The Memory Hierarchy Exploits Locality of Reference

- Idea: small amount of fast memory
- Keep frequently used data in the fast memory
- LRU replacement policy
  - Keep recently used data in cache
  - To free space, remove Least Recently Used data

Traversing an Array

- One miss for every 4 accesses in a traversal

Example Memory Hierarchy Statistics

<table>
<thead>
<tr>
<th>Name</th>
<th>Extra CPU cycles used to access</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 (on chip) cache</td>
<td>0</td>
<td>32 KB</td>
</tr>
<tr>
<td>L2 cache</td>
<td>8</td>
<td>512 KB</td>
</tr>
<tr>
<td>RAM</td>
<td>35</td>
<td>256 MB</td>
</tr>
<tr>
<td>Hard Drive</td>
<td>500,000</td>
<td>8 GB</td>
</tr>
</tbody>
</table>

Cache Details (simplified)

Iterative MergeSort

- Cache Size
- cache misses
- cache hits
Iterative MergeSort – cont’d

Cache Size
no temporal locality!

“Tiled” MergeSort – better

Cache Size

“Tiled” MergeSort – cont’d

Cache Size

QuickSort

- Initial partition causes a lot of cache misses
- As subproblems become smaller, they fit into cache
- Good cache performance

Radix Sort – Very Naughty

- On each BucketSort
  - Sweep through input list – cache misses along the way (bad!)
  - Append to output list – indexed by pseudo-random digit (ouch!)

Cache Misses

<table>
<thead>
<tr>
<th>set size in keys</th>
<th>cache misses per key</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0</td>
</tr>
<tr>
<td>100000</td>
<td>0</td>
</tr>
<tr>
<td>1e+06</td>
<td>0</td>
</tr>
</tbody>
</table>

memory-tuned heapsort
memory-tuned quicksort
memory-tuned mergesort
radix sort
Conclusions

• Speed of cache, RAM, and external memory has a huge impact on sorting (and other algorithms as well)
• Algorithms with same asymptotic complexity may be best for different kinds of memory
• Tuning algorithm to improve cache performance can offer large improvements (iterative vs. tiled mergesort)