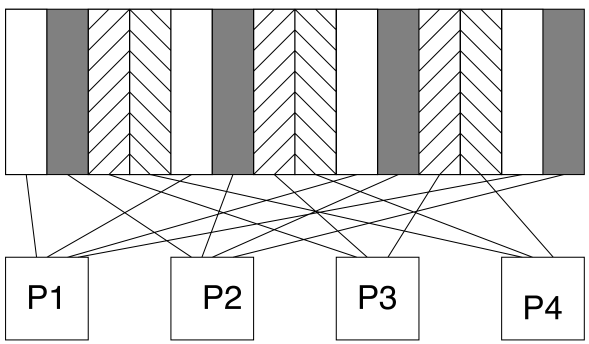
Load Balancing and Scheduling



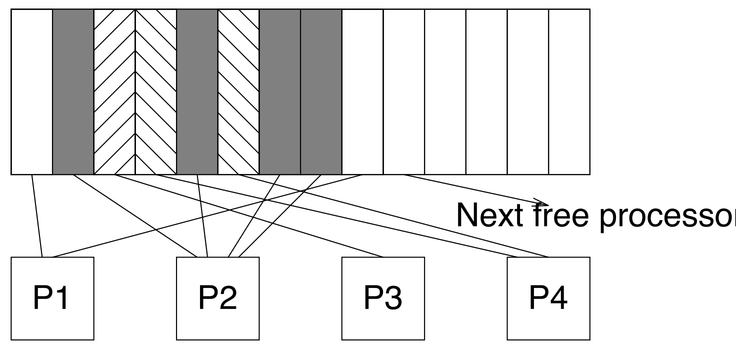
(a)



(b)



(c)



(d)

Load Balancing and Scheduling

- Prescheduling
- Static blockwise scheduling
- Static interleaved scheduling
- Dynamic scheduling

Amdahl's Law

- The law states that the performance improvement that can be gained by a parallel implementation is limited by the fraction of time parallelism can actually be used in an application
- Consider an application that takes T time units when executed in a serial mode on a single processor
- When the application is parallelized, we assume that a parameter α constitutes the fraction that cannot be parallelized (serial fraction)
- Assuming p processors in a parallel implementation, the time for execution is given by the following expression (assuming perfect speedups in the portion of the application that can be parallelized):

$$T_p = (\alpha + \frac{1-\alpha}{p}) \cdot T$$

Amdahl's Law

- The speedup that is achievable on p processors is:

$$S_p = \frac{T_s}{T_p} = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

- If we assume that the serial fraction is fixed, then the speedup for infinite processors is limited by $\frac{1}{\alpha}$.

- For example, if $\alpha = 10\%$, then the maximum speedup is 10, even if we use an infinite number of processors.

Comments on Amdahl's Law

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

- An effective parallel algorithm has:

$$\alpha(n, p) \rightarrow 0 \quad \text{as} \quad n \rightarrow \infty$$

- For such a case, even if one fixes p, the number of processors, we can get linear speedups by choosing a suitably large problem size

$$S_p = \frac{T_s}{T_p} = \frac{p}{1 + (p-1)\alpha(n, p)} \rightarrow p \quad \text{as} \quad n \rightarrow \infty$$

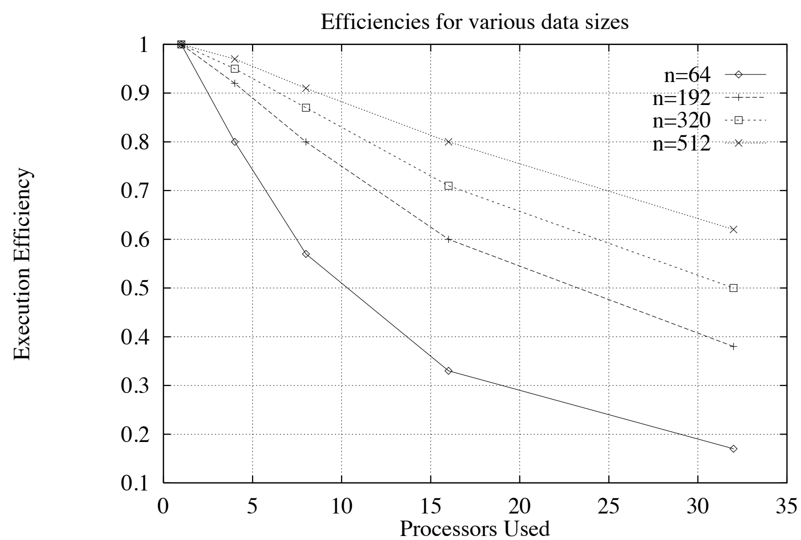
- Practically, the problem size that we can run for a particular problem is limited by the memory of the parallel computer

Scalability

- Scalability of an algorithm is a measure of its capacity to increase speedup in proportion to increase in the number of processors
- Assume a problem of size n gives a speedup of x on p processors.
- If the system size is doubled to $2p$ processors, a scalable algorithm will give a speedup of $2x$ for a problem size $m \geq n$
- Alternately, scalability is the ability of an algorithm to maintain efficiency at a fixed value by increasing the problem size simultaneously with system size

Scalability

- Efficiency (S_p/p) of adding n numbers in parallel

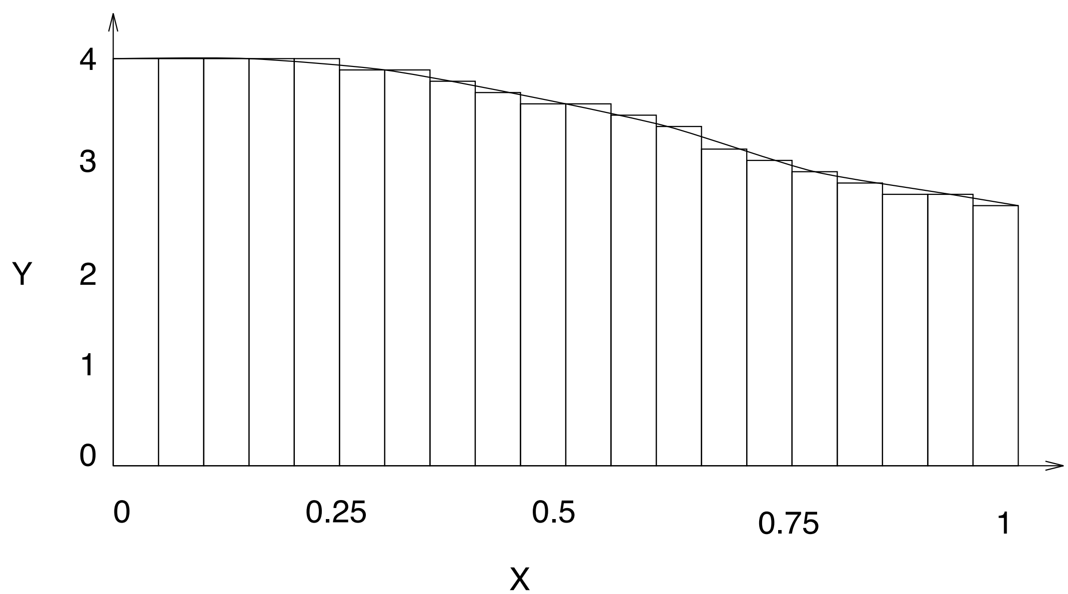


  - For an efficiency of 0.80 on 4 processors, $n = 64$
  - For an efficiency of 0.80 on 8 processors, $n = 192$
  - For an efficiency of 0.80 on 16 processors, $n = 512$

Compute Pi: Problem

- Consider parallel algorithm for computing the value of $\pi = 3.1416...$ through the following numerical integration

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



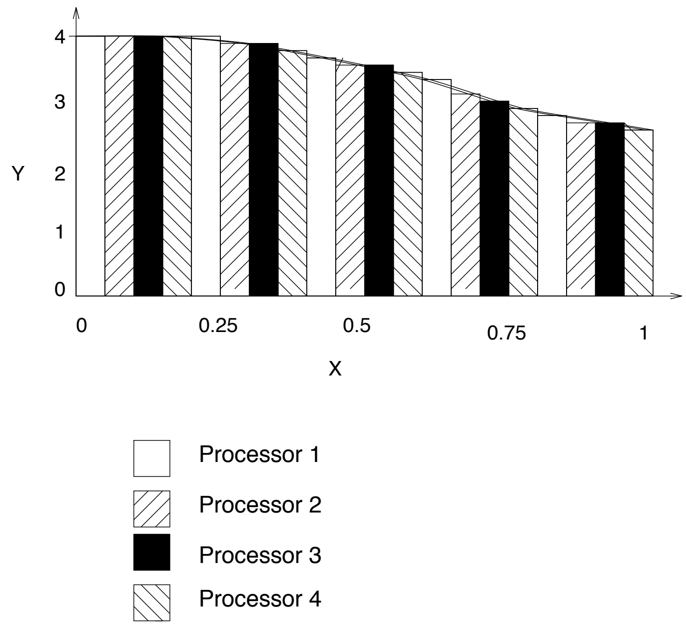
Compute Pi: Sequential Algorithm

```
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 Pi: 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 Pi: Parallel Algorithm

```
computepi()
{
   id = my_processor_id();
   nprocs = number_of_processors();
   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 Pi: Analysis

- Assume that the computation of π is performed over n points

- The sequential algorithm performs six operations (two multiplications, one division, three addition) per point on the X axis. Hence, for n points, the number of operations executed in the sequential algorithm is:

$$T_s = 6n$$

- The parallel algorithm uses p processors with static interleaved scheduling. Each processor computes on a set of m points which are allocated to each processor 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:

$$T_p = 6m = 6\frac{n}{p} + 6$$

Compute Pi: 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 including the parallel computation and the combining is:

$$T_p \; = \; 6\frac{n}{p} + 6 + \; \log(p)$$

- The speedup of the parallel algorithm is:

$$S_p \; = \; \frac{T_s}{T_p} \; = \; \frac{6n}{6\frac{n}{p} + 6 + \log(p)}$$

Compute Pi: Analysis

- The Amdahl's fraction for this parallel algorithm can be determined by rewriting the previous equation as:

$$S_p = \frac{p}{1 + \frac{p}{n} + \frac{p\log(p)}{6n}} \Rightarrow S_p = \frac{p}{1 + (p-1)\alpha(n,p)}$$

- Hence, the Amdahl's fraction $\alpha(n,p)$ is:

$$\alpha(n,p) = \frac{p}{(p-1)n} + \frac{p\log(p)}{6n(p-1)}$$

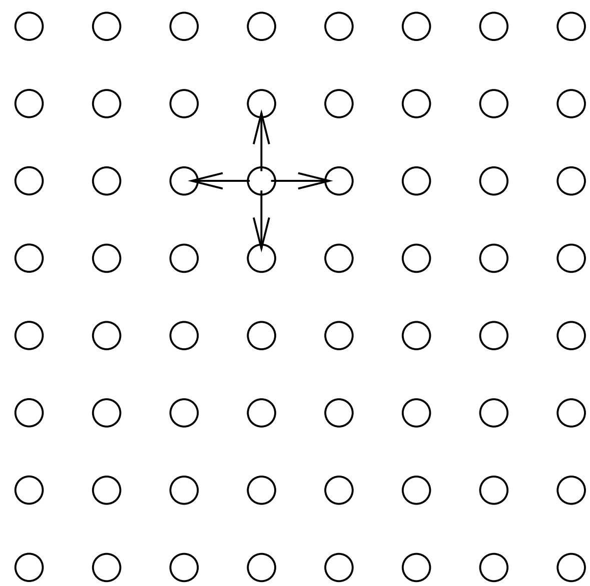
- The parallel algorithm is effective because:

$$\alpha(n,p) \rightarrow 0 \quad \text{as} \quad n \rightarrow \infty \quad \text{for fixed} \quad p$$

Finite Differences: Problem

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

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

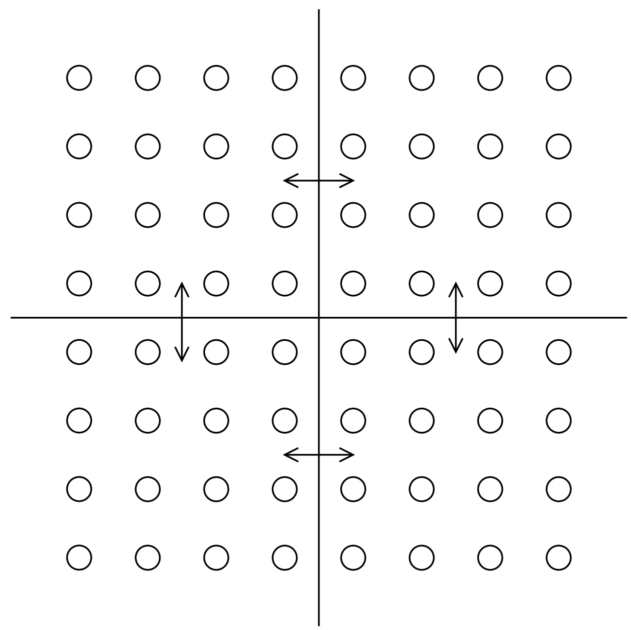


Finite Differences: Serial Algorithm

-
```
finitediff()
{
   for (t=0;t<T;t++) {
     for (i=0;i<n;i++) {
       for (j=0;j<n;j++) {
         x_new[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];
       }
     }
     swap (x_new, x);
   }
}
```

Finite Differences: Parallel Algorithm

- Each processor computes on a sub-grid of $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ points
- Sychronization 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 = number_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_new[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(); swap (x_new, x);
     }
  }
  ```
  ```

Finite Differences: Analysis

- The sequential algorithm performs six operations (two multiplications, four additions) every iteration per point on the grid. Hence, for an $n \times n$ grid and T iterations, the number of operations executed in the sequential algorithm is:

$$T_s \;=\; 6n^2T$$

- The parallel algorithm uses p processors with static blockwise scheduling. Each processor computes on an $m \times m$ sub-grid allocated to each processor in a blockwise manner

- The expression for m is given by $m \leq \lceil \frac{n}{\sqrt{p}} \rceil$. The runtime for the parallel algorithm for the parallel computation of the local values of points on an $m \times m$ sub-grid for T iterations is:

$$T_p \;=\; 6m^2T \;=\; 6(\lceil \frac{n}{\sqrt{p}} \rceil)^2T$$

Finite Differences: Analysis

- The barrier synchronization needed for each iteration can be optimally performed in $\log(p)$ steps

- The total runtime for the parallel algorithm for the computation of finite differences is:

$$T_p \;=\; 6(\lceil \frac{n}{\sqrt{p}} \rceil)^2 T + \log(p) T \;=\; 6\frac{n^2}{p} T + \log(p) T$$

- The speedup of the parallel algorithm is:

$$S_p \;=\; \frac{T_s}{T_p} \;=\; \frac{6n^2}{6\frac{n^2}{p} + \log(p)}$$

Finite Differences: Analysis

- The Amdahl's fraction for this parallel algorithm can be determined by rewriting the previous equation as:

$$S_p = \frac{p}{1 + \frac{p \log(p)}{6n^2}} \Rightarrow S_p = \frac{p}{1 + (p-1)\alpha(n,p)}$$

- Hence, the Amdahl's fraction $\alpha(n,p)$ is:

$$\alpha(n,p) = \frac{p \log(p)}{(p-1)6n^2}$$

- We finally note that

$$\alpha(n,p) \rightarrow 0 \quad \text{as} \quad n \rightarrow \infty \quad \text{for fixed} \quad p$$

- Hence, the parallel algorithm is effective

Summary

- What is parallel programming
- Why parallel programming
- Basics of parallel programs
- Examples of parallel programs
- NEXT CLASS : Architectural Features