Applied Algorithms Lec 4: External Memory Model

Sam McCauley

September 16, 2025

Williams College

Admin

- Assignment 1 due Thursday night
- TA Hours Tomorrow (Wed) 7–9pm
  - New: Mon 7–8:30pm
- Some reading today! Optional/potentially useful for reference. We don't cover the topic in exactly the same way
  - For ex: we'll have $K = 1$; no distribution sort; no B-trees
- Handout from me also posted with examples of the external memory model

C Debugging: Step by Step

- Make sure your code editor is happy with your code

C Debugging: Step by Step

- Make sure your code editor is happy with your code
- Then: run `gcc` (probably using `make`)

C Debugging: Step by Step

- Make sure your code editor is happy with your code
- Then: run `gcc` (probably using `make`)
- Then: run `valgrind`

C Debugging: Step by Step

- Make sure your code editor is happy with your code
- Then: run `gcc` (probably using `make`)
- Then: run `valgrind`
  - Then: `valgrind test.out smallData.txt` (or larger datasets)

C Debugging: Step by Step

- Make sure your code editor is happy with your code
- Then: run `gcc` (probably using `make`)
- Then: run `valgrind`
  - Then: `valgrind test.out smallData.txt` (or larger datasets)
  - It's slow. But will (often) literally just tell you every memory-based bug in your program. Note: it will complain if you don't `free()`

C Debugging: Step by Step

- Make sure your code editor is happy with your code
- Then: run `gcc` (probably using `make`)
- Then: run `valgrind`
  - Then: `valgrind test.out smallData.txt` (or larger datasets)
  - It's slow. But will (often) literally just tell you every memory-based bug in your program. Note: it will complain if you don't `free()`
  - If you run `make clean` and `make debug` so `valgrind` will give you line-by-line pointers

C Debugging: Step by Step

- Make sure your code editor is happy with your code
- Then: run `gcc` (probably using `make`)
- Then: run `valgrind`
  - Then: `valgrind test.out smallData.txt` (or larger datasets)
  - It's slow. But will (often) literally just tell you every memory-based bug in your program. Note: it will complain if you don't `free()`
  - If you run `make clean` and `make debug` so `valgrind` will give you line-by-line pointers
  - To check for memory leaks: `valgrind --leak-check=full test.out testData.txt timeData.txt`

C Debugging: Step by Step

- Make sure your code editor is happy with your code
- Then: run `gcc` (probably using `make`)
- Then: run `valgrind`
  - Then: `valgrind test.out smallData.txt` (or larger datasets)
  - It's slow. But will (often) literally just tell you every memory-based bug in your program. Note: it will complain if you don't `free()`
  - If you run `make clean` and `make debug` so `valgrind` will give you line-by-line pointers
  - To check for memory leaks: `valgrind --leak-check=full test.out testData.txt timeData.txt`
- Then normal debugging with `gdb` etc.

External Memory Model

Measuring cache misses

- Cache performance is often *more important* than number of operations
- But algorithmic analysis measures number of operations
- Can we algorithmically examine the cache performance of a program?
- Yes: with the external memory model

What do we want out of this model?

- Simple, but able to capture major performance considerations
- Parameters for the model? How can we make it universal across computers that may have very different cache parameters?
    - Answer: we'll use parameters. (The exact size of cache, and a cache line, can *drastically* affect algorithmic performance.)
- Do we want asymptotics? Worst case?
    - Yes!

External memory model basics

- Cache of size M

External memory model basics

- Cache of size M
  - Usually assume that $M = \Omega(\log n)$; often bigger in practice

External memory model basics

- Cache of size M
  - Usually assume that $M = \Omega(\log n)$; often bigger in practice
- Cache line of size B

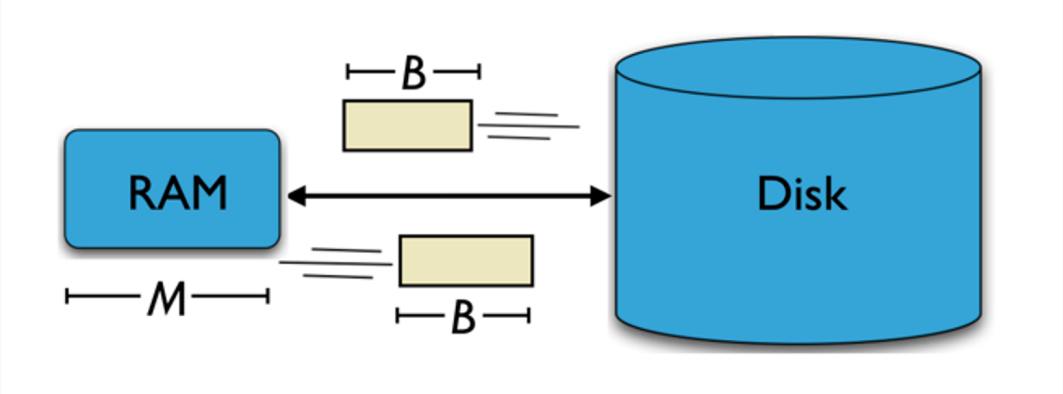
External memory model basics

- Cache of size M
  - Usually assume that $M = \Omega(\log n)$; often bigger in practice
- Cache line of size B
- Computation is free: *only* count number of “cache misses.” Can perform arbitrary computation on items in cache.

External memory model basics

- Cache of size M
  - Usually assume that $M = \Omega(\log n)$; often bigger in practice
- Cache line of size B
- Computation is free: *only* count number of "cache misses." Can perform arbitrary computation on items in cache.
- We will say something like "$O(n/B)$ cache misses" rather than "$O(n)$ operations" to emphasize the model.

External Memory Model Basics



Transferring *B* *consecutive* items to/from the disk costs 1. Can only store *M* things in cache.

Memory Evictions

- Can only hold *M* items in cache!

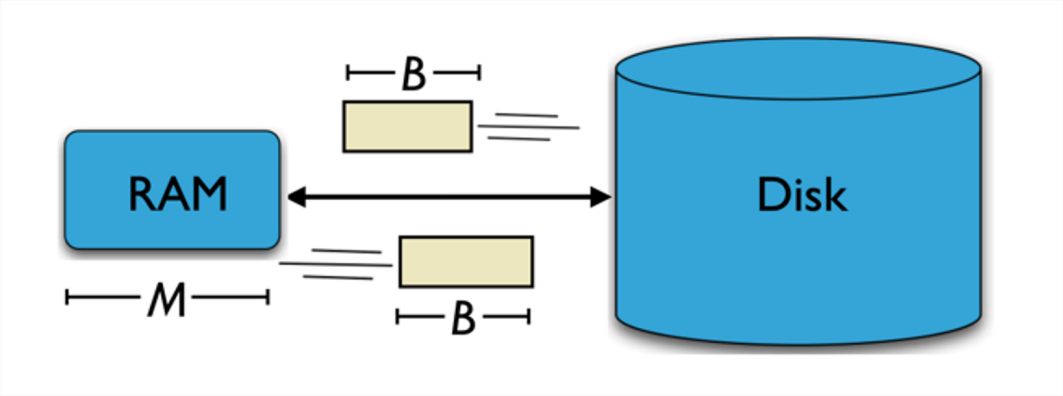
Memory Evictions

- Can only hold *M* items in cache!
- So when we bring *B* in, need to write *B* items back to disk. (We can bring them in later if we need them again)

Memory Evictions

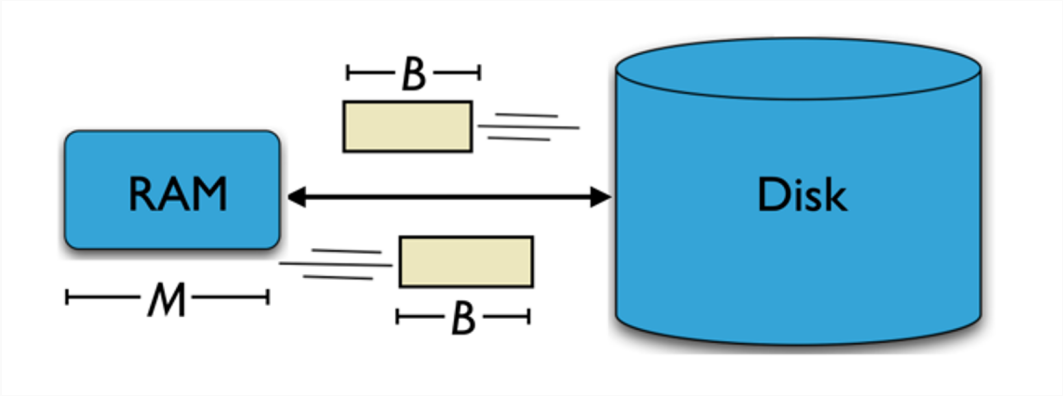
- Can only hold M items in cache!
- So when we bring B in, need to write B items back to disk. (We can bring them in later if we need them again)
- Assume that the computer does this optimally.
  - Reasonable; it's really good at it. Very cool algorithms behind this!

Vocabulary



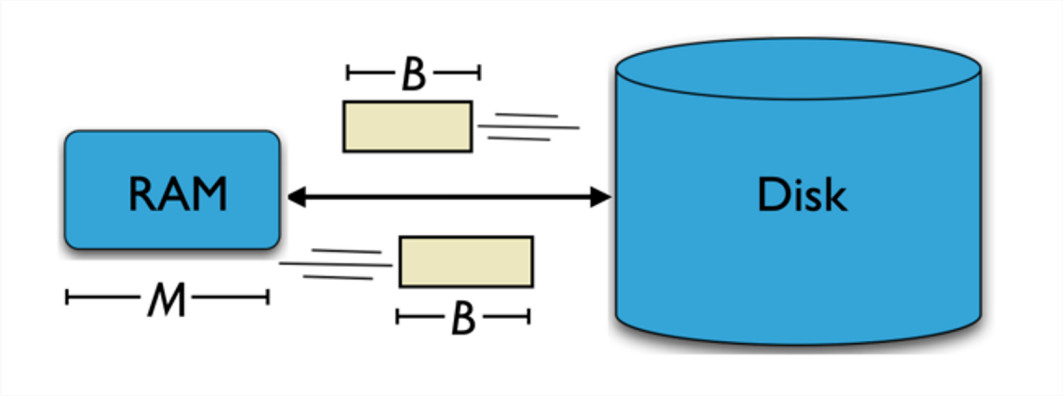
- “Cache” of size *M*; “disk” of unlimited size

Vocabulary



- “Cache” of size *M*; “disk” of unlimited size
- With the cost of one “cache miss” can bring in *B* consecutive items
  - (Also called “memory access” or “I/Os”; I will try not to use those terms.)

Vocabulary



- “Cache” of size *M*; “disk” of unlimited size
- With the cost of one “cache miss” can bring in *B* consecutive items
  - (Also called “memory access” or “I/Os”; I will try not to use those terms.)
- These *B* items are called a “block” or a “cache line”.

Let's revisit `sortedlinkedlist.c`

- What is the cost of our algorithm in the external memory model if the items are stored in order?

Let's revisit `sortedlinkedlist.c`

- What is the cost of our algorithm in the external memory model if the items are stored in order?
- Answer: $O(n/B)$

Let's revisit `sortedlinkedlist.c`

- What is the cost of our algorithm in the external memory model if the items are stored in order?
- Answer: $O(n/B)$
- What is the cost of our algorithm in the external memory model if the items have stride $B + 1$?

Let's revisit `sortedlinkedlist.c`

- What is the cost of our algorithm in the external memory model if the items are stored in order?
- Answer: $O(n/B)$
- What is the cost of our algorithm in the external memory model if the items have stride $B + 1$?
- Answer: $O(n)$

Let's revisit `sortedlinkedlist.c`

- What is the cost of our algorithm in the external memory model if the items are stored in order?
- Answer: $O(n/B)$
- What is the cost of our algorithm in the external memory model if the items have stride $B + 1$?
- Answer: $O(n)$
- The external memory model predicts the real-world slowdown of this process.

Let's revisit `sortedlinkedlist.c`

- What is the cost of our algorithm in the external memory model if the items are stored in order?
- Answer: $O(n/B)$
- What is the cost of our algorithm in the external memory model if the items have stride $B + 1$?
- Answer: $O(n)$
- The external memory model predicts the real-world slowdown of this process.
- (Actual performance is *better* in this case: we get a slowdown of ≈ 1.2, whereas the number of nodes in a cache line is 4. Last year it was *worse* than predicted. I imagine that this is due to prefetching???)

Finding the minimum element in an unsorted array

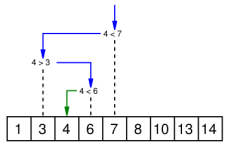
- How many cache misses in the external memory model?

Finding the minimum element in an unsorted array

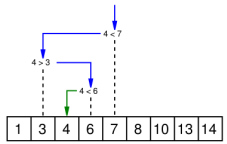
- How many cache misses in the external memory model?

- Answer: $O(n/B)$

Binary search?

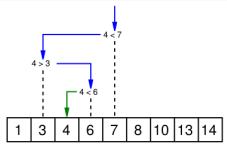


Binary search?



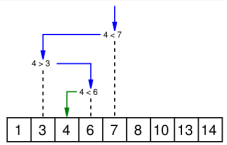
- Does binary search seem cache efficient? Discuss in pairs what its cache efficiency should be in the external memory model.

Binary search?



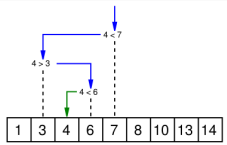
- Does binary search seem cache efficient? Discuss in pairs what its cache efficiency should be in the external memory model.
- What is the recurrence for binary search in terms of number of operations?

Binary search?



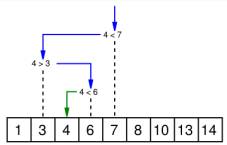
- Does binary search seem cache efficient? Discuss in pairs what its cache efficiency should be in the external memory model.
- What is the recurrence for binary search in terms of number of operations?
- What is the recurrence for binary search in terms of the number of cache misses?

Binary search?



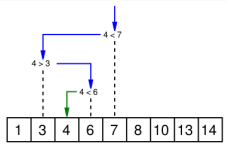
- Does binary search seem cache efficient? Discuss in pairs what its cache efficiency should be in the external memory model.
- What is the recurrence for binary search in terms of number of operations?
- What is the recurrence for binary search in terms of the number of cache misses?
- Each recursive call takes 1 cache miss—until we reach an array of size $O(B)$, after which we are done

Binary search?



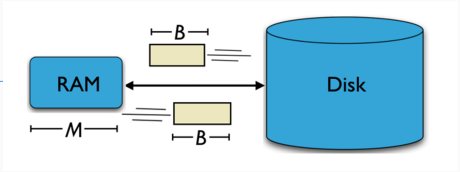
- Does binary search seem cache efficient? Discuss in pairs what its cache efficiency should be in the external memory model.
- What is the recurrence for binary search in terms of number of operations?
- What is the recurrence for binary search in terms of the number of cache misses?
- Each recursive call takes 1 cache miss—until we reach an array of size $O(B)$, after which we are done
- Base case: can perform *all* operations on B items with only 1 cache miss

Binary search?



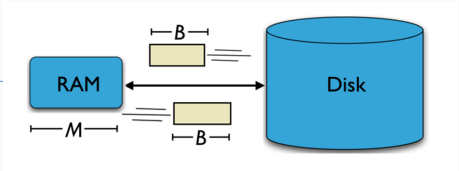
- Does binary search seem cache efficient? Discuss in pairs what its cache efficiency should be in the external memory model.
- What is the recurrence for binary search in terms of number of operations?
- What is the recurrence for binary search in terms of the number of cache misses?
- Each recursive call takes 1 cache miss—until we reach an array of size $O(B)$, after which we are done
- Base case: can perform *all* operations on B items with only 1 cache miss
- Total: $O(\log_2(n/B))$ cache misses.

Fitting in Cache



- If you have a sequence of operations on a dataset of size at most *M*, there is no further cost so long as they all stay in cache!

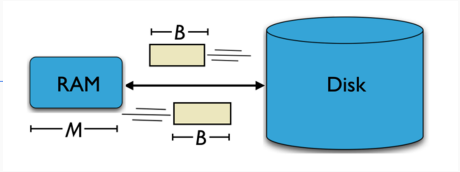
Fitting in Cache



- If you have a sequence of operations on a dataset of size at most M, there is no further cost so long as they all stay in cache!

- $O(M/B)$ to load the items into cache, then all computation is free

Fitting in Cache



- If you have a sequence of operations on a dataset of size at most M, there is no further cost so long as they all stay in cache!
- $O(M/B)$ to load the items into cache, then all computation is free
- Real-world time: what if instead of a linked list of 100 million items, we repeatedly access a linked list of 100 thousand items?
  - `smallunsortedlist.c`

Why does the external memory model make sense?

- Simple model that captures *one level* of the memory hierarchy

Why does the external memory model make sense?

- Simple model that captures *one level* of the memory hierarchy
- Idea: usually one level has by far the largest cost.

Why does the external memory model make sense?

- Simple model that captures *one level* of the memory hierarchy
- Idea: usually one level has by far the largest cost.
    - Small programs may be dominated by L1 cache misses

Why does the external memory model make sense?

- Simple model that captures *one level* of the memory hierarchy
- Idea: usually one level has by far the largest cost.
  - Small programs may be dominated by L1 cache misses
  - Larger programs it may be by L3 cache misses

Why does the external memory model make sense?

- Simple model that captures *one level* of the memory hierarchy
- Idea: usually one level has by far the largest cost.
  - Small programs may be dominated by L1 cache misses
  - Larger programs it may be by L3 cache misses
- External memory model zooms in on one crucial level of the memory hierarchy (with particular B, M); gives asymptotics for how well we do on that level.

Question about External Memory Model Basics?

Matrix Multiplication in External Memory

Matrix Multiplication Reminder

- Given two $n \times n$ matrices A, B

Matrix Multiplication Reminder

- Given two $n \times n$ matrices A, B
- Want to compute their product C:

Matrix Multiplication Reminder

- Given two $n \times n$ matrices A, B
- Want to compute their product C:
- $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$

Matrix Multiplication Reminder

- Given two $n \times n$ matrices A, B
- Want to compute their product C:
- $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$

Example:

$$\begin{bmatrix} 1 & 2 \\ 8 & -1 \end{bmatrix} \times \begin{bmatrix} 2 & 3 \\ -2 & 7 \end{bmatrix} = \begin{bmatrix} -2 & 17 \\ 18 & 17 \end{bmatrix}$$

Compute Product Directly

```
for i = 1 to n:
  for j = 1 to n:
    for k = 1 to n:
      C[i][j] += A[i][k] + B[k][j]
```

- Recall: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$

Compute Product Directly

```
for i = 1 to n:
  for j = 1 to n:
    for k = 1 to n:
      C[i][j] += A[i][k] + B[k][j]
```

- Recall: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$
- How many cache misses does this take?

Compute Product Directly

```
for i = 1 to n:
  for j = 1 to n:
    for k = 1 to n:
      C[i][j] += A[i][k] + B[k][j]
```

- Recall: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$
- How many cache misses does this take?
- Assume matrices are stored in row-major order.
  - First: assume $M > 3n^2$

Compute Product Directly

```
for i = 1 to n:
  for j = 1 to n:
    for k = 1 to n:
      C[i][j] += A[i][k] + B[k][j]
```

- Recall: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$
- How many cache misses does this take?
- Assume matrices are stored in row-major order.
  - First: assume $M > 3n^2$ Then we can fit A, B, and C in cache; $O(n^2/B)$ cache misses

Compute Product Directly

```
for i = 1 to n:
  for j = 1 to n:
    for k = 1 to n:
      C[i][j] += A[i][k] + B[k][j]
```

- Recall: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$
- How many cache misses does this take?
- Assume matrices are stored in row-major order.
  - First: assume $M > 3n^2$ Then we can fit A, B, and C in cache; $O(n^2/B)$ cache misses
  - What if $M < n^2$?

Compute Product Directly

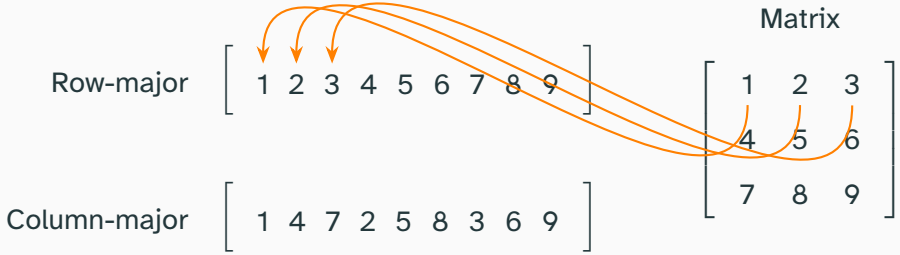
```
for i = 1 to n:
  for j = 1 to n:
    for k = 1 to n:
      C[i][j] += A[i][k] + B[k][j]
```

- Recall: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$
- How many cache misses does this take?
- Assume matrices are stored in row-major order.
  - First: assume $M > 3n^2$ Then we can fit A, B, and C in cache; $O(n^2/B)$ cache misses
  - What if $M < n^2$?
  - Answer: $O(n^3)$ cache misses. Every operation requires a cache miss for matrix B.

Any ideas for how to improve this?

- One idea: transpose *B* (store in column-major order)
  - A good idea; works well!
  - What is the cache efficiency?

```
1  for i = 1 to n:
2    for j = 1 to n:
3      for k = 1 to n:
4        C[i][j] += A[i][k]
             + B[k][j]
```



Any ideas for how to improve this?

- One idea: transpose *B* (store in column-major order)
  - A good idea; works well!
  - What is the cache efficiency?

```
1  for i = 1 to n:
2    for j = 1 to n:
3      for k = 1 to n:
4        C[i][j] += A[i][k]
             + B[k][j]
```

Any ideas for how to improve this?

- One idea: transpose *B* (store in column-major order)
  - A good idea; works well!
  - What is the cache efficiency?

```
1  Convert B to column-major
       order
2  for i = 1 to n:
3    for j = 1 to n:
4      for k = 1 to n:
5        C[i][j] += A[i][k]
             + B[k][j]
```

Cache misses for the transposing:

Any ideas for how to improve this?

- One idea: transpose B (store in column-major order)
  - A good idea; works well!
  - What is the cache efficiency?

```
1  Convert B to column-major
       order
2  for i = 1 to n:
3    for j = 1 to n:
4      for k = 1 to n:
5        C[i][j] += A[i][k]
             + B[k][j]
```

Cache misses for the transposing: $O(n^2)$ cache misses

Any ideas for how to improve this?

- One idea: transpose B (store in column-major order)
  - A good idea; works well!
  - What is the cache efficiency?

```
1  Convert B to column-major
       order
2  for i = 1 to n:
3    for j = 1 to n:
4      for k = 1 to n:
5        C[i][j] += A[i][k]
             + B[k][j]
```

Cache misses for the transposing: $O(n^2)$ cache misses

Each time I have a cache miss for $B[k][j]$, no further cache miss until $B[k+B][j]$.

Any ideas for how to improve this?

- One idea: transpose B (store in column-major order)
  - A good idea; works well!
  - What is the cache efficiency?

```
1  Convert B to column-major
       order
2  for i = 1 to n:
3    for j = 1 to n:
4      for k = 1 to n:
5        C[i][j] += A[i][k]
             + B[k][j]
```

Cache misses for the transposing: $O(n^2)$ cache misses

Each time I have a cache miss for $B[k][j]$, no further cache miss until $B[k+B][j]$.

Each time I have a cache miss accessing $C[i][j]$, no further cache miss until $C[i][j+B]$. Each time I have a cache miss for $A[i][k]$, no further cache miss until $A[i][k+B]$.

Any ideas for how to improve this?

- One idea: transpose B (store in column-major order)
  - A good idea; works well!
  - What is the cache efficiency?

```
1  Convert B to column-major
       order
2  for i = 1 to n:
3    for j = 1 to n:
4      for k = 1 to n:
5        C[i][j] += A[i][k]
             + B[k][j]
```

Cache misses for the transposing: $O(n^2)$ cache misses

Each time I have a cache miss for $B[k][j]$, no further cache miss until $B[k+B][j]$.

Each time I have a cache miss accessing $C[i][j]$, no further cache miss until $C[i][j+B]$. Each time I have a cache miss for $A[i][k]$, no further cache miss until $A[i][k+B]$.

Total: $n^3/B + n^2/B + n^3/B = O(n^3/B)$ cache misses.

Any ideas for how to improve this?

- Another idea: swap the loops!

Original:

```
for i = 1 to n:
  for j = 1 to n:
    for k = 1 to n:
      C[i][j] += A[i][k] + B[k][j]
```

Improved(?):

```
for i = 1 to n:
  for k = 1 to n:
    for j = 1 to n:
      C[i][j] += A[i][k] + B[k][j]
```

Any ideas for how to improve this?

- How many cache misses is this?

```
for i = 1 to n:
  for k = 1 to n:
    for j = 1 to n:
      C[i][j] += A[i][k] + B[k][j]
```

Any ideas for how to improve this?

- How many cache misses is this?

```
1 for i = 1 to n:
2   for k = 1 to n:
3     for j = 1 to n:
4       C[i][j] += A[i][k] + B[k][j]
```

- Let's say $A[i][k]$ is a cache miss. No more cache misses until $A[i][k']$ with $k' = k + B$.

Any ideas for how to improve this?

- How many cache misses is this?

```
1 for i = 1 to n:
2   for k = 1 to n:
3     for j = 1 to n:
4       C[i][j] += A[i][k] + B[k][j]
```

- Let's say $A[i][k]$ is a cache miss. No more cache misses until $A[i][k']$ with $k' = k + B$.
- Let's say $B[k][j]$ is a cache miss. No more cache misses until $B[i][j']$ with $j' = j + B$.

Any ideas for how to improve this?

- How many cache misses is this?

```
for i = 1 to n:
  for k = 1 to n:
    for j = 1 to n:
      C[i][j] += A[i][k] + B[k][j]
```

- Let's say $A[i][k]$ is a cache miss. No more cache misses until $A[i][k']$ with $k' = k + B$.
- Let's say $B[k][j]$ is a cache miss. No more cache misses until $B[i][j']$ with $j' = j + B$.
- Let's say $C[i][j]$ is a cache miss. No more cache misses until $C[i][j']$ with $j' = j + B$.

Any ideas for how to improve this?

- How many cache misses is this?

```
for i = 1 to n:
  for k = 1 to n:
    for j = 1 to n:
      C[i][j] += A[i][k] + B[k][j]
```

- Let's say $A[i][k]$ is a cache miss. No more cache misses until $A[i][k']$ with $k' = k + B$.
- Let's say $B[k][j]$ is a cache miss. No more cache misses until $B[i][j']$ with $j' = j + B$.
- Let's say $C[i][j]$ is a cache miss. No more cache misses until $C[i][j']$ with $j' = j + B$.
- Sum up each: $O(n^3/B)$ total

Any ideas for how to improve this?

- How many cache misses is this?

```
for i = 1 to n:
  for k = 1 to n:
    for j = 1 to n:
      C[i][j] += A[i][k] + B[k][j]
```

- Let's say $A[i][k]$ is a cache miss. No more cache misses until $A[i][k']$ with $k' = k + B$.
- Let's say $B[k][j]$ is a cache miss. No more cache misses until $B[i][j']$ with $j' = j + B$.
- Let's say $C[i][j]$ is a cache miss. No more cache misses until $C[i][j']$ with $j' = j + B$.
- Sum up each: $O(n^3/B)$ total
- Is this worth doing?

Yep!

I am given two functions for finding the product of two matrices:

```
void MultiplyMatrices_1(int **a, int **b, int **c, int n){
     for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
             for (int k = 0; k < n; k++)
                 c[i][j] = c[i][j] + a[i][k]*b[k][j];
 }

void MultiplyMatrices_2(int **a, int **b, int **c, int n){
     for (int i = 0; i < n; i++)
         for (int k = 0; k < n; k++)
             for (int j = 0; j < n; j++)
                 c[i][j] = c[i][j] + a[i][k]*b[k][j];
}
```

I ran and profiled two executables using `gprof`, each with identical code except for this function. The second of these is significantly (about 5 times) faster for matrices of size 2048 x 2048. Any ideas as to why?

c | algorithm | matrix | matrix-multiplication | gprof

We haven’t used the cache yet!

We haven’t used the cache yet!

- We haven’t really stored anything in the cache except the current cache line! (Doesn’t matter how big the cache is.)
- No *M*s in any running times—except when the whole problem fits in cache

We haven’t used the cache yet!

- We haven’t really stored anything in the cache except the current cache line! (Doesn’t matter how big the cache is.)
- No *M*s in any running times—except when the whole problem fits in cache
- Why? All algorithms so far have read the data once and then thrown it away.

We haven't used the cache yet!

- We haven't really stored anything in the cache except the current cache line! (Doesn't matter how big the cache is.)
- No *M*s in any running times—except when the whole problem fits in cache
- Why? All algorithms so far have read the data once and then thrown it away.
- Goal: bring items into cache so that we can perform *many* computations on them before writing them back.

We haven't used the cache yet!

- We haven't really stored anything in the cache except the current cache line! (Doesn't matter how big the cache is.)
- No Ms in any running times—except when the whole problem fits in cache
- Why? All algorithms so far have read the data once and then thrown it away.
- Goal: bring items into cache so that we can perform *many* computations on them before writing them back.
- Note: can't do this with linear scan. $O(n/B)$ is optimal. But we did do this with `smallunsortedlinkedlist.c`

Blocking

- Standard technique for improving cache performance of algorithms.

Blocking

- Standard technique for improving cache performance of algorithms.
- Remember: cache efficiency can get WAY better when the problem fits in cache. Let's find *subproblems* that can fit in cache.

Blocking

- Standard technique for improving cache performance of algorithms.
- Remember: cache efficiency can get WAY better when the problem fits in cache. Let's find *subproblems* that can fit in cache.
- Idea: break problems into subproblems of size $O(M)$

Blocking

- Standard technique for improving cache performance of algorithms.
- Remember: cache efficiency can get WAY better when the problem fits in cache. Let's find *subproblems* that can fit in cache.
- Idea: break problems into subproblems of size $O(M)$
    - Can solve any such problem in $O(M/B)$ cache misses

Blocking

- Standard technique for improving cache performance of algorithms.
- Remember: cache efficiency can get WAY better when the problem fits in cache. Let's find *subproblems* that can fit in cache.
- Idea: break problems into subproblems of size $O(M)$
  - Can solve any such problem in $O(M/B)$ cache misses
  - Efficiently combine them for a cache-efficient solution

Blocked Matrix Multiplication

- Split A, B, and C into blocks of size $M/3$
  - $\sqrt{M/3} \times \sqrt{M/3}$ matrices
  - Really want blocks with size $T = \lfloor \sqrt{M/3} \rfloor$. Assume that T divides n for now so there's no rounding

Blocked Matrix Multiplication

- Split A, B, and C into blocks of size $M/3$
  - $\sqrt{M/3} \times \sqrt{M/3}$ matrices
  - Really want blocks with size $T = \lfloor \sqrt{M/3} \rfloor$. Assume that T divides n for now so there's no rounding

- Multiply blocks one at a time

Decomposing matrices into blocks

Classic result: if we treat the blocks as single elements of the matrices, and multiply (and add) them as normal, we obtain the same result as we would have in normal matrix multiplication.

Decomposing matrices into blocks

Classic result: if we treat the blocks as single elements of the matrices, and multiply (and add) them as normal, we obtain the same result as we would have in normal matrix multiplication.

- This idea is used in recursive matrix multiplication

Decomposing matrices into blocks

Classic result: if we treat the blocks as single elements of the matrices, and multiply (and add) them as normal, we obtain the same result as we would have in normal matrix multiplication.

- This idea is used in recursive matrix multiplication
- And Strassen's algorithm for matrix multiplication

Decomposing matrices into blocks

Example: Recall how to multiply $2x2$ matrices:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{bmatrix}$$

Decomposing matrices into blocks

Example: Recall how to multiply $2x2$ matrices:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{bmatrix}$$

We can use this principle to multiply two larger matrices.

Decomposing matrices into blocks

Example: Recall how to multiply $2x2$ matrices:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{bmatrix}$$

We can use this principle to multiply two larger matrices.

$$\begin{bmatrix} 17 & 15 & 20 & 4 \\ 15 & 3 & 20 & 8 \\ 1 & 10 & 15 & 2 \\ 3 & 19 & 3 & 14 \end{bmatrix} \cdot \begin{bmatrix} 4 & 12 & 9 & 1 \\ 4 & 6 & 11 & 2 \\ 13 & 18 & 8 & 20 \\ 3 & 11 & 18 & 9 \end{bmatrix} =$$

Decomposing matrices into blocks

Example: Recall how to multiply 2x2 matrices:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{bmatrix}$$

We can use this principle to multiply two larger matrices.

$$\begin{bmatrix} 17 & 15 & 20 & 4 \\ 15 & 3 & 20 & 8 \\ 1 & 10 & 15 & 2 \\ 3 & 19 & 3 & 14 \end{bmatrix} \cdot \begin{bmatrix} 4 & 12 & 9 & 1 \\ 4 & 6 & 11 & 2 \\ 13 & 18 & 8 & 20 \\ 3 & 11 & 18 & 9 \end{bmatrix} =$$

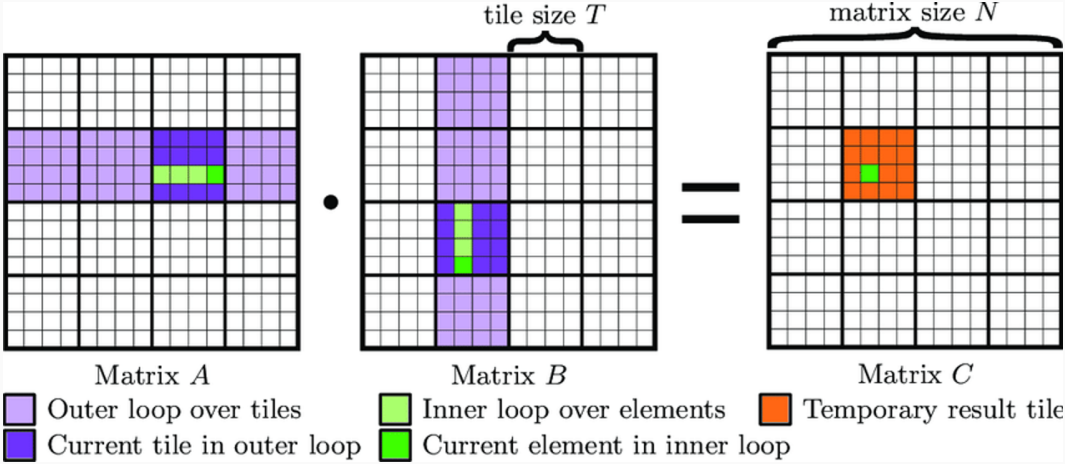
$$\begin{bmatrix} \begin{bmatrix} 17 & 15 \\ 15 & 3 \end{bmatrix} \cdot \begin{bmatrix} 4 & 12 \\ 4 & 6 \end{bmatrix} + \begin{bmatrix} 20 & 4 \\ 20 & 8 \end{bmatrix} \cdot \begin{bmatrix} 13 & 8 \\ 3 & 11 \end{bmatrix} & \begin{bmatrix} 17 & 15 \\ 15 & 3 \end{bmatrix} \cdot \begin{bmatrix} 9 & 1 \\ 11 & 2 \end{bmatrix} + \begin{bmatrix} 20 & 4 \\ 20 & 8 \end{bmatrix} \cdot \begin{bmatrix} 8 & 20 \\ 18 & 9 \end{bmatrix} \\ \begin{bmatrix} 1 & 10 \\ 3 & 19 \end{bmatrix} \cdot \begin{bmatrix} 4 & 12 \\ 4 & 6 \end{bmatrix} + \begin{bmatrix} 15 & 2 \\ 3 & 14 \end{bmatrix} \cdot \begin{bmatrix} 13 & 8 \\ 3 & 11 \end{bmatrix} & \begin{bmatrix} 1 & 10 \\ 3 & 19 \end{bmatrix} \cdot \begin{bmatrix} 9 & 1 \\ 11 & 2 \end{bmatrix} + \begin{bmatrix} 15 & 2 \\ 3 & 14 \end{bmatrix} \cdot \begin{bmatrix} 8 & 20 \\ 18 & 9 \end{bmatrix} \end{bmatrix}$$

Blocked Matrix Multiplication

- Decompose matrix into blocks of length T (where $T^2 = M/3$)

Blocked Matrix Multiplication

- Decompose matrix into blocks of length T (where $T^2 = M/3$)
- Do a normal $n/T \times n/T$ matrix multiplication



Blocked Matrix Multiplication Pseudocode

```
MatrixMultiply(A, B, C, n, T):
    for i = 1 to n/T:
      for j = 1 to n/T:
        for k = 1 to n/T:
          A' = TxT matrix with upper left corner A[Ti][Tk]
          B' = TxT matrix with upper left corner B[Tk][Tj]
          C' = TxT matrix with upper left corner C[Ti][Tj]
          BlockMultiply(A', B', C', T)

BlockMultiply(A, B, C, n):
    for i = 1 to n:
        for j = 1 to n:
            for k = 1 to n:
                C[i][j] += A[i][k] + B[k][j]
```

Blocked Matrix Multiplication Pseudocode

```
MatrixMultiply(A, B, C, n, T):
    for i = 1 to n/T:
      for j = 1 to n/T:
        for k = 1 to n/T:
          A' = TxT matrix with upper left corner A[Ti][Tk]
          B' = TxT matrix with upper left corner B[Tk][Tj]
          C' = TxT matrix with upper left corner C[Ti][Tj]
          BlockMultiply(A', B', C', T)

BlockMultiply(A, B, C, n):
    for i = 1 to n:
        for j = 1 to n:
            for k = 1 to n:
                C[i][j] += A[i][k] + B[k][j]
```

Let's analyze the cost of this algorithm in the EM model together on the board!

Analysis

- Creating A', B', C' and passing them to `BlockMultiply` all can be done in $O(T^2/B + T)$ cache misses.

Analysis

- Creating A', B', C' and passing them to `BlockMultiply` all can be done in $O(T^2/B + T)$ cache misses. If $B = O(T)$ then we can just write $O(T^2/B)$; let's assume this for simplicity.

Analysis

- Creating A', B', C' and passing them to `BlockMultiply` all can be done in $O(T^2/B + T)$ cache misses. If $B = O(T)$ then we can just write $O(T^2/B)$; let's assume this for simplicity.

- `BlockMultiply` only accesses elements of A', B', C'. Since all three matrices are in cache, it requires zero additional cache misses

Analysis

- Creating A', B', C' and passing them to `BlockMultiply` all can be done in $O(T^2/B + T)$ cache misses. If $B = O(T)$ then we can just write $O(T^2/B)$; let's assume this for simplicity.

- `BlockMultiply` only accesses elements of A', B', C'. Since all three matrices are in cache, it requires zero additional cache misses

- Therefore, our total running time is the number of loop iterations times the cost of a loop. This is $O((n/T)^3 \cdot T^2/B) = O((n/\sqrt{M})^3 \cdot M/B) = O(n^3/B\sqrt{M})$.

Implementation questions!

- What do we do if n is not divisible by T?
  - Easy answer: pad it out! Doesn't change asymptotics.
  - Can carefully make it work without padding as well

Implementation questions!

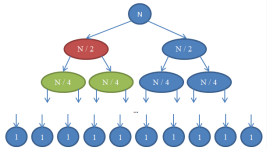
- What do we do if n is not divisible by T?
  - Easy answer: pad it out! Doesn't change asymptotics.
  - Can carefully make it work without padding as well
- How do we figure out M?
  - We can look up the size of the cache on the computer
  - But: this is a simplified model. We don't have a two-level cache and we're ignoring that space is used for other programs, other variables, etc.
  - Experiment! Try different values of M and see what's fastest on a particular machine.

Implementation questions!

- What do we do if n is not divisible by T?
  - Easy answer: pad it out! Doesn't change asymptotics.
  - Can carefully make it work without padding as well
- How do we figure out M?
  - We can look up the size of the cache on the computer
  - But: this is a simplified model. We don't have a two-level cache and we're ignoring that space is used for other programs, other variables, etc.
  - Experiment! Try different values of M and see what's fastest on a particular machine.
- Is blocking actually worthwhile?
  - Yes; it is used all the time to speed up programs with poor cache performance.
  - (Not a panacea; some programs (like linear scan, binary search) can't be blocked.)

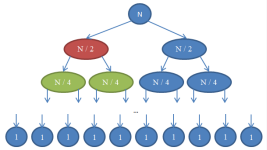
Sorting in External Memory

What about algorithms we know?



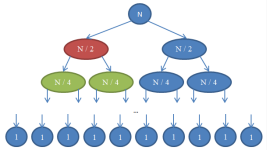
- In pairs: how long does Mergesort take in external memory?

What about algorithms we know?



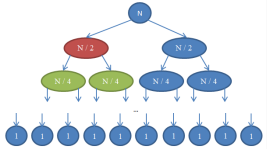
- In pairs: how long does Mergesort take in external memory?
- Merge is $O(n/B)$; base case is when $n = B$, so total is $O(n/B \log_2 n/B)$.

What about algorithms we know?



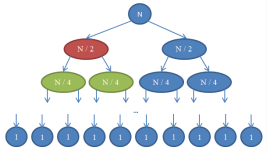
- In pairs: how long does Mergesort take in external memory?
- Merge is $O(n/B)$; base case is when $n = B$, so total is $O(n/B \log_2 n/B)$.
- How about quicksort?

What about algorithms we know?



- In pairs: how long does Mergesort take in external memory?
- Merge is $O(n/B)$; base case is when $n = B$, so total is $O(n/B \log_2 n/B)$.
- How about quicksort?
- Essentially same; partition is $O(n/B)$; total is $O(n/B \log_2 n/B)$.

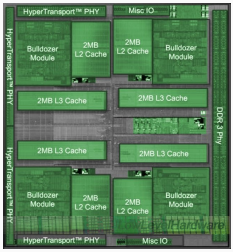
What about algorithms we know?



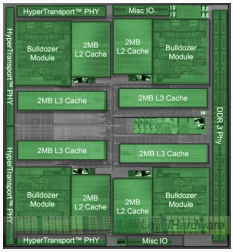
- In pairs: how long does Mergesort take in external memory?
- Merge is $O(n/B)$; base case is when $n = B$, so total is $O(n/B \log_2 n/B)$.
- How about quicksort?
- Essentially same; partition is $O(n/B)$; total is $O(n/B \log_2 n/B)$.
- Seems pretty good! Can we do better?

Using the cache

- Blocking? A little unclear. (We'll come back to this.)



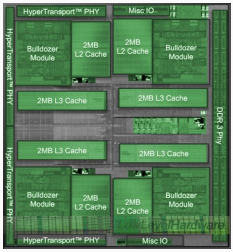
Using the cache



- Blocking? A little unclear. (We'll come back to this.)

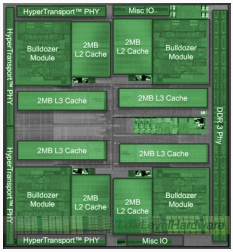
- Does anyone know the sorting lower bound? Where does $n \log n$ come from?

Using the cache



- Blocking? A little unclear. (We'll come back to this.)
- Does anyone know the sorting lower bound? Where does $n \log n$ come from?
- Answer: each time you compare two numbers, can only have two outcomes.

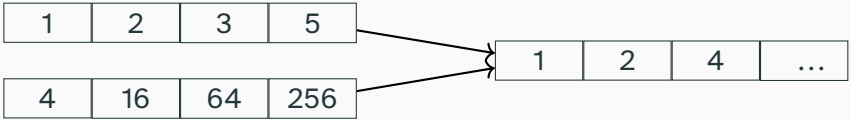
Using the cache



- Blocking? A little unclear. (We'll come back to this.)
- Does anyone know the sorting lower bound? Where does $n \log n$ come from?
- Answer: each time you compare two numbers, can only have two outcomes.
- Each time we bring a cache line into cache, how many more things can we compare it to?

Merge sort reminder

- Divide array into two equal parts
- Recursively sort both parts
- Merge them in $O(n)$ time (and $O(n/B)$ cache misses)



M/B-way merge sort

M/B-way merge sort

- Divide array into M/B equal parts

M/B-way merge sort

- Divide array into M/B equal parts
- Recursively sort all M/B parts

M/B-way merge sort

- Divide array into M/B equal parts
- Recursively sort all M/B parts
- Merge all M/B arrays in $O(n)$ time (and $O(n/B)$ cache misses)

Diagram of M/B-way merge sort

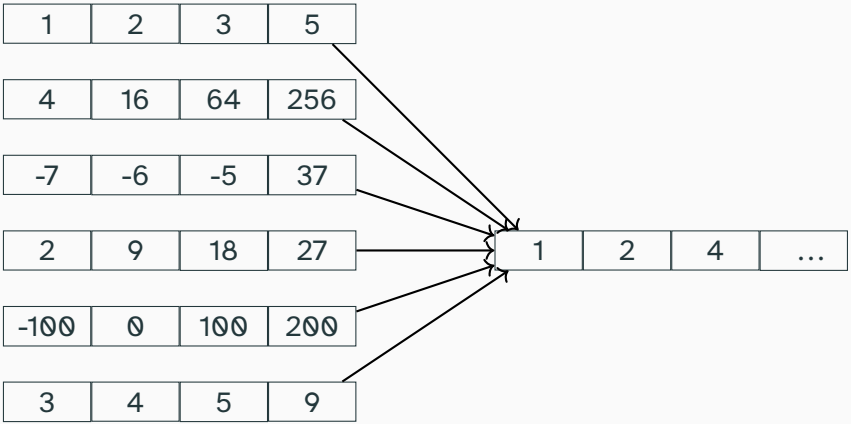


Diagram of M/B-way merge sort

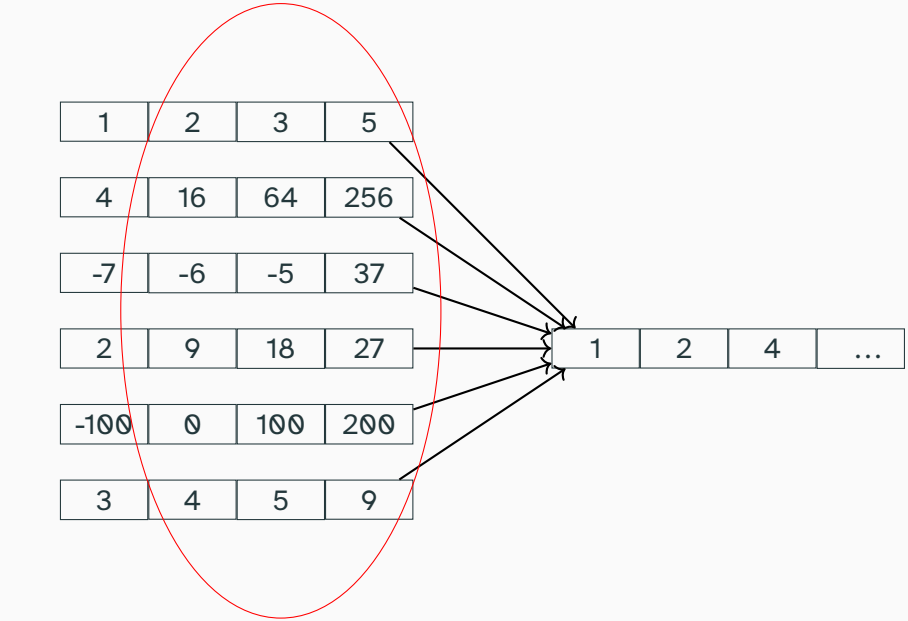
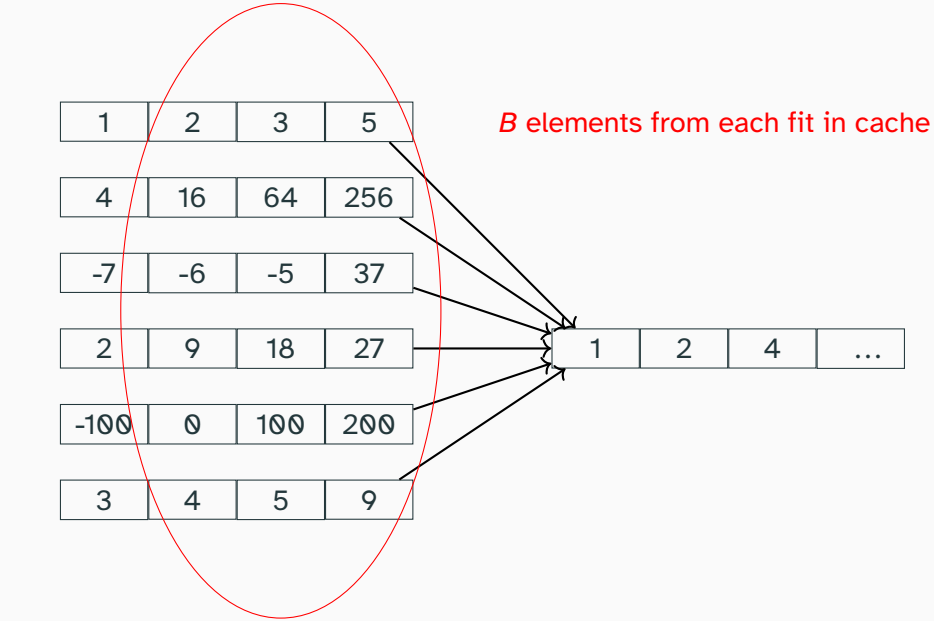


Diagram of M/B-way merge sort



More Detail on merges

- Keep B slots for each array in cache. (M/B arrays so this fits!)

More Detail on merges

- Keep B slots for each array in cache. (M/B arrays so this fits!)
- When all B slots are empty for the array, take B more items from the array in cache.

More Detail on merges

- Keep B slots for each array in cache. (M/B arrays so this fits!)
- When all B slots are empty for the array, take B more items from the array in cache.
- Example on board

Analysis

- Divide array into M/B parts; combine in $O(N/B)$ cache misses.

Analysis

- Divide array into M/B parts; combine in $O(N/B)$ cache misses.
- Recursion:

$$T(N) = T(N/(M/B)) + O(N/B)$$

$$T(B) = O(1)$$

Analysis

- Divide array into M/B parts; combine in $O(N/B)$ cache misses.
- Recursion:

$$T(N) = T(N/(M/B)) + O(N/B)$$

$$T(B) = O(1)$$

- Solves to $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ cache misses

Analysis

- Divide array into M/B parts; combine in $O(N/B)$ cache misses.
- Recursion:

$$T(N) = T(N/(M/B)) + O(N/B)$$

$$T(B) = O(1)$$

- Solves to $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ cache misses
- Optimal!

Useful?

- Can be useful if your data is VERY large

Useful?

- Can be useful if your data is VERY large
- Distribution sort: similar idea, but with Quicksort instead of Mergesort

Useful?

- Can be useful if your data is VERY large
- Distribution sort: similar idea, but with Quicksort instead of Mergesort
- Another method is most popular in practice: Timsort

Useful?

- Can be useful if your data is VERY large
- Distribution sort: similar idea, but with Quicksort instead of Mergesort
- Another method is most popular in practice: Timsort
- Merges a “stack” of runs. Somewhat similar to M/B-way merge sort, achieves strong cache efficiency in practice.

Useful?

- Can be useful if your data is VERY large
- Distribution sort: similar idea, but with Quicksort instead of Mergesort
- Another method is most popular in practice: Timsort
- Merges a “stack” of runs. Somewhat similar to M/B-way merge sort, achieves strong cache efficiency in practice.
- If we have time, let’s talk about engineering a sorting algorithm on the board