An example Memory Hierarchy --- what is the cost of a Memory Access?

- **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)

**Smaller, faster, costlier per byte**

**Larger, slower, cheaper per byte**

- 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 (ns)</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></td>
<td>4</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:**

  ![Cache Memories Diagram](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>What is cached?</th>
<th>Where is it cached?</th>
<th>Latency (cycles)</th>
<th>Managed by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>4-32 bytes (words)</td>
<td>CPU core</td>
<td>0</td>
<td>Compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>Address translations</td>
<td>On-chip TLB</td>
<td>0</td>
<td>Hardware</td>
</tr>
<tr>
<td>L1 cache</td>
<td>64-bytes block</td>
<td>On-Chip L1</td>
<td>$O(10^3)$</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>64-bytes block</td>
<td>On/Off-Chip L2</td>
<td>$O(10^4)$</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>4 KB page</td>
<td>Main memory</td>
<td>$O(10^6)$</td>
<td>Hardware &amp; OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>Parts of files</td>
<td>Main memory</td>
<td>$O(10^6)$</td>
<td>OS</td>
</tr>
<tr>
<td>Disk cache</td>
<td>Disk sectors</td>
<td>Disk controller</td>
<td>$O(10^5)$</td>
<td>Disk firmware</td>
</tr>
<tr>
<td>Network buffer cache</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>$O(10^5)$</td>
<td>AFS/NFS client</td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>$O(10^5)$</td>
<td>Web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>Web pages</td>
<td>Remote server disks</td>
<td>$O(10^5)$</td>
<td>Web proxy server</td>
</tr>
</tbody>
</table>

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

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

- **Data references**
  - Reference array elements in succession (stride-1 reference pattern).  
    Spatial locality
  - Reference variable `sum` each iteration.  
    Temporal locality
- **Instruction references**
  - Reference instructions in sequence.  
    Spatial locality
  - Cycle through loop repeatedly.  
    Temporal locality

Memory Hierarchy in a Multicore Processor

- Memory hierarchy for a single Intel Xeon Quad-core E5440 HarperTown processor chip
  - A SUGAR 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 a SUG@R 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 a SUG@R 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\_\_place=-", here); // Parent task's place
    for (T1 a = . . .) {
        async at (a.affinity) { // Execute async at place with affinity to a
            System.out.println("Child\_\_place=-", here); // Child task's place
            } // async
        } // for
    } // finish
    . . .
```

Chunked Fork-Join Iterative Averaging Example with Places

```java
for (point [iter] : [0:iterations-1]) {
    finish for (point [i] : [0:tasks-1]) {
        async at (place.factory.place(i % place.MAX_PLACES)) {
            int start = i * batchSize + 1;
            for (point [j] : [start:Math.min(start+batchSize-1,n)]) {
                myNew[j] = (myVal[j-1] + myVal[j+1]) / 2.0;
            }
        } // async
    } // finish for
    double[] temp = myNew; myNew = myVal; myVal = temp;
}
```

* Assume a -places 4:4 configuration with 4 places and 4 workers per places for execution on a 16-core machine
  * Set tasks = 16 so as to create one async per worker
  * Use i % place.MAX_PLACES to compute destination place for each async
  * Each subarray is processed at same place for successive iterations of for-iter loop

<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>
Distributions

• A distribution maps points in a rectangular index space (region) to places e.g.,
  — \( i \rightarrow \text{place.factory.place}(i \% \text{place.MAX PLACES}-1) \)

• 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, \( \text{hj.lang.dist} \), to simplify working with distributions

• Some public members available in an instance \( d \) of \( \text{hj.lang.dist} \) are as follows
  — \( d.\text{rank} \) = 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

- \texttt{dist.factory.block([lo:hi])} creates a block distribution over the one-dimensional region, \texttt{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.

<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></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>

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: \texttt{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>1</td>
<td>2</td>
<td>3</td>
<td></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

- 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 \( d.\text{get}(p) \).

- 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.

```c
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 \( 0 \) … \( \text{place.MAX PLACES} - 1 \) 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>

- Example in Table 4: `dist.factory.cyclic([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>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>
Announcements (REMINDER)

• Homework 3 due on Wednesday, Feb 22nd
  —Performance results for parts 2 and 3 of assignment must be obtained on Sugar (see Section 4)

• No lab next week
  —Use the time for HW3 and to prepare for Exam 1

• Exam 1 will be held in the lecture on Friday, Feb 24th
  —Closed book 50-minute exam
  —Scope of exam includes lectures up to Monday, Feb 20th
  —Feb 22nd lecture will be a midterm review before exam
  —Contact me ASAP if you have an extenuating circumstance and need to take the midterm at an alternate time