Review

Chapter 3: How to design parallel algorithms/programs

Identifying data or task parallelism
Understand if we are dealing with arrays or streams of data and apply the same “task”
Example: Matrix algorithms

Can we divide the task into “independent” subtasks
Parallelizing of Queries

Once we decide on a decomposition, we can build task graphs
Normally we want to start with as much parallelism as possible make tasks as fine grained as possible

Homework # 2: 3.4, 3.15, 3.16
Due Feb. 4, 2019

Figure 3.5: Abstractions of the task graphs of Figures 3.2 and 3.3, respectively.
Task interaction graphs

Task interactions (or synchronizations) exist even if we have a shared memory. These interactions may cause delays because of producer-consumer dependencies or mutual exclusion.

*In the following example, each row of A and one element b are assigned to a task.*

![Task interaction graph example](image)

Mapping tasks to processors

Once we have a task graph (and/or interaction graph) we can think of assigning different tasks to different processes (on the same processor or on different processors).

Some Issues to Consider

- Size of tasks and interactions among the tasks
- Costs associated with tasks: creating (and deleting) tasks, computation performed by tasks, do we know this a priori?
- Memory (data) requirements of tasks
- Load balancing
- Interactions: communication overhead, delays due to mutual exclusion
More on techniques for decomposing a problem for parallel processing (in addition to data or task parallel decompositions)

1. Recursive decomposition
e.g., quicksort (see page 96)
   Start with one task to divide the list into two (using a pivot)
   two tasks to divide each of the two lists into 2 (total 4 lists)
   etc.
Works if you started with a recursive sequential algorithm
Later we will see some difficulties in parallelizing recursive algorithms such as depth first search.

2. Data decomposition
   Assign task to deal with different parts of data either
   based on input
e.g. Dot matrix tasks handle portions of input arrays
   based on output
e.g. matrix multiplication, different tasks compute different output elements
   hybrid
depending on intermediate data
   we will see examples later

3. Exploratory decomposition
good for optimization problems that find a solution by searching the solution space.

   Most AI problems, planning problems fall in this category.
   Consider solving the 15 puzzle problem

   We start with a 4x4 grid with one blank square (and 15 filled squares)
   The problem is to move to a target configuration of the grid from a starting configuration.

   Note at any given configuration you have at most four possible moves
   leading to (up to) 4 new paths in the search space.

4. Speculative decomposition
   Start with tasks even before knowing if they should be started

   Parallel discrete event simulation (page 109)
   Time warp simulations
   Speculative threads (in hardware or software)
Critical path in a task graph
Assigning tasks to processors → priority to tasks along the critical path

Chapter 4 Cost of Communication and Common Communication Operations
- one to one
- one to all (simple broadcast)
- all to all
- one to all but different messages (scatter)
- all to one (gather)

What are the actions performed to complete a communication?

- Prepare a message
- Ask OS to buffer the message; prepare to send on the network
- Send the message to the next neighboring node on its way
- The next node may buffer the message and then route it along the way
- Transferred from the destination node to OS buffers
- From OS buffers to application buffers
Two types of communication techniques

**Store and Forward networks:** The entire message (all packets) must be received and stored at each intermediate node before being forwarded to the next node in the path.

Advantage: Easy to manage, no need to worry about packets received out of order
Disadvantage: Slow and wasteful (buffers at intermediate nodes must be large enough to store the entire message)

**Cut-through (wormhole routing).** Message packets are forwarded immediately at intermediate nodes faster but difficult to deal with failures

*Not very common in modern systems*

- **Start-Up Cost:** Fixed one time cost ($t_s$)
- **Per Hop Cost:** Normally Fixed -- amount of time needed to reserve buffers at intermediate nodes ($t_h$). Also known as per node latency
- **Per word Cost:** Variable and depends on messages size (and bandwidth) ($t_w$)

In most systems the overhead at intermediate nodes is very small compared to the size of the message, so the total cost is simplified to:

**Total Cost = $t_s + t_w * m * l$**

where:

- $l$ is the number of hops or intermediate nodes to travel
- $m$ is the size of the message
- $t_s$ is the start-up cost
- $t_w$ is the per hop cost
- $t_h$ is the per node latency

Again if we ignore the cost at intermediate nodes $t_h$,

**Total Cost = $t_s + t_w * m$**

Cut-Through or Worm-Hole Routing. At intermediate node, we do not store the entire message -- only enough of header information to know which route to use.

**Total Cost = $t_s + t_w * m + t_h * l$**
CSCE 5160 Parallel Processing

Most algorithms will involve one of the following forms of communication

- one-to-one
- one-to-small group
- one-to-all broadcast
- all-to-all broadcast
- all-to-one receive
- all-to-all receive

We need to analyze the cost of these forms of communication for different interconnection networks

- 2-D mesh
- hypercube
- tree
- ring

We will use store-and-forward and use \( t_s + t_w \cdot m \cdot l \) as cost of sending one message of size \( m \)

CSCE 5160 Parallel Processing

Assumptions for the algorithms presented

1. Mesh networks are assumed to have wrap-around connections
2. Links are bidirectional
3. A processor can send only one message at a time on its links
4. A processor can only receive one message at a time on its links

Consider sending one-to-one message in a 2-D mesh with wrap-around

The communication cost = \( t_s + t_w \cdot m \cdot l \)

What is the upper bound?

With \( \sqrt[4]{p} \cdot \sqrt[4]{p} \) mesh, the longest distance a message has to travel is \( 2 \cdot (\sqrt[4]{p} / 2) \)

So the upper bound on communication cost = \( t_s + t_w \cdot m \cdot 2 \cdot (\sqrt[4]{p} / 2) \)

or \( O(p^{1/2}) \)

For Hypercube with \( p \) processors?

\[ t_s + t_w \cdot m \cdot \log_2(p) \quad \text{or} \quad O(\log(p)) \]
Broadcast. An easy way to send a message from a source to all other nodes is to send \( p-1 \) messages (where \( p \) is the number of messages) with a cost of
\[
\sum_{i=1}^{p-1} (t_s + t_w \times m \times l_i)
\]

Can we do better?

This is wasteful, since intermediate nodes would have received the message. So, a better way is to keep a copy of the message and then send it on to the next node.

Consider a 2-D mesh and see how many messages must be sent to achieve the one-to-all broadcast.

First message from the source node to all other processors in the same row

Cost = \( t_s + t_w \times m \times (\sqrt{p}/2) \)

Note that the message is sent to the farthest node in a row at a distance of \( \sqrt{p}/2 \)

Each node in that row then sends a message to the farthest node in its column

So, what is the cost of broadcast?

Two messages, each with a cost of \( t_s + t_w \times m \times (\sqrt{p}/2) \)

So broadcast in 2D mesh costs

\[
2 \times (t_s + t_w \times m \times (\sqrt{p}/2))
\]

Interestingly this is \( O(p^{1/2}) \) same as sending one message

What about for a hypercube?

Source sends one message at a time to each of its \( d \)-neighbors
Each of these nodes sends messages to their neighbors
These messages can be sent in parallel
Total cost = (t_s + t_w * m) (log_2 p)

The reverse (or dual) of one-to-all broadcast is called Single-Node Accumulation or Gather operation.

How does this work? Consider reversing the order of sending messages. At each step, intermediates collates messages and forwards the collated message. We can also perform “sum”, “product” to achieve reduction.

Cost Equations

**one-to-one using**

- Ring: \( t_s + t_w * m * (p-1) \)
- 2-D Mesh: \( t_s + t_w * m * 2^{(\sqrt{p}/2)} \)
- Hypercube: \( t_s + t_w * m * \log_2(p) \)

**one-to-all broadcast**

- Ring: \( t_s + t_w * m * (p/2) \)
- 2-D Mesh: \( 2^n \left[t_s + t_w * m * (\sqrt{p}/2)\right] \)
- Hypercube: \( (t_s + t_w * m) (\log_2 p) \)
CSCE 5160 Parallel Processing

Consider *All-To-All* broadcast in a ring

We can overlap (or pipeline) broadcasting message from different source nodes.

In fact we need only \((p-1)\) steps to broadcast all messages from all to all.

\[
\text{cost} = (t_s + t_w + m)(p-1)
\]

Note the size of the message increases – so we need to modify the equation to include the size of the message for each message being sent

\[
\text{cost} = (p-1)t_s + t_w \sum_{i=1}^{p} m
\]

\[
= [t_s + t_w \frac{(p-1)(p-2)}{2}] \cdot m
\]

This can be approximated to \(O(p^2 \cdot m)\)

Likewise we should be able to find costs for other types of networks

We will use such costs to compare different types of algorithms

We need to include communication and computation costs

We will develop analyses and/or algorithms for other types of communications as needed
Measuring performance
How do you compute execution time of a program?

Execution Time of a sequential program (on one processor)
\[ \text{Execution Time} = (\#\text{instructions}) \times (\text{Avg. Cycles per Instruction}) \times (\text{clock-time}) \]

What if you have more than one processor?

Speed up = \( \frac{\text{Execution time on one processor}}{\text{Execution time on n processors}} \)

What is the maximum speedup you can get?

Maximum speedup = \# processors

Efficiency = \( \frac{(\text{Speed up})}{\text{\# processors}} \)

What is the maximum efficiency?

\[ \text{Efficiency} = \frac{(\text{Speed up})}{\text{\# processors}} = \left( \frac{T_1}{p \times T_2} \right) \]

Efficiency = \( \frac{(\text{Speed up})}{\text{\# processors}} \)

Ideal Efficiency = 100%

Super Linear Speed Up?
That is, Efficiency greater than 100%
or if we use 5 processors, we may get 6 times speed up?

Is it possible?

Note that when we are using \( p \) processors, the size of the memory available to the application is often increased by a factor of \( p \).

The effective cache size has also increased -- fewer cache misses and page faults
So, depending on the application, we may achieve super linear speed up

Note we are only talking about cases where we have \( p \) processors
not using concurrent programming on a single processor.
CSCE 5160 Parallel Processing

Consider the example on page 5.3 on page 200
If we use one processor with 64KB cache, the miss rate = 20%
clock cycle = 2ns and miss penalty is 100 ns

Effective access time 2ns*0.8 (on hit) + 0.2*100 (on miss) = 21.6ns

With two processors, each processor needs to deal with half as much data
But each has 64KB cache

The miss rate now is 10%
Access time = 2*0.9 + 0.1*100 = 11.8ns
Speed up (assuming everything stays the same) = (21.6*2)/11.8 = 3.66
(text book also adds other factors such as DRAM access)

Another Example of super linear speedup (problem 5.2 on page 228)

Super-linearity is also possible when we divide the search space among multiple nodes.
Consider the following DFS tree where the shaded node represents the solution. If we assume
that it takes t time units to visit each node of the tree (including traversing the link to reach the
next node and any computation to decide if further search is needed). What is the speed up if
we divide the search tree among to processors as shown?

Speed up = 12/5 = 2.4
CSCE 5160 Parallel Processing

If we have one processor it will take 12 time units (visiting 12 nodes)

If we have two processors, the second processor will find a solution in 5 time units)

Speed up = 12/5 = 2.4
Super-linear!

In most cases, we do not get the maximum speedup
One reason is given by the Amdahl’s law

**Amdahl’s law.**
Speed up is significantly affected by the serial fraction

\[ S = \frac{T_1}{T_n} \]

Let f be the serial fraction of the total program.
So, \( S = 1 \left( \frac{1}{(1-f)/n} + f \right) = \left( n/(1+f(n-1)) \right) \)

<table>
<thead>
<tr>
<th>f</th>
<th>speed up (n = 100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5%</td>
<td>16.8</td>
</tr>
<tr>
<td>10%</td>
<td>9.17</td>
</tr>
<tr>
<td>15%</td>
<td>6.1</td>
</tr>
</tbody>
</table>

Cost and cost optimal
Cost is normally related to the number of processors but may also include communication cost

The cost of a parallel algorithm can be viewed as \( p \times T_p \)

Efficiency = \( \frac{\text{Speedup}}{p} = \frac{T_s}{T_p \times p} \)

A parallel algorithm is cost optimal if its efficiency is 1

Consider an example of adding n numbers using p processors.

```c
for (i=0; i<n; i++) sum = sum + a[i];
Sequential time = O(n)
```

Each processor adds \( n/p \) elements and then we use a tree like accumulation.
For this example, we will assume that it takes 1 time unit to add two numbers and 1 unit to communicate over a link.

In the optimal algorithm, we view the accumulation of partial sums as a binary tree of adders
the total execution time using \( p \) processors = \( (n/p) + 2 \times \log(p) \)
The “2” comes from 1 for communication and 1 for computation.
CSCE 5160 Parallel Processing

Speed up = (sequential execution time)/(parallel execution time)

\[ \frac{n}{(n/p) + 2\log(p)} \]

= \[ \frac{n^p}{(n^p + 2p\log(p))} \]

Efficiency = speedup/p

= \[ \frac{n}{n + 2p\log(p)} \]

This is not cost optimal (efficiency is less than 1)

Consider plotting the efficiency as a function of p and n

<table>
<thead>
<tr>
<th>n</th>
<th>p=1</th>
<th>p=4</th>
<th>p=8</th>
<th>p=16</th>
<th>p=32</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>1</td>
<td>0.8</td>
<td>0.57</td>
<td>0.33</td>
<td>0.17</td>
</tr>
<tr>
<td>192</td>
<td>1</td>
<td>0.92</td>
<td>0.8</td>
<td>0.6</td>
<td>0.38</td>
</tr>
<tr>
<td>320</td>
<td>1</td>
<td>0.95</td>
<td>0.87</td>
<td>0.71</td>
<td>0.5</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>0.97</td>
<td>0.91</td>
<td>0.8</td>
<td>0.62</td>
</tr>
</tbody>
</table>

CSCE 5160 Parallel Processing

One observation: As we increase p, Efficiency is dropping for the same n

Second observation: What should we do to get the same efficiency as we increase p?

We need to increase the size of the problem (or n)

For example, in order to maintain a fixed efficiency off say 0.8,
if we have p= 4 processors, we need n = 64
if we have p= 8 processors, we need n = 192
if we use p =16 processors, we need n= 512
If we use p = 32 processors, we need n = 1280

Here our goal was to maintain a fixed efficiency -- we call this ISO-Efficiency.

As we add more processors, we need to solve larger problem sizes –
But does the size increase linearly with number of processors?

Scalability of the algorithm