COMP 322: Fundamentals of Parallel Programming

Lecture 30: Task Affinity with Places

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Worksheet #29: Characterizing Solutions to the Dining Philosophers Problem

For the five solutions studied in Lecture #29, indicate in the table below which of the following conditions are possible and why:

1. **Deadlock**: when all philosopher tasks are blocked
2. **Livelock**: when all philosopher tasks are executing (i.e., no philosopher is blocked) but ALL philosophers are starved (never get to eat)
3. **Starvation**: when one or more philosophers are starved (never get to eat)
4. **Non-Concurrency**: when more than one philosopher cannot eat at the same time, even when resources are available i.e., not being used

**NOTE**: Deadlock implies Starvation, and Livelock implies Starvation
<table>
<thead>
<tr>
<th></th>
<th>Deadlock</th>
<th>Livelock</th>
<th>Starvation</th>
<th>Non-concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution 1: synchronized</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Solution 2: tryLock/unLock</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Solution 3: isolated</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Solution 4: object-based isolation</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Solution 5: semaphores</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
An example Memory Hierarchy --- what is the cost of a Memory Access?

- **CPU registers**: hold words retrieved from L1 cache
- **L1 cache (Static RAM)**: holds cache lines retrieved from L2 cache
- **L2 cache (Static RAM)**: holds cache lines retrieved from main memory
- **Main memory (Dynamic RAM)**: holds disk blocks retrieved from local disks
- **Local secondary storage (local disks)**: holds files retrieved from disks on remote network servers
- **Remote secondary storage (tapes, distributed file systems, Web servers)**: slower, cheaper per byte

## Storage Trends

### SRAM

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$/MB</td>
<td>19,200</td>
<td>2,900</td>
<td>320</td>
<td>256</td>
<td>100</td>
<td>75</td>
<td>60</td>
<td>320</td>
</tr>
<tr>
<td>access (ns)</td>
<td>300</td>
<td>150</td>
<td>35</td>
<td>15</td>
<td>3</td>
<td>2</td>
<td>1.5</td>
<td>200</td>
</tr>
</tbody>
</table>

### DRAM

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$/MB</td>
<td>8,000</td>
<td>880</td>
<td>100</td>
<td>30</td>
<td>1</td>
<td>0.1</td>
<td>0.06</td>
<td>130,000</td>
</tr>
<tr>
<td>access (ms)</td>
<td>375</td>
<td>200</td>
<td>100</td>
<td>70</td>
<td>60</td>
<td>50</td>
<td>40</td>
<td>9</td>
</tr>
<tr>
<td>typical size (MB)</td>
<td>0.064</td>
<td>0.256</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>2,000</td>
<td>8,000</td>
<td>125,000</td>
</tr>
</tbody>
</table>

### Disk

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$/MB</td>
<td>500</td>
<td>100</td>
<td>8</td>
<td>0.30</td>
<td>0.01</td>
<td>0.005</td>
<td>0.0003</td>
<td>1,600,000</td>
</tr>
<tr>
<td>access (ms)</td>
<td>87</td>
<td>75</td>
<td>28</td>
<td>10</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>29</td>
</tr>
<tr>
<td>typical size (MB)</td>
<td>1</td>
<td>10</td>
<td>160</td>
<td>1,000</td>
<td>20,000</td>
<td>160,000</td>
<td>1,500,000</td>
<td>1,500,000</td>
</tr>
</tbody>
</table>

Cache Memories

- **Cache memories** are small, fast SRAM-based memories managed automatically in hardware.
  —Hold frequently accessed blocks of main memory
- CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory.
- Typical system structure:

Source: http://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/lectures/09-memory-hierarchy.pptx
Examples of Caching in the Hierarchy

Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available. ... We are ... forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

A. W. Burks, H. H. Goldstine, and J. von Neumann

Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (1946)

Ultimate goal: create a large pool of storage with average cost per byte that approaches that of the cheap storage near the bottom of the hierarchy, and average latency that approaches that of fast storage near the top of the hierarchy.

Source: http://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/lectures/09-memory-hierarchy.pptx
Locality

• Principle of Locality:
  — Empirical observation: Programs tend to use data and instructions with addresses near or equal to those they have used recently

• Temporal locality:
  — Recently referenced items are likely to be referenced again in the near future

• Spatial locality:
  — Items with nearby addresses tend to be referenced close together in time
  — A Java programmer can only influence spatial locality at the intra-object level
  — The garbage collector and memory management system determines inter-object placement

Source: http://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/lectures/09-memory-hierarchy.pptx
Locality Example

- Data references
  - Reference array elements in succession (stride-1 reference pattern).
  - Reference variable sum each iteration.

- Instruction references
  - Reference instructions in sequence.
  - Cycle through loop repeatedly.

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

Memory hierarchy for a single Intel Xeon Quad-core E5440 HarperTown processor chip

—A SUG@R node contains TWO such chips, for a total of 8 cores
Programmer Control of Task Assignment to Processors

- The parallel programming constructs that we’ve studied thus far result in tasks that are assigned to processors *dynamically* by the HJ runtime system
  — Programmer does not worry about task assignment details

- Sometimes, programmer control of task assignment can lead to significant performance advantages due to improved locality

- Motivation for HJ “places”
  — Provide the programmer a mechanism to map each task to a set of processors when the task is created
Places in HJ

HJ programmer defines mapping from HJ tasks to set of places

HJ runtime defines mapping from places to one or more worker Java threads per place

The option “-places p:w” when executing an HJ program can be used to specify $p$, the number of places $w$, the number of worker threads per place
Example of –places 4:2 option on an 8-core node (4 places w/ 2 workers per place)
Places in HJ

**here** = place at which current task is executing

**place.MAX_PLACES** = total number of places (runtime constant)
   Specified by value of **p** in runtime option, **-places p:w**

**place.factory.place(i)** = place corresponding to index **i**

**<place-expr>.toString()** returns a string of the form “place(id=0)”

**<place-expr>.id** returns the id of the place as an int

**async at(P) S**
   - Creates new task to execute statement **S** at place **P**
   - **async S** is equivalent to **async at(here) S**
   - **Main program task starts at place.factory.place(0)**

Note that **here** in a child task refers to the place **P** at which the child task is executing, not the place where the parent task is executing.
Example of –places 4:2 option on an 8-core node (4 places w/ 2 workers per place)

// Main program starts at place 0
async at(place.factory.place(0)) S1;
async at(place.factory.place(0)) S2;

async at(place.factory.place(1)) S3;
async at(place.factory.place(1)) S4;
async at(place.factory.place(1)) S5;

async at(place.factory.place(2)) S6;
async at(place.factory.place(2)) S7;
async at(place.factory.place(2)) S8;

async at(place.factory.place(3)) S9;
async at(place.factory.place(3)) S10;
Example of –places 1:8 option (1 place w/ 8 workers per place)

All async’s run at place 0 when there’s only one place!
Example HJ program with places

```java
class T1 {
    final place affinity;
    ...
    // T1's constructor sets affinity to place where instance was created
    T1() { affinity = here; ... }
    ...
}

finish { // Inter-place parallelism
    System.out.println("Parent\n    place = ", here); // Parent task's place
    for (T1 a = . . .) {
        async at (a.affinity) { // Execute async at place with affinity to a
            a.foo();
            System.out.println("Child\n            place = ", here); // Child task's place
        } // async
    } // for
} // finish
...
Distributions --- hj.lang.dist

- A distribution maps points in a rectangular index space (region) to places e.g.,
  - \( i \mapsto \text{place.factory.place}(i \% \text{place.MAX_PLACES}) \)

- Programmers are free to create any data structure they choose to store and compute these mappings

- For convenience, the HJ language provides a predefined type, \texttt{hj.lang.dist}, to simplify working with distributions

- Some public members available in an instance \( d \) of \texttt{hj.lang.dist} are:
  - \( d.\text{rank} = \) number of dimensions in the input region for distribution \( d \)
  - \( d.\text{get}(p) = \) place for point \( p \) mapped by distribution \( d \). It is an error to call \( d.\text{get}(p) \) if \( p.\text{rank} \neq d.\text{rank} \).
  - \( d.\text{places}() = \) set of places in the range of distribution \( d \)
  - \( d.\text{restrictToRegion}(pl) = \) region of points mapped to place \( pl \) by distribution \( d \)
Block Distribution

- `dist.factory.block([lo:hi])` creates a block distribution over the one-dimensional region, `lo:hi`.
- A block distribution splits the region into contiguous subregions, one per place, while trying to keep the subregions as close to equal in size as possible.
- Block distributions can improve the performance of parallel loops that exhibit spatial locality across contiguous iterations.
- Example in Table 1: `dist.factory.block([0:15])` for 4 places

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place id</td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Block Distribution (contd)

- If the input region is multidimensional, then a block distribution is computed over the linearized one-dimensional version of the multidimensional region

- Example in Table 2: dist.factory.block([0:7,0:1]) for 4 places

<table>
<thead>
<tr>
<th>Index</th>
<th>0,0</th>
<th>0,1</th>
<th>1,0</th>
<th>1,1</th>
<th>2,0</th>
<th>2,1</th>
<th>3,0</th>
<th>3,1</th>
<th>4,0</th>
<th>4,1</th>
<th>5,0</th>
<th>5,1</th>
<th>6,0</th>
<th>6,1</th>
<th>7,0</th>
<th>7,1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place id</td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Distributed Parallel Loops

- Listing 2 shows the typical pattern used to iterate over an input region r, while creating one async task for each iteration p at the place dictated by distribution d i.e., at place d.get(p).

- This pattern works correctly regardless of the rank and contents of input region r and input distribution d i.e., it is not constrained to block distributions

```java
finish {
    region r = ... ; // e.g., [0:15] or [0:7,0:1]
    dist d = dist.factory.block(r);
    for (point p : r)
        async at(d.get(p)) {
            // Execute iteration p at place specified by distribution d

        }
    } // finish
... 
```
Cyclic Distribution

- `dist.factory.cyclic([lo:hi])` creates a cyclic distribution over the one-dimensional region, lo:hi.
- A cyclic distribution “cycles” through places 0 … place.MAX PLACES – 1 when spanning the input region.
- Cyclic distributions can improve the performance of parallel loops that exhibit load imbalance.
- Example in Table 3: `dist.factory.cyclic([0:15])` for 4 places.

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place id</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>[0,0]</th>
<th>[0,1]</th>
<th>[1,0]</th>
<th>[1,1]</th>
<th>[2,0]</th>
<th>[2,1]</th>
<th>[3,0]</th>
<th>[3,1]</th>
<th>[4,0]</th>
<th>[4,1]</th>
<th>[5,0]</th>
<th>[5,1]</th>
<th>[6,0]</th>
<th>[6,1]</th>
<th>[7,0]</th>
<th>[7,1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place id</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Chunked Fork-Join Iterative Averaging Example with Places

1. public void runDistChunkedForkJoin(int iterations,
   int numChunks, dist d) {
2.     for (int iter = 0; iter < iterations; iter++) {
3.         finish for (point [jj] : [0:numChunks-1]) {
4.             async at(d.get(jj)) {
5.                 for (point [j] : getChunk([1:n],numChunks,jj))
6.                     myNew[j] = (myVal[j-1] + myVal[j+1]) / 2.0;
7.             } // finish-for-async
8.         } // for iter
9.     double[] temp = myNew; myNew = myVal; myVal = temp;
10. } // runDistChunkedForkJoin

• Chunk jj is always executed in the same place for each iter
• Method runDistChunkedForkJoin can be called with different values of distribution parameter d

Let's try another example of a distributed parallel loop in Worksheet 30!
Worksheet #30: impact of distribution on parallel completion time

1. public void sampleKernel(int iterations,
2. int numChunks, dist d) {
3.     for (int iter = 0; iter < iterations; iter++) {
4.         finish for (point [jj] : [0:numChunks-1])
5.             async at(d.get(jj)) {
6.                 perf.doWork(jj);
7.                 // Assume that time to process chunk jj = jj units
8.             } // finish-for-async
9.         double[] temp = myNew; myNew = myVal; myVal = temp;
10.     } // for iter
11. } // sample kernel

• Assume an execution with n places using the option, -places n:1
• Will a block or cyclic distribution for d have a smaller abstract completion time, assuming that all tasks on the same place are serialized?