6.895 Theory of Parallel Systems Lecture 3

High Performance Serial Programming and Caching

Lecturer: Bradley Kuszmaul and Michael Bender Scribe: Kenneth Barr and Zardosht Kasheff

Lecture Summary

1. *High Performance Serial Programming*
   Before writing fast parallel code, you must understand the hardware on which your program will run and how to write fast *serial* code on this hardware.

2. *A Theoretic Machine Model*
   In this section, we formalize a cache-dependent model for analyzing code on a particular computer system.

3. *Analysis of Several Algorithms*
   In this section, we analyze several algorithms using the model defined in the previous section

1 High Performance Serial Programming

1.1 Computer Hardware

High performance programming is impossible without knowledge of the hardware in the host computer. This section discusses one of the most important elements of the hardware: the memory hierarchy. Figure 1 sketches the memory structures in computer. Each level of the memory hierarchy is larger and slower than the one that preceded it. Closest to the processor is a small, fast register file. Data words (e.g., 64 bits) are brought to the register file from the cache. A cache contains a subset of the words in main memory and deals with ***lines*** (e.g., 128 bytes) rather than individual words. Table 1 shows the sizes and speeds of memory hierarchies for two recent computer systems.

	ygg: Origin 2000 w/R10000	Pentium 4
Processor speed	195 MHz	2500 MHz
Registers	64	8
L1 Cache size	32KB	8KB
L1 Cache access time (ns)	10	1
L1 Cache line size	128B	64B
L1 Associativity	2-way	4-way
L1 Cache lines	256	64
L2 Cache size	4MB	512KB
L2 Cache access time (ns) ns	62	8
L2 Cache line size	128B	128B
L2 Associativity	2	8-way
L2 Cache lines	32K	4K
Memory access time (ns)	484	150

Table 1: System parameters

Caches are small, fast subsets of main memory. Since they are smaller than main memory, words necessarily map to more than one location in the cache. Therefore, the address of the data (the tag) is part of the information that is stored alongside the data itself. To reduce miss-rates, associative caches, in which

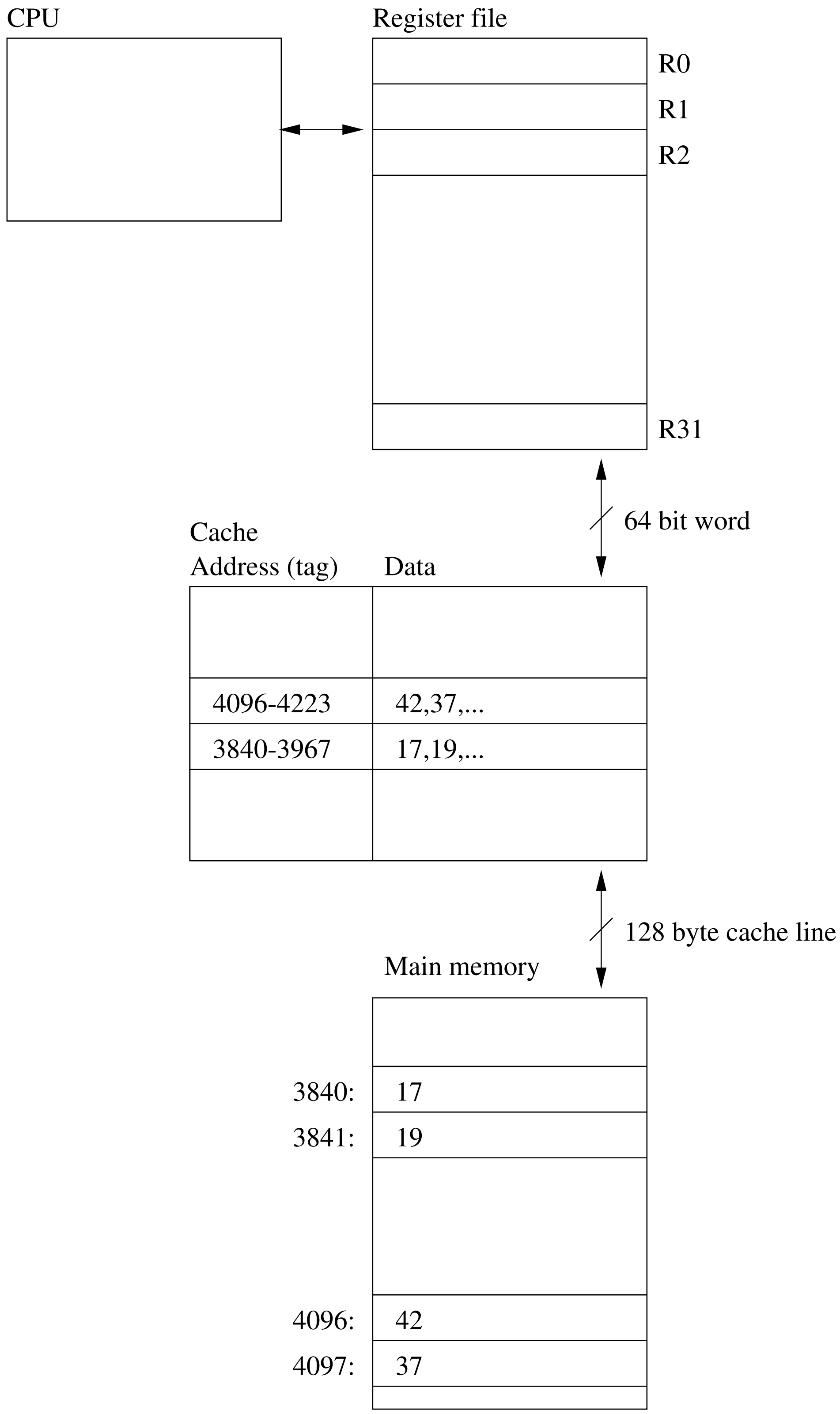

Figure 1: Memory hierarchy.

a word may be found in several locations, are more popular than direct-mapped caches (which we did not discuss). There are two types of associativity:

Fully associative Any word of memory can be stored in any cache line.

Set associative Words of memory can be placed in restricted set of places in the cache. A word is mapped onto a set and can be placed anywhere in that set. If there are n words in a set, the cache is n-way set associative.

For example, a 512 Kilobyte (KB), 4-way set associative cache with 128 byte lines would contain 1024 sets. Memory locations that are 128KB apart in memory are in the same set and compete for space in that set. If the replacement policy and data access pattern are compatible, the four ways permit four such addresses to coexist in the cache without evicting each other.

1.2 Compilers

Compilers translate high-level language to assembly language.[1] The handout [MIT03] shows what this assembly language looks like. Note that instructions are simple (usually one or two operands, one result). To produce the best set of simple assembly instructions, you must use a "good compiler." Though SGI `cc` has a good reputation, `gcc` can outperform it occasionally. Here are some more explicit tricks and tools.

- Profile your code so that you know where to focus your optimization effort. Use the `-pg` option to gcc or the `-p` to `cc`. Then run `gprof` or `prof`, respectively. Profiling uses statistical sampling (e.g., every 1ms it asks: what function is currently executing?). This is the "real-life" way to do what we did in last lecture with theory (that is, attack the $O(n)$ merge).
- Debug your code. `-g` inserts debugging symbols for gdb. Caveat: we don't appear to have a working debugger on `ygg`.
- Enable compiler optimization with `-O2`; usually gets a dramatic speedup.
- `-S` produces assembly listings so you can examine the results of compilation textually.
- Become familiar with all of the other switches available to you by reading the manual:

```
ygg\% man cc
ygg\% info gcc
```

1.3 High-level optimizations

Ideally the compiler would perform all high-level optimizations for you, but in practice reorganizing/optimizing your code manually is necessary to help produce a fast implementation.

[1] Usually, the compiler will invoke an assembler to translate the assembly language to machine code

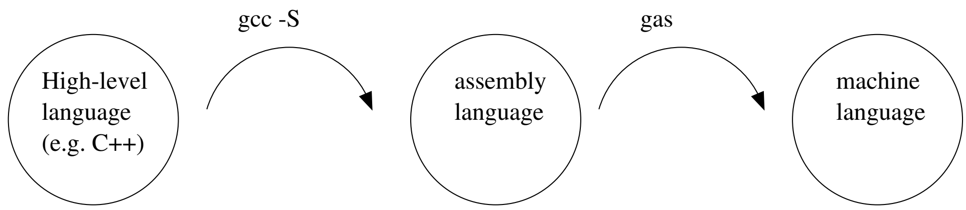

Figure 2: Compilation process

A good reference is AMD's Software Optimization Guide [AMD03]. Though written with the AMD64 instruction set and microarchitecture in mind, many of the techniques can be applied broadly. Other vendors have similar guides which may have useful machine-specific tricks. The remainder of this section contains the high-level optimizations discussed in class.

1.3.1 Exploit registers

Registers are the fastest storage elements at our disposal, so structure your code to do as much work as possible with data in registers before that data is pushed further down the memory hierarchy. For example, a 2x2 matrix is small enough to allow all operands and results to fit in registers. Since each operand is used several times, it is best to leave them in registers for the duration of the computation.

1.3.2 Exploit fast memory

Carrying the above idea further, we would prefer to work on data in L1 cache to L2 cache, and we'd prefer L2 to main memory. Size your problems appropriately; use cache-conscious data layout; or use optimal, cache-oblivous algorithms (see Section 2).

1.3.3 Move computations out of loops

This loop initializes the array with $\sin x$ which is effectively a constant, but the compiler does not know this and will make 1000 calls to the sin library function. If you were doing `a[i]=x+y`, the compiler might be able to tell and remove the add from the loop, but it doesn't know that sin() is different (here) than an arbitrary function call.

```
for(i=0; i<1000; i++){
  a[i]=sin(x);
}
```

1.3.4 Strength reduction

Reduce complex operations to simpler forms (e.g., exponentiation into a multiply or multiply into an add). The examples from lecture:

```
for(i=0; i<1000; i++){
  a[i]=pow(S,i);
}
```

becomes

```
tmp=1;
for(i=0; i<1000; i++){
  a[i]=tmp;
  tmp*=S;
}
```

1.3.5 Unroll loops

To eliminate the overhead associated with looping, try unrolling your loops to some degree.

```
for(i=0; i<N; i++)
  f(i);
```

becomes

```
for(i=0; i+1<N; i+=2){
  f(i);
  f(i+1);
}
for(; i<N; i++)         //clean up last case
  f(i);
```

This effectively changes $r+ = a(i) * b(i)$ to $r+ = a(i) * b(i) + a(i+1) * b(i+1)$ Sometimes the compiler can do this for you, sometimes it cannot.

1.3.6 Remove control dependencies

In deeply pipelined machines, a mispredicted branch can require tens of cycles to repair, squashing hundreds of instructions. Sometimes you can replace a hard-to-predict branch with a predicated instruction such as MIPS's conditional move:

```
movz Ri, Rj, Rn       #if(Rn==0) Ri=Rj
```

Here a control dependence has become a data dependence....

1.4 Example: isort vs. sort5 [MIT03]

Recall that despite its $O(n^3)$ running time, insertion sort needs few steps to complete for for small values of n. Despite a larger number of operations, the `sort5` code is measureably faster when sorting five elements. Why? The `isort` code is slowed down by unpredictable branches and more memory references. To demonstrate, Table 2 shows the time taken by each routine with varying compiler and machine combinations.

	gcc	cc	gcc (Pentium)
isort	599ns	580ns	88ns
sort5	275ns	112ns	34ns

Table 2: Sort timing

2 A Theoretic Machine Model

2.1 Motivation of Model

A model that takes caching into account is needed because data locality has a significant effect on the practical runtime of algorithms.

Poor Matrix Multiplication Algorithm. Below is the trivial algorithm for Matrix Multiplication for the n by n matrices $C = AB$. This algorithm works well on the RAM model, where all instructions complete in one timestep, but works poorly in practice.

Mult(C,A,B)
  for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
      $c_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$.

This algorithm performs poorly because it lacks data locality hence causing bad cache performance.

2.2 Memory Heirarchies

2.2.1 Multilevel Memory Heirarchy

A multilevel memory heirarchy has many parameters, making it complicated and difficult to analyze; see Figure 3.

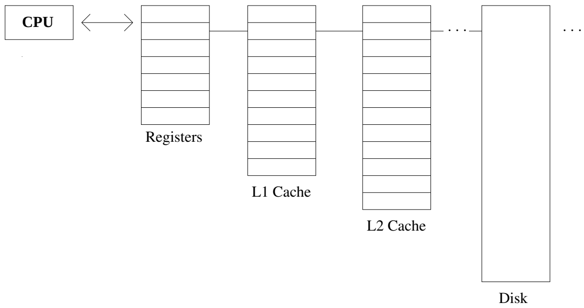

Figure 3: Multievel Memory Hierarchy. Each level has a different size, block size, and number of cache lines. The time for a memory transfer between levels differs.

2.2.2 Two-Level Memory Heirarchy

The following two-level memory model is a simplification that is easier to analyze; see Figure 4.

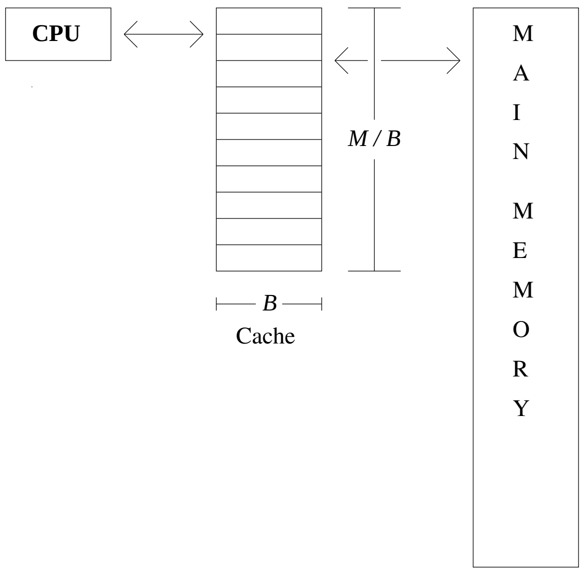

Figure 4: Two-Level Memory Hierarchy with block size B, cache size M, and M/B lines of cache.

Description. This two-level memory hierachy consists of a CPU, cache, and main memory. The cache is divided into cache lines, or ***blocks*** of size B. The cache has size M and is composed of M/B blocks. The act of moving a block between cache and main memory is called a ***memory transfer***. When a memory transfer brings a block from main memory into a "slot" in cache, the memory block that previously occupied that slot is ***evicted***.

Objective. We want to minimize the number of memory transfers in an algorithm, because they are the most expensive operations.

Although referred to as "cache" and "main memory", these two levels can represent any two contiguous levels in the memory heirarchy. For the remainder of these notes, we use the two-level memory model and not the multilevel memory model.

2.3 Block Replacement Strategies

One issue is how to select which block to evict during a memory transfer. Below we compare ideal solutions to realistic ones.

2.3.1 Ideal Cache Model vs. Other Replacement Strategies

Description. According to the ***ideal cache model*** [FLPR99], the best block to replace is the block that is accessed farthest in the future [Belady]. We call this model ***FIF***, which stands for "farthest in future".

Difficulty: Cannot Look into future. Although the optimal strategy, FIF is not implementable. One cannot look into the future to select which block to evict. However, there exists a simulation that avoids this difficulty.

Theorem 1 *LRU/FIFO with cache of size $M \leq$ 2OPT with cache of size $2M$.*

For further details see [FLPR99].

Difficulty: Limited Assiciativity. The ideal cache model assumes full associativity while actual hardware has limited associativity. In our ideal cache model, we assume any block of memory can be stored anywhere in cache (***full associativity***). Realistically, caches do not have this property. Caches can be ***direct-mapped***, meaning block i can *only* be cached in slot $i \mod (M/B)$. Caches can also be ***2-way set associative***, meaning only two blocks can be stored in each slot ($M/2B$ slots total).

As described in Section 1, actual hardware has limited associativity and also has a simpler replacement policy.

Theorem 2 *[FLPR99] Auto-replacement, fully associative LRU/FIFO cache can be simulated using a manual replacement, direct mapped cache in $O(1)$ expected time*

For further details see [FLPR99].

2.4 Cache-Aware v. Cache-Oblivious

Sections 2.1 - 2.3 give a well defined computational model involving caching. We define two different types of algorithms:

Cache-Oblivious Algorithms. A cache-oblivious algorithm does not use the variables B and M, but its performance still depends on B and M. The cache-oblivious model enables us to reason about a simple two-level memory model but prove results about an unknown multilevel memory model. Cache-oblivious algorithms perform an *asymptotically optimal* number of memory transfers for *any* memory heirarchy at *all levels* of the hierarchy. The number of memory transfers between any two levels is within a constant factor of optimal. Any linear combination of the transfer counts is optimized.

Cache-Aware Algorithms. A cache-aware algorithm is an optimal algorithm that is aware of and uses the variables M and B.

3 Analysis of Several Algorithms

3.1 Method of Asymptotic Analysis

The goal is to use algorithms with a solid foundation to predict performance. Thus, we present in Section 2 a simple algorithmic model that gives more accurate running time than the standard RAM model.

Defining Cost. We define the ***cost*** of an algorithm as the number of memory transfers. Note we do not take into account the time to transfer memory from the cache to the CPU or operations on data in cache, because the time needed to perform these operations is much less than the time needed to perform a memory transfer between cache and main memory.

3.2 Example 1: Array-scanning

Consider array-scanning. The goal is to read in order the elements in an array $A[1...n]$.

```
for i = 1 to n
  sum += A[i].
```

Cost. We see:

$$\text{Cost} \leq \left\lceil \frac{N}{B} \right\rceil + 1.$$

An example is in Figure 5.

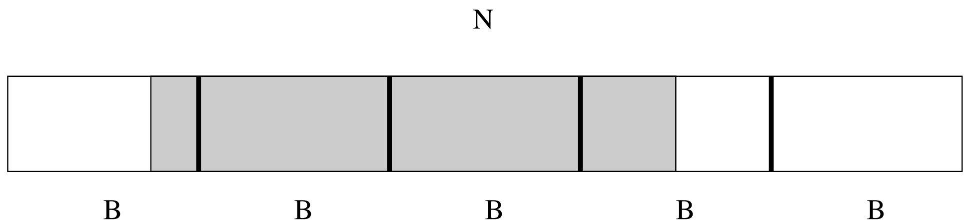

Figure 5: Array spread across blocks. The clear rectangles are blocks in cache. The shaded region is the location of the array in cache.

Analysis. At worst, this algorithm is optimal to within 1 memory transfer.

3.3 Example 2: Binary Search

Consider performing a binary search on an array using the traditional algorithm.

Cost. The total number of memory transfers is

$$\begin{aligned} \text{cost} &= \lg \frac{N}{B} && (1) \\ &= \lg N - \lg B. && (2) \end{aligned}$$

Analysis. This cost is *not* optimal. The optimal cost is $\log_B N$. We achieve this performance bound using B-trees (which are out of the scope of this lecture).

3.4 Example 3: Recursive Matrix Multiplication

Consider the recursive algorithm where $C = AB$. The recursive method, breaks each matrix into four parts as such:

$$\begin{aligned}
\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \\
&= \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}.
\end{aligned}$$

Layout of Matrix in Memory. We first consider how to lay the matrix out in memory. For the bad serial algorithm presented above, A being stored as row-major and B being stored as column-major would be good.

For the recursive algorithm, however, a better layout exists in Figure 6.

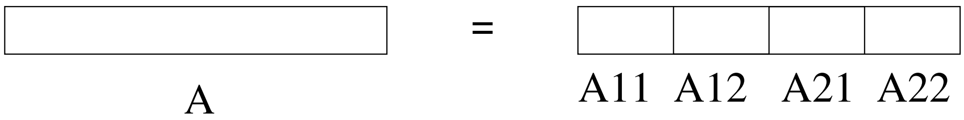

Figure 6: Representation of Matrix in memory.

The above representation has been referred to as ***Mortan Order***, and is similar to a *space-filling curve* or a *Hilbert curve.*

Cost.

Derivation of Recurrence Equation. We express this algorithm by the recurrence:

$$T(N) = 8T\left(\frac{N}{2}\right) + O\left(\left\lceil \frac{N^2}{B} \right\rceil\right).$$

The second term of the recurrence represents the number of memory transfers needed to perform the addition, as presented in Lecture 2. Adding N^2 elements in order causes $\lceil N^2/B \rceil$ memory transfers, thus it is $O\left(\lceil N^2/B \rceil\right)$.

Base Cases. We analyze the two base cases.

The first base case is,

$$T(\sqrt{B}) = O(1).$$

In this case, the entire matrix fits within one cache block.

The second base case is,

$$T(c\sqrt{M}) = O\left(\frac{M}{B}\right),$$

for a sufficiently small c such that all 3 matrices fit in cache. The result is $O(M/B)$ because the data is brought into memory. All other computation is performed in cache.

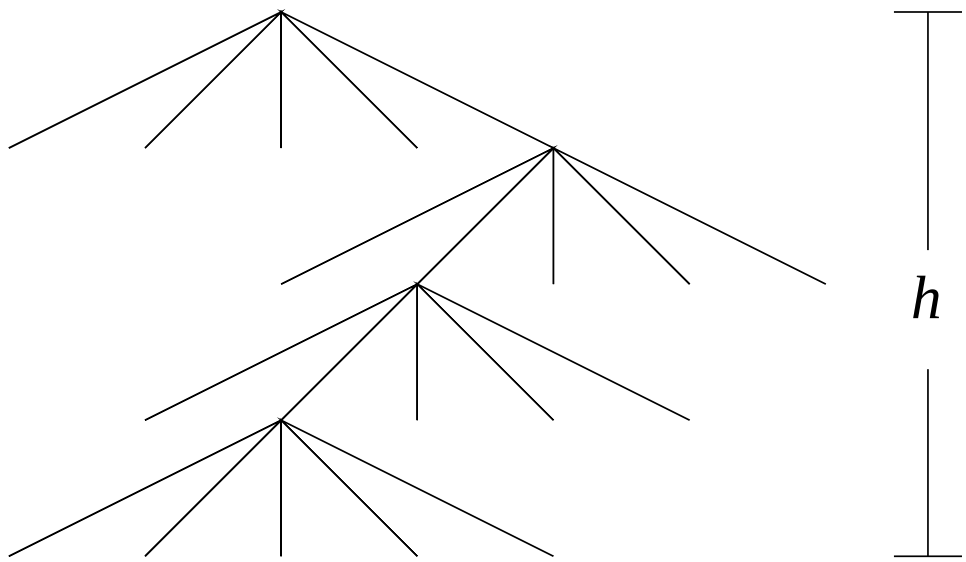

Figure 7: Recursion Tree Sketch. $h = \lg\left(\frac{N}{\sqrt{M}}\right)$

Solving the Recurrence Equation. We solve this recurrence using a tree instead of the Master Theorem. A sketch is in Figure 7.

The cost of the root is $O\left(\lceil N^2/B \rceil\right)$. We traverse the tree until we reach problems of size $O(\sqrt{M})$. The height of this tree is :

$$h = \lg\left(\frac{N}{O(\sqrt{M})}\right).$$

Thus, the number of leaves ℓ is:

$$\ell = 8^{\lg\left(\frac{N}{O(\sqrt{M})}\right)} \tag{3}$$

$$= O\left(\left(\frac{N}{\sqrt{M}}\right)^3\right). \tag{4}$$

The cost per leaf $= \Theta(M/B)$, thus the cost of all leaves is:

$$\text{cost of all leaves} = \Theta\left(\frac{M}{B}\frac{N^3}{M^{\frac{3}{2}}}\right) \tag{5}$$

$$= \Theta\left(\frac{N^3}{B\sqrt{M}}\right). \tag{6}$$

$$\tag{7}$$

Thus, the cost of the root is,

$$\text{cost of root} = \Theta\left(\left\lceil\frac{N^2}{B}\right\rceil\right). \tag{8}$$

Asymptotically, the leaves dominate the cost. Thus:

$$\text{Total cost} = \Theta\left(\frac{N^3}{B\sqrt{M}}\right).$$

Analysis. The cost of this cache-oblivious algorithm is optimal. It can be proved using pebble games, but the proof is out of the scope of the lecture.

References

[AMD03] AMD. Software optimization guide for AMD Athlon 64 and AMD Opteron processors. Technical report, AMD, 2003.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, October 1999.

[MIT03] Code for sort algorithms. MIT 6.895: Handout 3, 2003.