CDP Tutorial 3
Basics of Parallel Algorithm Design

uses some of the slides for chapters 3 and 5 accompanying
“Introduction to Parallel Computing”,
http://www-users.cs.umn.edu/~karypis/parbook
Preliminaries: Decomposition, Tasks, and Dependency Graphs

- Parallel algorithm should decompose the problem into tasks that can be executed concurrently.
- A decomposition can be illustrated in the form of a directed graph (task dependency graph)

![Graph Diagram]

5 → 7 → 3 → 4 → 2
Dependency Graphs

- nodes = tasks
- edges = dependencies
- The result of one task is required for processing the next
Degree of Concurrency

- Determines the maximum amount of tasks which can indeed run in parallel
  - An upper bound on the parallel algorithm speedup

Degree of concurrency = 3
Critical Path

- The longest path in the dependency graph
- A lower bound on program runtime

Critical path = 7 + 4 + 2 = 13
Average Degree of Concurrency

- The ratio between the critical path to the sum of all the tasks
- The speed up of the parallel algorithm

Critical path = 7+4+2 = 13

Avg. Deg. of concurrency = \frac{5+7+3+4+2}{13} = 1.6
Example: Multiplying a Dense Matrix with a Vector

- Computation of each element of output vector $y$ is independent of other elements.
- Based on this, a dense matrix-vector product can be decomposed into $n$ independent tasks.
Example: Multiplying a Dense Matrix with a Vector

- While tasks share data (namely, the vector $b$), they do not have any control dependencies
  - no task needs to wait for the (partial) completion of any other
- All tasks are of the same size in terms of number of operations

Is this the maximum number of tasks we could decompose this problem into?
Multiplying a dense matrix with a vector – 2n CPUs available

On what kind of platform will we have 2N processors?
### Multiplying a dense matrix with a vector – 2n CPUs available

A matrix is divided into parts assigned to different tasks. Each task is assigned to a subset of CPUs. The matrix `A` is split into parts assigned to different tasks.

<table>
<thead>
<tr>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 2n+1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 2n-1</td>
<td>Task 2n</td>
<td>Task 3n</td>
</tr>
</tbody>
</table>

The matrix multiplication process involves assigning tasks to CPUs and distributing the computation load efficiently.
Granularity of Task Decompositions

- The size of tasks into which a problem is decomposed is called granularity.
Parallelization Limitations

- Parallel algorithm scalability factors:
  - In theory:
    - Amdahl’s law: \( \frac{1}{(1-A)+\frac{A}{N}} \) (upper bound on speedup)
  - In practice:
    - Task interaction overhead
    - CPU utilization
      - stalls due to data dependencies
      - imperfect load balancing
    - Excess computations (different algorithm, reuse of previous computation etc.)
Guidelines for good parallel algorithm design

- Maximize concurrency.
- Spread tasks evenly between processors to avoid idling and achieve load balancing.
- Execute tasks on critical path as soon as the dependencies are satisfied.
- Minimize (overlap) communication between processors.
Evaluating parallel algorithm

- **Speedup**
  - \( S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \)

- **Efficiency**
  - \( E = \frac{S}{\#\text{CPUs}} \)

- **Cost**
  - \( C = \#\text{CPUs} \cdot T_{\text{parallel}} \)

- **Scalability**
  - For a fixed task size: speedup as a function of number of processors (efficiency)
Evaluating parallel algorithm

- **Speedup**
  - \( S = \frac{T_{serial}}{T_{parallel}} \)

- **Efficiency**
  - \( E = \frac{S}{\#CPUs} \)

- **Cost**
  - \( C = \#CPUs \cdot T_{parallel} \)

- **Scalability**
  - For a fixed task size: speedup as a function of number of processors (efficiency)
Evaluating parallel algorithm

- **Speedup**
  \[ S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \]

- **Efficiency**
  \[ E = \frac{S}{\#\text{CPUs}} \]

- **Cost**
  \[ C = \#\text{CPUs} \cdot T_{\text{parallel}} \]

- **Scalability**
  For a fixed task size: speedup as a function of number of processors (efficiency)
Evaluating parallel algorithm

- **Speedup**
  - \( S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \)

- **Efficiency**
  - \( E = \frac{S}{\# \text{CPUs}} \)

- **Cost**
  - \( C = \# \text{CPUs} \cdot T_{\text{parallel}} \)

- **Scalability**
  - For a fixed task size: speedup as a function of number of processors (efficiency)
Simple example

- Summing $n$ numbers on $p$ CPUs
  - Allocating $\frac{n}{p}$ numbers for each CPU
  - Each CPU sum its portion of the numbers
  - Now we have $p$ numbers
Simple example

- Summing \( p \) numbers on \( \frac{p}{2} \) CPUs
  - Allocating 2 numbers for each CPU
  - Each CPU sum it’s numbers
  - Now we have \( \frac{p}{2} \) numbers

This is the result
Summing \( n \) numbers on \( p \) CPUs

- Serial Time = \( \Theta(n) \)

- Parallel Time = \( \Theta \left( \frac{n}{p} + \log p \right) \)

- Speedup = \( \Theta \left( \frac{n}{\frac{n}{p} + \log p} \right) \)

- Efficiency = \( \Theta \left( \frac{n}{\frac{n}{p} + \log p} \right) = \Theta \left( \frac{n}{n + p \cdot \log p} \right) \)

- Cost = \( \Theta \left( \left( \frac{n}{p} + \log p \right) \cdot p \right) = \Theta(n + p \cdot \log p) \)
Parallel multiplication of 3 matrices

- A, B, C – rectangular matrices \( N \times N \)
- We want to compute in parallel: \( A \times B \times C \)
- How do we achieve the maximum performance?
Parallel multiplication of 3 matrices

Every cell in T can be calculated independently from the others

Are the blue and yellow depended tasks?
Parallel multiplication of 3 matrices

\[ T_{1,y} \rightarrow R_{x,y} \quad T_{n,y} \rightarrow R_{u,y} \quad T_{1,v} \rightarrow R_{x,v} \quad T_{n,v} \rightarrow R_{u,v} \]

\[ X \quad = \quad C \quad R \]

\[ x \quad u \quad y \quad v \]
Dynamic Task Creation: Simple Master-Worker

- Identify independent computations
- Create a queue of tasks
- Each process picks from the queue and may insert a new task to the queue
Task Granularity

- How do we select the size of each task?
Hypothetic Implementation: Multiple Threads

- Let's assign separate thread for each task we defined before.
- How do we coordinate the threads?
  - Initialize threads to know their dependencies.
  - Build “producer-consumer” like logic for every thread.