Parallel Programs

- Conditions of Parallelism:
  - Data Dependence
  - Control Dependence
  - Resource Dependence
  - Bernstein’s Conditions
- Asymptotic Notations for Algorithm Analysis
- Parallel Random-Access Machine (PRAM)
  - Example: sum algorithm on P processor PRAM
- Network Model of Message-Passing Multicomputers
  - Example: Asynchronous Matrix Vector Product on a Ring
- Levels of Parallelism in Program Execution
- Hardware Vs. Software Parallelism
- Parallel Task Grain Size
- Example Motivating Problems With high levels of concurrency
- Limited Concurrency: Amdahl’s Law
- Parallel Performance Metrics: Degree of Parallelism (DOP)
- Concurrency Profile
- Steps in Creating a Parallel Program:
  - Decomposition, Assignment, Orchestration, Mapping
  - Program Partitioning Example
  - Static Multiprocessor Scheduling Example

(PCA Chapter 2.1, 2.2)
Parallel Programs: Definitions

- A parallel program is comprised of a number of tasks running as threads on a number of processing elements as part of a single parallel computation.

- **Task:**
  - Arbitrary piece of undecomposed work in parallel computation
  - Executed sequentially on a single processor; concurrency in parallel computation is only across tasks.

- **Parallel or Independent Tasks:**
  - Tasks that with no dependencies among them and thus can run in parallel on different processing elements.

- **Parallel Task Grain Size:** The amount of computations in a task.

- **Process (thread):**
  - Abstract entity that performs the computations assigned to a task
  - Processes communicate and synchronize to perform their tasks

- **Processor or (Processing Element):**
  - Physical computing engine on which a process executes sequentially
  - Processes virtualize machine to programmer
    - First write program in terms of processes, then map to processors
Dependency Analysis & Conditions of Parallelism

- **Dependency analysis** is concerned with detecting the presence and type of dependency between tasks that prevent tasks from being independent and from running in parallel on different processors and can be applied to tasks of any grain size.

- Dependencies between tasks can be algorithm/program related or hardware resource related.

- **Algorithm/program Task Dependencies:**
  - **Data Dependence:**
    - True Data or Flow Dependence
    - Antidependence
    - Output dependence
  - **Control Dependence**

- **Hardware/Architecture Resource Dependence**
Conditions of Parallelism: Data Dependence

Assume task S2 follows task S1 in sequential program order

1. **True Data or Flow Dependence:** Task S2 is data dependent on task S1 if an execution path exists from S1 to S2 and if at least one output variable of S1 feeds in as an input operand used by S2 denoted by $S1 \rightarrow S2$

2. **Antidependence:** Task S2 is antidependent on S1, if S2 follows S1 in program order and if the output of S2 overlaps the input of S1 denoted by $S1 \leftarrow S2$

3. **Output dependence:** Two tasks S1, S2 are output dependent if they produce the same output variable denoted by $S1 \circ \rightarrow S2$
Data Dependence Example

Here each instruction is treated as a task

S1: Load R1, A / R1 ← Memory(A) /
S2: Add R2, R1 / R2 ← R1 + R2 /
S3: Move R1, R3 / R1 ← R3 /
S4: Store B, R1 /Memory(B) ← R1 /

True Date Dependence:
(S1, S2) (S3, S4)

Output Dependence:
(S1, S3)

Anti-dependence:
(S2, S3)

Dependence graph
Can instruction 4 (second L.D) be moved just after instruction 1 (first L.D)?
If not what dependencies are violated?

Can instruction 3 (first S.D) be moved just after instruction 4 (second L.D)?
How about moving 3 after 5 (the second ADD.D)?
If not what dependencies are violated?

Data Dependence Example

(From 551)
Conditions of Parallelism

• **Control Dependence:**
  - Order of execution cannot be determined before runtime due to conditional statements.

• **Resource Dependence:**
  - Concerned with conflicts in using shared resources among parallel tasks, including:
    - Functional units (integer, floating point), memory areas, communication links etc.

• **Bernstein’s Conditions:**
  Two processes $P_1$, $P_2$ with input sets $I_1$, $I_2$ and output sets $O_1$, $O_2$ can execute in parallel (denoted by $P_1 \parallel P_2$) if:
  
  $I_1 \cap O_2 = \emptyset$
  
  $I_2 \cap O_1 = \emptyset$
  
  $O_1 \cap O_2 = \emptyset$
Bernstein’s Conditions: An Example

- For the following instructions $P_1, P_2, P_3, P_4, P_5$:
  - Each instruction requires one step to execute
  - Two adders are available

$P_1 : C = D \times E$
$P_2 : M = G + C$
$P_3 : A = B + C$
$P_4 : C = L + M$
$P_5 : F = G \div E$

Using Bernstein’s Conditions after checking statement pairs:

$P_1 \parallel P_5, \quad P_2 \parallel P_3, \quad P_2 \parallel P_5, \quad P_3 \parallel P_5, \quad P_4 \parallel P_5$

Dependence graph:
- Data dependence (solid lines)
- Resource dependence (dashed lines)

Parallel execution in three steps assuming two adders are available per step
Asymptotic Notations for Algorithm Analysis

- Asymptotic analysis of computing time complexity of an algorithm $T(n)= f(n)$ ignores constant execution factors and concentrates on:
  - Determining the order of magnitude of algorithm performance.
  - How quickly does the running time grow as a function of the input size.

- We can compare algorithms based on their asymptotic behavior and select the one with lowest rate of growth of complexity in terms of input size or problem size $n$ independent of the computer hardware.

♦ **Upper bound: Order Notation (Big Oh)**
Used in worst case analysis of algorithm performance.

\[
f(n) = O(g(n))\]

iff there exist two positive constants $c$ and $n_0$ such that
\[
|f(n)| \leq c |g(n)| \quad \text{for all } n > n_0
\]

$\Rightarrow$ i.e. $g(n)$ is an upper bound on $f(n)$

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$
Asymptotic Notations for Algorithm Analysis

♦ **Asymptotic Lower bound:** Big Omega Notation
   
   Used in the analysis of the lower limit of algorithm performance

   \[ f(n) = \Omega(g(n)) \]

   if there exist positive constants \( c, n_0 \) such that
   \[ |f(n)| \geq c|g(n)| \quad \text{for all} \quad n > n_0 \]

   \[ \Rightarrow \quad \text{i.e.} \quad g(n) \text{ is a lower bound on } f(n) \]

♦ **Asymptotic Tight bound:** Big Theta Notation
   
   Used in finding a tight limit on algorithm performance

   \[ f(n) = \Theta(g(n)) \]

   if there exist constant positive integers \( c_1, c_2, \) and \( n_0 \) such that
   \[ c_1|g(n)| \leq |f(n)| \leq c_2|g(n)| \quad \text{for all} \quad n > n_0 \]

   \[ \Rightarrow \quad \text{i.e.} \quad g(n) \text{ is both an upper and lower bound on } f(n) \]
Graphs of $O$, $\Omega$, $\Theta$

- $f(n) = O(g(n))$  \textit{Upper Bound}
- $f(n) = \Omega(g(n))$  \textit{Lower Bound}
- $f(n) = \Theta(g(n))$  \textit{Tight bound}
# Rate of Growth of Common Computing Time Functions

<table>
<thead>
<tr>
<th>$\log n$</th>
<th>$n$</th>
<th>$n \log n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>4096</td>
<td>65536</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>160</td>
<td>1024</td>
<td>32768</td>
<td>4294967296</td>
</tr>
</tbody>
</table>
Rate of Growth of Common Computing Time Functions

- $2^n$
- $n^2$
- $n\log(n)$
- $n$
- $\log(n)$
Theoretical Models of Parallel Computers

- **Parallel Random-Access Machine (PRAM):**
  - \( n \) processor, global shared memory model.
  - Models idealized parallel computers with zero synchronization or memory access overhead.
  - Utilized parallel algorithm development and scalability and complexity analysis.

- **PRAM variants:** More realistic models than pure PRAM
  - **EREW-PRAM:** Simultaneous memory reads or writes to/from the same memory location are not allowed.
  - **CREW-PRAM:** Simultaneous memory writes to the same location is not allowed. (Better to model SAS MIMD?)
  - **ERCW-PRAM:** Simultaneous reads from the same memory location are not allowed.
  - **CRCW-PRAM:** Concurrent reads or writes to/from the same memory location are allowed.
Example: sum algorithm on P processor PRAM

begin
1. for j = 1 to l ( = n/p) do
   Set B(l(s - 1) + j): = A(l(s-1) + j)
2. for h = 1 to log n do
   2.1 if (k- h - q ≥ 0) then
      for j = 2^{k-h-q}(s-1) + 1 to 2^{k-h-q}s do
         Set B(j): = B(2j - 1) + B(2s)
   2.2 else {if (s ≤ 2^{k-h}) then
         Set B(s): = B(2s - 1) + B(2s)}
3. if (s = 1) then set S: = B(1)
end

Running time analysis:
• Step 1: takes O(n/p) each processor executes n/p operations
• The hth of step 2 takes O(n / (2^h p)) since each processor has
   to perform (n / (2^h p)) operations
• Step three takes O(1)
• Total Running time: $T_p(n) = O\left(\frac{n}{p} + \sum_{h=1}^{\log n} \left[ \frac{n}{2^h p} \right]\right) = O\left(\frac{n}{p} + \log n\right)$
Example: Sum Algorithm on P Processor PRAM

\[ T = 2 + \log_2(8) = 2 + 3 = 5 \]

For \( n = 8 \) \( p = 4 \)

Processor allocation for computing the sum of 8 elements on 4 processor PRAM

Operation represented by a node is executed by the processor indicated below the node.

\[ S = B(1) \]

\[ T = O(\log_2 n) \]

Cost = \( O(n \log_2 n) \)

\[ T = O(\log_2 n) \]

\[ \text{Cost} = O(n \log_2 n) \]
Performance of Parallel Algorithms

- Performance of a parallel algorithm is typically measured in terms of worst-case analysis.

- For problem Q with a PRAM algorithm that runs in time $T(n)$ using $P(n)$ processors, for an instance size of $n$:
  - The time-processor product $C(n) = T(n) \cdot P(n)$ represents the cost of the parallel algorithm.
  - For $P \leq P(n)$, each of the $T(n)$ parallel steps is simulated in $O(P(n)/p)$ substeps. Total simulation takes $O(T(n)P(n)/p)$
  - The following four measures of performance are asymptotically equivalent:
    - $P(n)$ processors and $T(n)$ time
    - $C(n) = P(n)T(n)$ cost and $T(n)$ time
    - $O(T(n)P(n)/p)$ time for any number of processors $p \leq P(n)$
    - $O(C(n)/p + T(n))$ time for any number of processors.
Matrix Multiplication On PRAM

- Multiply matrices $A \times B = C$ of sizes $n \times n$
- Sequential Matrix multiplication:
  
  For $i = 1$ to $n$ {
    
    For $i = 1$ to $n$ {
      $C(i, j) = \sum_{i=1}^{n} a(i, t) \times b(t, j)$
    }
  }

  Sequential Matrix multiplication time complexity $O(n^3)$
- Matrix multiplication on PRAM with $p = n^3$ processors.
  - Compute in parallel for all $i, j, t = 1$ to $n$
    
    $c(i,j,t) = a(i, t) \times b(t, j)$
    \[ O(1) \]
  - Compute in parallel for all $i; j = 1$ to $n$:
    \[ C(i, j) = \sum_{t=1}^{n} c(i, j, t) \quad O(\log_2 n) \]

  Thus time complexity of matrix multiplication on PRAM with $n^3$ processors = $O(\log_2 n)$  \[ \text{Cost}(n) = O(n^3 \log_2 n) \]
  - Time complexity of matrix multiplication on PRAM with $n^2$ processors = $O(n \log_2 n)$
  - Time complexity of matrix multiplication on PRAM with $n$ processors = $O(n^2 \log_2 n)$
The Power of The PRAM Model

- Well-developed techniques and algorithms to handle many computational problems exist for the PRAM model.

- Removes algorithmic details regarding synchronization and communication, concentrating on the structural and fundamental data dependency properties of the algorithm.

- Captures several important parameters of parallel computations. Operations performed in unit time, as well as processor allocation.

- The PRAM design paradigms are robust and many network algorithms can be directly derived from PRAM algorithms.

- It is possible to incorporate synchronization and communication into the shared-memory PRAM model.
Network Model of Message-Passing Multicomputers

- A network of processors can be viewed as a graph $G (N,E)$
  - Each node $i \in N$ represents a processor
  - Each edge $(i,j) \in E$ represents a two-way communication link between processors $i$ and $j$.
  - Each processor is assumed to have its own local memory.
  - No shared memory is available.
  - Operation is synchronous or asynchronous (message passing).
  - Typical message-passing communication constructs:
    - $\text{send}(X,i)$ a copy of $X$ is sent to processor $P_i$, execution continues.
    - $\text{receive}(Y, j)$ execution suspended until the data from processor $P_j$ is received and stored in $Y$ then execution resumes.
Network Model of Multicomputers

- **Routing** is concerned with delivering each message from source to destination over the network.

- Additional important network topology parameters:
  - The **network diameter** is the maximum distance between any pair of nodes.
  - The **maximum degree** of any node in G

- Example:
  - **Linear array**: P processors $P_1, ..., P_p$ are connected in a linear array where:
    - Processor $P_i$ is connected to $P_{i-1}$ and $P_{i+1}$ if they exist.
    - Diameter is $p-1$; maximum degree is 2 (1 or 2).
  
  - **A ring** is a linear array of processors where processors $P_1$ and $P_p$ are directly connected. Degree = 2, Diameter = $p/2$.
A Four-Dimensional Hypercube

- In a d-dimensional binary hypercube, each node is assigned a d-bit address.
- Two processors are connected if their binary addresses differ in one bit position.
- Degree = Diameter = d

Here \( d = 4 \)
Example: Asynchronous Matrix Vector Product on a Ring

• **Input:**
  - \( n \times n \) matrix \( A \); vector \( x \) of order \( n \)
  - The processor number \( i \). The number of processors
  - The \( i \)th submatrix \( B = A(1:n, (i-1)r + 1 ; ir) \) of size \( n \times r \) where \( r = n/p \)
  - The \( i \)th subvector \( w = x(i - 1)r + 1 : ir \) of size \( r \)

• **Output:**
  - Processor \( P_i \) computes the vector \( y = A_1x_1 + \ldots A_i x_i \) and passes the result to the right
  - Upon completion \( P_1 \) will hold the product \( Ax \)

Begin

1. Compute the matrix vector product \( z = Bw \)
2. If \( i = 1 \) then set \( y := 0 \)
   
   else receive\((y, \text{left})\)
3. Set \( y := y + z \)
4. send\((y, \text{right})\)
5. if \( i = 1 \) then receive\((y, \text{left})\)

End

\[
T_{\text{comp}} = k(n^2/p) \\
T_{\text{comm}} = p(l+ mn) \\
T = T_{\text{comp}} + T_{\text{comm}} \\
= k(n^2/p) + p(l+ mn)
\]

\( k, l, m \) constants
Matrix Vector Product $y = Ax$ on a Ring

- For each processor $i$ compute in parallel
  \[ Z_i = B_i w_i \]

\[ Y = Ax = Z_1 + Z_2 + \ldots + Z_p = B_1 w_1 + B_2 w_2 + \ldots + B_p w_p \]

- For processor #1: set $y = 0$, $y = y + Z_i$, send $y$ to right processor
- For every processor except #1
  Receive $y$ from left processor, $y = y + Z_i$, send $y$ to right processor
- For processor #1 receive final result from left processor $p$
- $T = O(n^2/p + pn)$
Creating a Parallel Program

• Assumption: Sequential algorithm to solve problem is given
  – Or a different algorithm with more inherent parallelism is devised.
  – Most programming problems have several parallel solutions or algorithms. The best solution may differ from that suggested by existing sequential algorithms.

One must:
  – Identify work that can be done in parallel
  – Partition work and perhaps data among processes
  – Manage data access, communication and synchronization
  – Note: work includes computation, data access and I/O

Main goal: Speedup (plus low programming effort and resource needs)

\[
\text{Speedup} (p) = \frac{\text{Performance}(p)}{\text{Performance}(1)}
\]

For a fixed problem:

\[
\text{Speedup} (p) = \frac{\text{Time}(1)}{\text{Time}(p)}
\]
Levels of Parallelism in Program Execution

- **Level 5**: Jobs or programs (Multiprogramming)
- **Level 4**: Subprograms, job steps or related parts of a program
- **Level 3**: Procedures, subroutines, or co-routines
- **Level 2**: Non-recursive loops or unfolded iterations
- **Level 1**: Instructions or statements

Increasing communications demand and mapping/scheduling overhead leads to higher degree of parallelism:
- Coarse Grain
- Medium Grain
- Fine Grain
Hardware and Software Parallelism

• **Hardware parallelism:**
  – Defined by machine architecture, hardware multiplicity (number of processors available) and connectivity.
  – Often a function of cost/performance tradeoffs.
  – Characterized in a single processor by the number of instructions $k$ issued in a single cycle ($k$-issue processor).
  – A multiprocessor system with $n$ $k$-issue processor can handle a maximum limit of $nk$ parallel instructions.

• **Software parallelism:**
  – Defined by the control and data dependence of programs.
  – Revealed in program profiling or program flow graph.
  – A function of algorithm, parallel task grain size, programming style and compiler optimization.
Computational Parallelism and Grain Size

- Task grain size (granularity) is a measure of the amount of computation involved in a task in parallel computation:
  - **Instruction Level (Fine Grain Parallelism):**
    - At instruction or statement level.
    - 20 instructions grain size or less.
    - For scientific applications, parallelism at this level range from 500 to 3000 concurrent statements
    - Manual parallelism detection is difficult but assisted by parallelizing compilers.
  - **Loop level (Fine Grain Parallelism):**
    - Iterative loop operations.
    - Typically, 500 instructions or less per iteration.
    - Optimized on vector parallel computers.
    - Independent successive loop operations can be vectorized or run in SIMD mode.
Computational Parallelism and Grain Size

- **Procedure level (Medium Grain Parallelism):**
  - Procedure, subroutine levels.
  - Less than 2000 instructions.
  - More difficult detection of parallel than finer-grain levels.
  - Less communication requirements than fine-grain parallelism.
  - Relies heavily on effective operating system support.

- **Subprogram level (Coarse Grain Parallelism):**
  - Job and subprogram level.
  - Thousands of instructions per grain.
  - Often scheduled on message-passing multicomputers.

- **Job (program) level, or Multiprogramming:**
  - Independent programs executed on a parallel computer.
  - Grain size in tens of thousands of instructions.
Example Motivating Problem: Simulating Ocean Currents/Heat Transfer...

Model as two-dimensional grids
Discretize in space and time
  • finer spatial and temporal resolution => greater accuracy
Many different computations per time step
  • set up and solve equations iteratively (Gauss-Seidel)
Concurrency across and within grid computations per iteration
  • \( n^2 \) parallel computations per grid x number of grids

Expression for updating each interior point:
\[
\]

Concurrency: \( n^2 \) data parallel computations per grid
Limited Concurrency: Amdahl’s Law

– Most fundamental limitation on parallel speedup.
– Assume a fraction $s$ of sequential execution time on a single processor cannot be parallelized.
– Assuming that the problem size remains fixed and that the remaining fraction $(1-s)$ can be parallelized without any parallelization overheads to run on $p$ processors and thus reduced by a factor of $p$.
– The resulting speedup for $p$ processors:

$$\text{Speedup}(p) = \frac{\text{Sequential Execution Time}}{\text{Parallel Execution Time}}$$

Parallel Execution Time = $(S + (1-S)/P) \times \text{Sequential Execution Time}$

$$\text{Speedup}(p) = \frac{\text{Sequential Execution Time}}{(1 - F) + F/S) \times \text{Sequential Execution Time}} = \frac{1}{s + (1-s)/p}$$

– Thus for a fixed problem size, if fraction $s$ of sequential execution is inherently serial, speedup $\leq 1/s$
Amdahl’s Law Example

- Example: 2-phase calculation
  - sweep over $n$-by-$n$ grid and do some independent computation
  - sweep again and add each value to global sum sequentially
  - Time for first phase = $n^2/p$
  - Second phase serialized at global variable, so time = $n^2$

  \[
  \text{Speedup} \leq \frac{2n^2}{n^2/p + n^2} = \frac{1}{0.5/p + 0.5} \quad \text{or at most } 1/s = 1/0.5 = 2
  \]

- Possible Trick: divide second phase into two
  - Accumulate into private sum during sweep
  - Add per-process private sum into global sum
  - Parallel time is $n^2/p + n^2/p + p$, and speedup at best \[
  \frac{2n^2}{2n^2 + p^2}
  \]
Amdahl’s Law Example: A Pictorial Depiction

### Phase 1
- **Sequential Execution**
- **Parallel Execution**
  - Work done concurrently on p processors
  - Phase 1 time reduced by p
  - Phase 2 sequential

### Phase 2
- **Sequential Execution**
- **Parallel Execution**
  - Work done concurrently on p processors
  - Phase 1 time reduced by p
  - Phase 2 divided into two steps

#### Speedup Formulas
- **Maximum Possible Speedup**: $s = 0.5$
- **Speedup**:
  - $\frac{1}{0.5 + \frac{p}{p}}$
  - $\frac{2n^2}{2n^2 + p^2}$
Parallel Performance Metrics

Degree of Parallelism (DOP)

- For a given time period, DOP reflects the number of processors in a specific parallel computer actually executing a particular parallel program.

- **Average Parallelism A:**
  - given maximum parallelism = m
  - n homogeneous processors
  - computing capacity of a single processor $\Delta$
  - Total amount of work $W$ (instructions, computations):

\[
W = \Delta \int_{t_1}^{t_2} DOP(t) \, dt \quad \text{or as a discrete summation} \quad W = \Delta \sum_{i=1}^{m} i \cdot t_i
\]

Where $t_i$ is the total time that DOP = i and $\sum_{i=1}^{m} t_i = t_2 - t_1$

The average parallelism A:

\[
A = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} DOP(t) \, dt \quad \text{In discrete form} \quad A = \left( \sum_{i=1}^{m} i \cdot t_i \right) / \left( \sum_{i=1}^{m} t_i \right)
\]

Computations/sec

DOP Area

EECC756 - Shaaban
Example: Concurrency Profile of A Divide-and-Conquer Algorithm

- Execution observed from \( t_1 = 2 \) to \( t_2 = 27 \)
- Peak parallelism \( m = 8 \)
- \( A = \frac{\sum_{i=1}^{m} i \cdot t_i}{\sum_{i=1}^{m} t_i} \)

\[
A = \frac{1 \cdot 5 + 2 \cdot 3 + 3 \cdot 4 + 4 \cdot 6 + 5 \cdot 2 + 6 \cdot 2 + 8 \cdot 3}{5 + 3 + 4 + 6 + 2 + 2 + 3} = \frac{93}{25} = 3.72
\]

Degree of Parallelism (DOP)
Concurrency Profile & Speedup

For a parallel program DOP may range from 1 (serial) to a maximum $m$

- Area under curve is total work done, or time with 1 processor
- Horizontal extent is lower bound on time (infinite processors)
- Speedup is the ratio:
  $$\frac{\sum_{k=1}^{\infty} f_k k}{\sum_{k=1}^{\infty} f_k \left\lfloor \frac{k}{p} \right\rfloor}$$
  base case: $$\frac{1}{s + \frac{1-s}{p}}$$
  
  $k =$ degree of parallelism $f_k =$ Total time with degree of parallelism $k$

- Amdahl’s law applies to any overhead, not just limited concurrency.
Parallel Performance Example

- The execution time $T$ for three parallel programs is given in terms of processor count $P$ and problem size $N$.
- In each case, we assume that the total computation work performed by an optimal sequential algorithm scales as $N+N^2$.

1. For first parallel algorithm: $T = N + \frac{N^2}{P}$
   - This algorithm partitions the computationally demanding $O(N^2)$ component of the algorithm but replicates the $O(N)$ component on every processor. There are no other sources of overhead.

2. For the second parallel algorithm: $T = \frac{(N+N^2)}{P} + 100$
   - This algorithm optimally divides all the computation among all processors but introduces an additional cost of 100.

3. For the third parallel algorithm: $T = \frac{(N+N^2)}{P} + 0.6P^2$
   - This algorithm also partitions all the computation optimally but introduces an additional cost of $0.6P^2$.

- All three algorithms achieve a speedup of about 10.8 when $P = 12$ and $N=100$. However, they behave differently in other situations as shown next.
- With $N=100$, all three algorithms perform poorly for larger $P$, although Algorithm (3) does noticeably worse than the other two.
- When $N=1000$, Algorithm (2) is much better than Algorithm (1) for larger $P$.

EECC756 - Shaaban

#37  lec # 3  Spring2004  3-16-2004
All algorithms achieve:
Speedup = 10.8 when P = 12 and N=100

Algorithm 1: \( T = \frac{N + N^2}{P} \)
Algorithm 2: \( T = \frac{(N+N^2)}{P} + 100 \)
Algorithm 3: \( T = \frac{(N+N^2)}{P} + 0.6P^2 \)

N=1000, Algorithm (2) performs much better than Algorithm (1) for larger P.

Steps in Creating a Parallel Program

- **4 steps:**
  - Decomposition, Assignment, Orchestration, Mapping
  - Done by programmer or system software (compiler, runtime, ...)
  - Issues are the same, so assume programmer does it all explicitly

+ Scheduling
Decomposition

• Break up computation into concurrent tasks to be divided among processes:
  – Tasks may become available dynamically.
  – No. of available tasks may vary with time.
  – Together with assignment, also called *partitioning*.

  i.e. **identify concurrency** and **decide level at which to exploit it**.

• **Grain-size problem:**
  – To determine the number and size of grains or tasks in a parallel program.
  – Problem and machine-dependent.
  – Solutions involve *tradeoffs* between parallelism, communication and scheduling/synchronization overheads.

• **Grain packing:**
  – To combine multiple fine-grain nodes into a coarse grain node (task) to reduce communication delays and overall scheduling overhead.

**Goal:** Enough tasks to keep processes busy, but not too many.
  – No. of tasks available at a time is upper bound on achievable speedup
Assignment

• Specifying mechanisms to divide work up among processes:
  – Together with decomposition, also called *partitioning*.
  – Balance workload, reduce communication and management cost
    • May involve duplicating computation to reduce communication cost.

• Partitioning problem:
  – To partition a program into parallel branches, modules to give
    the shortest possible execution on a specific parallel architecture.

• Structured approaches usually work well:
  – Code inspection (parallel loops) or understanding of application.
  – Well-known heuristics.
  – *Static* versus *dynamic* assignment.

• As programmers, we worry about partitioning first:
  – *Usually* independent of architecture or programming model.
  – But cost and complexity of using primitives may affect decisions.
Orchestration

– Naming data.
– Structuring communication (using communication primitives)
– Synchronization (ordering using synchronization primitives).
– Organizing data structures and scheduling tasks temporally.

• Goals
  – Reduce cost of communication and synch. as seen by processors
  – Reserve locality of data reference (incl. data structure organization)
  – Schedule tasks to satisfy dependences early
  – Reduce overhead of parallelism management

• Closest to architecture (and programming model & language).
  – Choices depend a lot on comm. abstraction, efficiency of primitives.
  – Architects should provide appropriate primitives efficiently.
Mapping/Scheduling

• Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs.

• Mapping can be specified statically or determined at runtime by load-balancing algorithms (dynamic scheduling).

• Two aspects of mapping:
  – Which processes will run on the same processor, if necessary
  – Which process runs on which particular processor
    • mapping to a network topology/account for NUMA

• One extreme: space-sharing
  – Machine divided into subsets, only one app at a time in a subset
  – Processes can be pinned to processors, or left to OS.

• Another extreme: complete resource management control to OS
  – OS uses the performance techniques we will discuss later.

• Real world is between the two.
  – User specifies desires in some aspects, system may ignore
Program Partitioning Example

Example 2.4 page 64
Fig 2.6 page 65
Fig 2.7 page 66
In Advanced Computer Architecture, Hwang
(see handout)
Static Multiprocessor Scheduling

Dynamic multiprocessor scheduling is an NP-hard problem.

Node Duplication: to eliminate idle time and communication delays, some nodes may be duplicated in more than one processor.

Fig. 2.8 page 67

Example: 2.5 page 68
In Advanced Computer Architecture, Hwang
(see handout)
<table>
<thead>
<tr>
<th>Step</th>
<th>Architecture-Dependent?</th>
<th>Major Performance Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition</td>
<td>Mostly no</td>
<td>Expose enough concurrency but not too much</td>
</tr>
<tr>
<td>Assignment</td>
<td>Mostly no</td>
<td>Balance workload</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce communication volume</td>
</tr>
<tr>
<td>Orchestration</td>
<td>Yes</td>
<td>Reduce noninherent communication via data locality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce communication and synchronization cost</td>
</tr>
<tr>
<td></td>
<td></td>
<td>as seen by the processor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce serialization at shared resources</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Schedule tasks to satisfy dependences early</td>
</tr>
<tr>
<td>Mapping</td>
<td>Yes</td>
<td>Put related processes on the same processor if necessary</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exploit locality in network topology</td>
</tr>
</tbody>
</table>