COMP 322: Fundamentals of Parallel Programming

Lecture 31: Task Affinity with Places

(Start of Module 3 on Distribution & Locality)

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Worksheet #30: 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

NOTES:
• Deadlock implies Starvation and Non-Concurrency
• Livelock implies Starvation and Non-Concurrency
<table>
<thead>
<tr>
<th></th>
<th>Deadlock</th>
<th>Livelock</th>
<th>Starvation</th>
<th>Non-concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solution 1:</strong></td>
<td>synchronized</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Solution 2:</strong></td>
<td>tryLock/unLock</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Solution 3:</strong></td>
<td>isolated</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Solution 4:</strong></td>
<td>object-based isolation</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Solution 5:</strong></td>
<td>semaphores</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Organization of a Distributed-Memory Multiprocessor

Figure (a)

- Host node (Pc) connected to a cluster of processor nodes (P_0 \ldots P_m)
- Processors P_0 \ldots P_m communicate via an interconnection network which could be standard TCP/IP (e.g., for Map-Reduce) or specialized for high performance communication (e.g., for scientific computing)

Figure (b)

- Each processor node consists of a processor, memory, and a Network Interface Card (NIC) connected to a router node (R) in the interconnect

Processors communicate by sending messages via an interconnect
Cores communicate by reading and writing data in a “shared memory”

- Memory hierarchy for a single Intel Xeon (Nehalem) Quad-core processor chip
  - A STIC node contains TWO such chips, for a total of 8 cores
What is the cost of a Memory Access?
An example Memory Hierarchy

- **L0**: Registers
- **L1**: L1 cache (Static RAM)
- **L2**: L2 cache (Static RAM)
- **L3**: Main memory (Dynamic RAM)
- **L4**: Local secondary storage (local disks)
- **L5**: Remote secondary storage (tapes, distributed file systems, Web servers)

**CPU registers** hold words retrieved from **L1 cache**

**L1 cache** holds cache lines retrieved from **L2 cache**

**L2 cache** holds cache lines retrieved from **main memory**

**Main memory** holds disk blocks retrieved from local disks

**Local disks** hold files retrieved from disks on remote network servers

### 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:

![Diagram of CPU chip with cache memories, register file, ALU, bus interface, system bus, I/O bridge, memory bus, and main memory.](http://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/lectures/09-memory-hierarchy.pptx)
Examples of Caching in the Hierarchy

<table>
<thead>
<tr>
<th>Hierarchy Level</th>
<th>Memory Access Time</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Fast</td>
<td>Small</td>
</tr>
<tr>
<td>TLB</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>L1 cache</td>
<td>Medium</td>
<td>Large</td>
</tr>
<tr>
<td>L2 cache</td>
<td>Slow</td>
<td>Very Large</td>
</tr>
<tr>
<td>Virtual Buffer</td>
<td>Slow</td>
<td>Large</td>
</tr>
<tr>
<td>Disk cache</td>
<td>Slow</td>
<td>Very Large</td>
</tr>
<tr>
<td>Web cache</td>
<td>Slow</td>
<td>Large</td>
</tr>
</tbody>
</table>

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.

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

Locality Example

- Data references
  - Reference array elements in succession (stride-1 reference pattern).
  - Reference variable $\text{sum}$ 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 in a Multicore Processor

- Memory hierarchy for a single Intel Xeon (Nehalem) Quad-core processor chip
  —A STIC 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 restrict task execution to a subset of processors for improved locality
  - Current HJlib implementation supports one level of locality via places, but future HJlib versions will support hierarchical places
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 API calls

\[
\text{HjSystemProperty.\texttt{numPlaces}.set}(p); \quad \text{HjSystemProperty.\texttt{numWorkers}.set}(w);
\]

when executing an HJ program can be used to specify

- \(p\), the number of places
- \(w\), the number of worker threads per place

we will abbreviate this as \(p:w\)
Example of 4:2 option on an 8-core node (4 places w/ 2 workers per place)

Core A
- L1
- L1
- L2 unified cache

Core B
- L1
- Place 0

Core C
- L1
- Place 1

Core D
- L1
- L2 unified cache

Core E
- L1
- Place 2

Core F
- L1
- L2 unified cache

Core G
- L1
- Place 3

Core H
- L1
Places in HJlib

\( \text{here}() = \text{place at which current task is executing} \)

\( \text{numPlaces}() = \text{total number of places (runtime constant)} \)

Specified by value of \( p \) in runtime option:

\[ \text{HjSystemProperty.numPlaces.set}(p); \]

\( \text{place}(i) = \text{place corresponding to index } i \)

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

\( \text{<place-expr>.id()} \) returns the id of the place as an int

\( \text{asyncAt}(P, () \rightarrow S) \)

- Creates new task to execute statement \( S \) at place \( P \)
- \( \text{async}(() \rightarrow S) \) is equivalent to \( \text{asyncAt}(\text{here}(), () \rightarrow S) \)
- \( \text{Main program task starts at place}(0) \)

Note that \( \text{here}() \) 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 4:2 option on an 8-core node (4 places w/ 2 workers per place)

// Main program starts at place 0
asyncAt(place(0), () -> S1);
asyncAt(place(0), () -> S2);

asyncAt(place(1), () -> S3);
asyncAt(place(1), () -> S4);
asyncAt(place(1), () -> S5);

asyncAt(place(2), () -> S6);
asyncAt(place(2), () -> S7);
asyncAt(place(2), () -> S8);

asyncAt(place(3), () -> S9);
asyncAt(place(3), () -> S10);
Example of 1:8 option (1 place w/ 8 workers per place)

All async's run at place 0 when there's only one place!
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\_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\_place\_\_\_\_", here); // Child task's place
        } // async
    } // for
} // finish

...
Chunked Fork-Join Iterative Averaging Example with Places

1. public void runDistChunkedForkJoin(
   2.     int iterations, int numChunks, Dist dist) {
   3.     // dist is a user-defined map from int to HjPlace
   4.     for (int iter = 0; iter < iterations; iter++) {
   5.         finish(() -> {
   6.             forseq (0, numChunks - 1, (jj) -> {
   7.                 asyncAt(dist.get(jj), () -> {
   8.                     forseq (getChunk(1, n, numChunks, jj), (j) -> {
   9.                         myNew[j] = (myVal[j-1] + myVal[j+1]) / 2.0;
   10.                     });
   11.                 });
   12.             });
   13.         });
   14.         double[] temp = myNew; myNew = myVal; myVal = temp;
   15.     } // for iter
   16. }

• Chunk jj is always executed in the same place for each iter
• Method runDistChunkedForkJoin can be called with different values of distribution parameter d
Analyzing Locality of Fork-Join Iterative Averaging Example with Places

Locality benefits will be realized if all instances of chunk 0 execute on the same core and reuse data from the same cache.
Block Distribution

- A block distribution splits the index 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: `dist.get(index)` for a block distribution on 4 places, when index is in the range, 0…15

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Distributed Parallel Loops

- The pseudocode below 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.

```python
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

- 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: dist.get(index) for a cyclic distribution on 4 places, when index is in the range, 0…15

<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>