Analyzing Data Access of Algorithms and How to Make Them Cache-Friendly?

Kenjiro Taura

Contents

Contents

Motivation

- we've learned data access is an important factor that determines performance of your program

Motivation

- we've learned data access is an important factor that determines performance of your program
- it is thus clearly important to analyze how "good" your algorithm is, in terms of data access performance

Motivation

- we've learned data access is an important factor that determines performance of your program
- it is thus clearly important to analyze how "good" your algorithm is, in terms of data access performance
- we routinely analyze computational complexity of algorithms to predict or explain algorithm performance, but it ignores the differing costs of accessing memory hierarchy (all memory accesses are $O(1)$)

Motivation

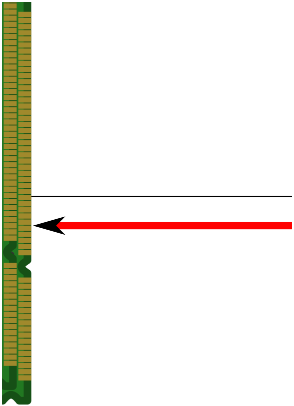
- we've learned data access is an important factor that determines performance of your program
- it is thus clearly important to analyze how "good" your algorithm is, in terms of data access performance
- we routinely analyze computational complexity of algorithms to predict or explain algorithm performance, but it ignores the differing costs of accessing memory hierarchy (all memory accesses are $O(1)$)
- we like to do *something analogous for data access*

Contents

Contents

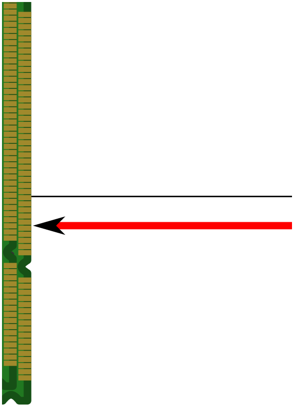
Analyzing data access performance of serial programs

- we like to analyze data access cost of serial programs (on pencils and papers)



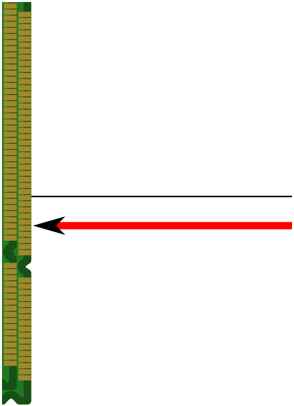
Analyzing data access performance of serial programs

- we like to analyze data access cost of serial programs (on pencils and papers)
- for this purpose, we calculate the amount of *data transferred between levels of memory hierarchy*



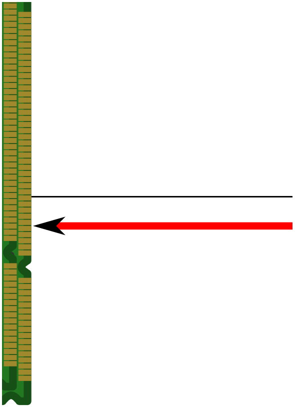
Analyzing data access performance of serial programs

- we like to analyze data access cost of serial programs (on pencils and papers)
- for this purpose, we calculate the amount of *data transferred between levels of memory hierarchy*
- in practical terms, this is a proxy of *cache misses*



Analyzing data access performance of serial programs

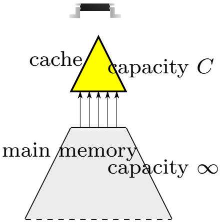
- we like to analyze data access cost of serial programs (on pencils and papers)
- for this purpose, we calculate the amount of *data transferred between levels of memory hierarchy*
- in practical terms, this is a proxy of *cache misses*
- the analysis assumes a simple two level memory hierarchy



Contents

Model (of a single processor machine)

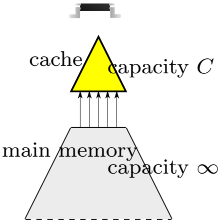
- a machine consists of
  - a processor,
  - a fixed size cache (C words), and
  - an arbitrarily large main memory
  - accessing a word not in the cache mandates transferring the word into the cache



For now, we assume a single processor machine.

Model (of a single processor machine)

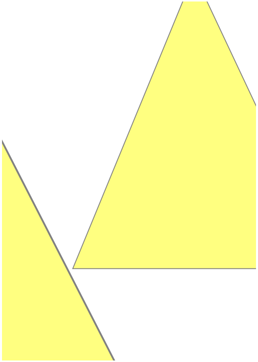
- a machine consists of
  - a processor,
  - a fixed size cache (C words), and
  - an arbitrarily large main memory
  - accessing a word not in the cache mandates transferring the word into the cache
- the cache holds the most recently accessed C words; $\approx$
  - line size = single word (whatever is convenient)
  - fully associative
  - LRU replacement



For now, we assume a single processor machine.

Gaps between our model and real machines

- hierarchical caches:

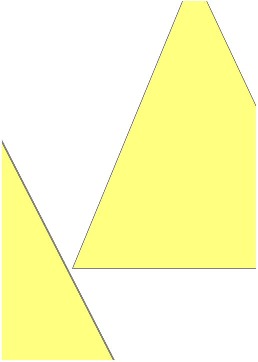


⇒ each level can be easily modeled separately, with caches of various sizes

Gaps between our model and real machines

- hierarchical caches:

  ⇒ each level can be easily modeled separately, with caches of various sizes

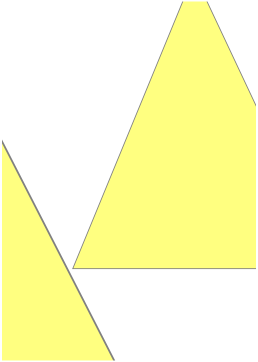


- concurrent accesses:
  - the model only counts the amount of data transferred
  - in practice the cost heavily depends on how many concurrent accesses you have
  - this model cannot capture the difference between link list traversal and random array access

Gaps between our model and real machines

- hierarchical caches:

  ⇒ each level can be easily modeled separately, with caches of various sizes



- concurrent accesses:
  - the model only counts the amount of data transferred
  - in practice the cost heavily depends on how many concurrent accesses you have
  - this model cannot capture the difference between link list traversal and random array access
- prefetch:

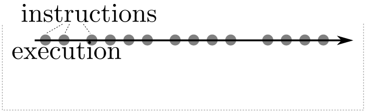
Contents

Terminologies

- an *execution* of a program is the sequence of instructions

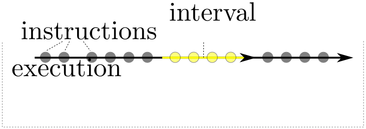
Terminologies

- an *execution* of a program is the sequence of instructions



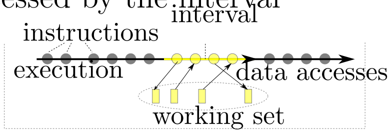
Terminologies

- an *execution* of a program is the sequence of instructions
- an *interval* in an execution is a consecutive sequence of instructions in the execution



Terminologies

- an *execution* of a program is the sequence of instructions
- an *interval* in an execution is a consecutive sequence of instructions in the execution
- the *working set* of an interval is the number of *distinct words* accessed by the interval

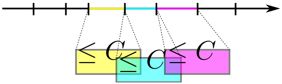


A basic methodology

to calculate the amount of data transferred in an execution,

1. split an execution into intervals each of which fits in the cache; i.e.,

$$\text{working set size of an interval } \leq C \text{ (cache size)} \qquad (*)$$



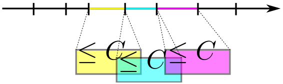
A basic methodology

to calculate the amount of data transferred in an execution,

1. split an execution into intervals each of which fits in the cache; i.e.,

$$\text{working set size of an interval } \leq C \text{ (cache size)} \qquad (*)$$

2. then, each interval transfers at most C words, because each word misses at most only once in the interval



A basic methodology

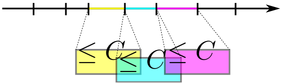
to calculate the amount of data transferred in an execution,

1. split an execution into intervals each of which fits in the cache; i.e.,

$$\text{working set size of an interval } \leq C \text{ (cache size)} \qquad (*)$$

2. then, each interval transfers at most C words, because each word misses at most only once in the interval
3. therefore,

$$\text{data transferred} \leq \sum_{I:\text{all intervals}} \text{working set size of } I$$



Remarks

- the condition $(*)$ is important to bound data transfers from above

$$\text{working set size of an interval } \leq C \text{ (cache size)} \qquad (*)$$

Remarks

- the condition $(*)$ is important to bound data transfers from above

$$\text{working set size of an interval } \leq C \text{ (cache size)} \qquad (*)$$

- each word in an interval misses at most only once, because
  - the cache is LRU, and
  - the condition $(*)$ guarantees that each word is never evicted within the interval

Remarks

- the condition $(*)$ is important to bound data transfers from above

$$\text{working set size of an interval } \leq C \text{ (cache size)} \qquad (*)$$

- each word in an interval misses at most only once, because
  - the cache is LRU, and
  - the condition $(*)$ guarantees that each word is never evicted within the interval
- in practical terms, an essential step to analyze data transfer is to *identify the largest intervals that fits in the cache*

Contents

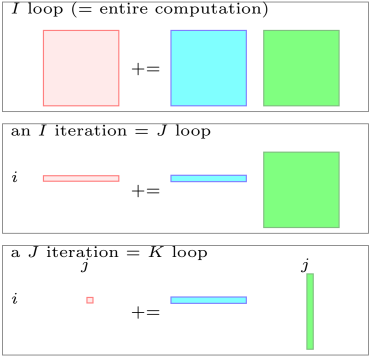
Applying the methodology

- we will apply the methodology to some of the algorithms we have studied so far
- the key is to *find subproblems (intervals) that fit in the cache*

Analyzing the triply nested loop

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    for (k = 0; k < n; k++) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

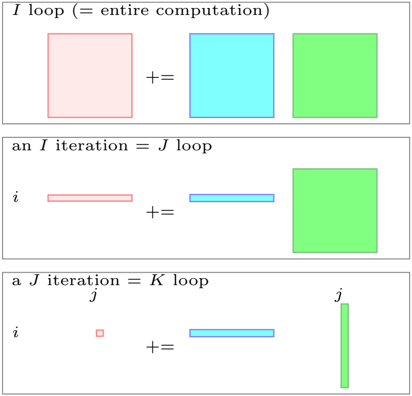
- key question: *which iteration fits the cache?*



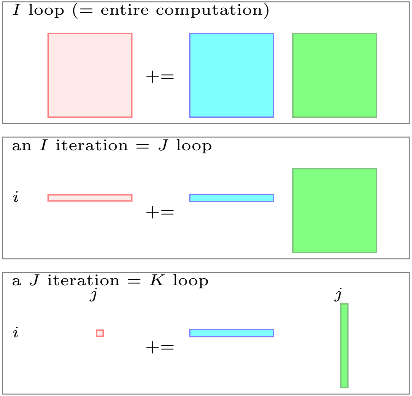
Working sets

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    for (k = 0; k < n; k++) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

level	working set size
I loop	$3n^2$
J loop	$2n + n^2$
K loop	$1 + 2n$
K iteration	3

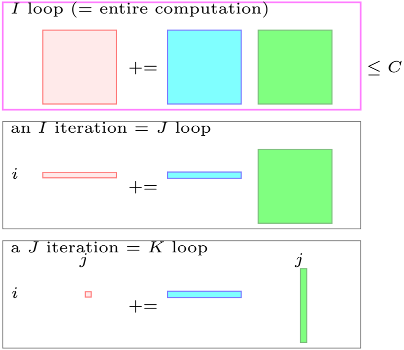


Cases to consider



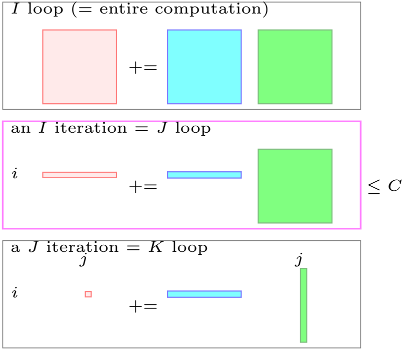
Cases to consider

- Case 1: the three matrices all fit in the cache ($3n^2 \leq C$)



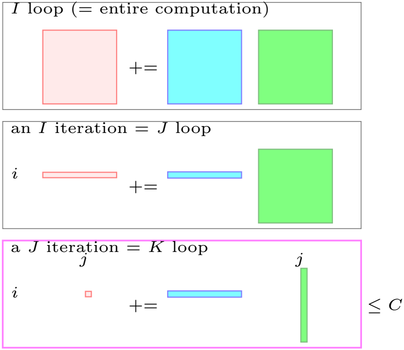
Cases to consider

- Case 1: the three matrices all fit in the cache ($3n^2 \leq C$)
- Case 2: a single i iteration ($\approx$ a matrix) fits in the cache ($2n + n^2 \leq C$)



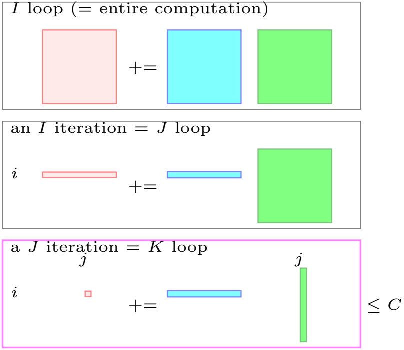
Cases to consider

- Case 1: the three matrices all fit in the cache ($3n^2 \leq C$)
- Case 2: a single i iteration ($\approx$ a matrix) fits in the cache ($2n + n^2 \leq C$)
- Case 3: a single j iteration ($\approx$ two vectors) fits in the cache ($1 + 2n \leq C$)



Cases to consider

- Case 1: the three matrices all fit in the cache ($3n^2 \leq C$)
- Case 2: a single i iteration ($\approx$ a matrix) fits in the cache ($2n + n^2 \leq C$)
- Case 3: a single j iteration ($\approx$ two vectors) fits in the cache ($1 + 2n \leq C$)
- Case 4: none of the above ($1 + 2n > C$)

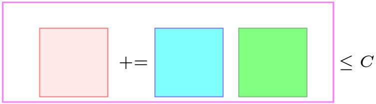


Case 1 ($3n^2 \leq C$)

- trivially, each element misses the cache only once. thus,

$$R(n) \leq 3n^2 = \frac{3}{n} \cdot n^3$$

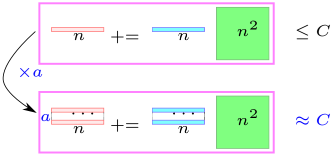
interpretation: each element of A, B, and C are reused n times



Case 2 $(2n + n^2 \leq C)$

- the maximum number of i-iterations that fit in the cache is:

$$a \approx \frac{C - n^2}{2n}$$



- each such set of iterations transfer $\leq C$ words, so

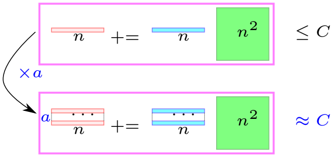
$$R(n) \leq \frac{n}{a} C = \frac{n}{a}(n^2 + 2an) = \left(\frac{1}{a} + \frac{2}{n}\right) n^3$$

interpretation: each element of B is reused a times in the cache; each element in A or C many $(\propto n)$ times

A Remark

- for a particular access pattern of the matrix multiplication, a better bound can be obtained
- as we know all elements of B are accessed in each i-iteration, they all stay in the cache
- so

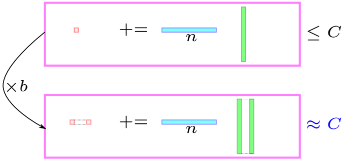
$$R(n) \leq 1 \cdot n^2 + \frac{n}{a}(2an) = \frac{3}{n} \cdot n^3$$



Case 3 $(1 + 2n \le C)$

- the maximum number of j-iterations that fit in the cache is:

$$b \approx \frac{C - n}{n + 1}$$



- each such set of iterations transfer $\le C$ words, so

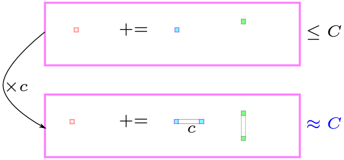
$$R(n) \le \frac{n^2}{b} C = \frac{n^2}{b}(n + b(n+1)) = \left(\frac{1}{b} + 1 + \frac{1}{n}\right) n^3$$

interpretation: each element in B is never reused; each element in A b times; each clement in C many $(\propto n)$ times

Case 4 $(1 + 2n > C)$

- the maximum number of k-iterations that fit in the cache is:

$$c \approx \frac{C - 1}{2}$$



- each such set of iterations transfer $\leq C$ words, so

$$R(n) \leq \frac{n^3}{c} C = \left(2 + \frac{1}{c}\right) n^3$$

interpretation: each element of B or A never reused; each element of C reused c times

Summary

- summarize $R(n)/n^3$, the number of misses per multiply-add $(0 \sim 3)$

condition	$R(n)/n^3$	range
$3n^2 \leq C$	$\frac{3}{n}$	~ 0
$2n + n^2 \leq C$	$\frac{1}{a} + \frac{2}{n}$	$0 \sim 1$
$1 + 2n \leq C$	$\frac{1}{b} + 1 + \frac{1}{n}$	$1 \sim 2$
$1 + 2n > C$	$2 + \frac{1}{c}$	$2 \sim 3$

So how to improve it?

- in general, the traffic increases when *the same amount of computation has a large working set*

So how to improve it?

- in general, the traffic increases when *the same amount of computation has a large working set*
- to reduce the traffic, you arrange the computation (order subcomputations) so that *you do a lot of computation on the same amount of data*

So how to improve it?

- in general, the traffic increases when *the same amount of computation has a large working set*
- to reduce the traffic, you arrange the computation (order subcomputations) so that *you do a lot of computation on the same amount of data*
- the notion is so important that it is variously called
  - compute/data ratio,
  - flops/byte,
  - compute intensity, or
  - arithmetic intensity

So how to improve it?

- in general, the traffic increases when *the same amount of computation has a large working set*
- to reduce the traffic, you arrange the computation *(*order subcomputations) so that *you do a lot of computation on the same amount of data*
- the notion is so important that it is variously called
  - compute/data ratio,
  - flops/byte,
  - compute intensity, or
  - arithmetic intensity
- the key is to identify the unit of computation (task) whose compute intensity is high (compute-intensive task)

The straightforward loop in light of compute intensity

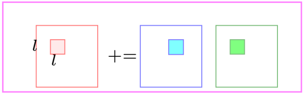
level	flops	working set size	ratio
I loop	$2n^3$	$3n^2$	$2/3n$
J loop	$2n^2$	$2n + n^2$	~ 2
K loop	$2n$	$1 + 2n$	~ 1
K iteration	2	3	$2/3$

- the outermost loop has an $O(n)$ compute intensity
- yet each iteration of which has only an $O(1)$ compute intensity

Cache blocking (tiling)

- for matrix multiplication, let l be the maximum number that satisfies $3l^2 \leq C$ (i.e., $l \approx \sqrt{C/3}$) and form a subcomputation that performs a $(l \times l)$ matrix multiplication
- ignoring remainder iterations, it looks like:

```
l = √(C/3);
for (ii = 0; ii < n; ii += l)
 for (jj = 0; jj < n; jj += l)
  for (kk = 0; kk < n; kk += l)
   /* working set fits in the cache below */
   for (i = ii; i < ii + l; i++)
    for (j = jj; j < jj + l; j++)
     for (k = kk; k < kk + l; k++)
       A(i,j) += B(i,k) * C(k,j);
```



Cache blocking (tiling)

- each subcomputation:
  - performs $2l^3$ flops and
  - touches $3l^2$ distinct words

```
l = √(C/3);
for (ii = 0; ii < n; ii += l)
 for (jj = 0; jj < n; jj += l)
  for (kk = 0; kk < n; kk += l)
   /* working set fits in the cache below */
   for (i = ii; i < ii + l; i++)
    for (j = jj; j < jj + l; j++)
     for (k = kk; k < kk + l; k++)
      A(i,j) += B(i,k) * C(k,j);
```

Cache blocking (tiling)

- each subcomputation:
  - performs $2l^3$ flops and
  - touches $3l^2$ distinct words
- it thus has the compute intensity:

$$\frac{2l^3}{3l^2} = \frac{2}{3}l \approx \frac{2}{3}\sqrt{\frac{C}{3}}$$

```
l = √(C/3);
for (ii = 0; ii < n; ii += l)
 for (jj = 0; jj < n; jj += l)
  for (kk = 0; kk < n; kk += l)
   /* working set fits in the cache below */
   for (i = ii; i < ii + l; i++)
    for (j = jj; j < jj + l; j++)
     for (k = kk; k < kk + l; k++)
      A(i,j) += B(i,k) * C(k,j);
```

Cache blocking (tiling)

- each subcomputation:
  - performs $2l^3$ flops and
  - touches $3l^2$ distinct words
- it thus has the compute intensity:

$$\frac{2l^3}{3l^2} = \frac{2}{3}l \approx \frac{2}{3}\sqrt{\frac{C}{3}}$$

- or, the traffic is

$$C \cdot \left(\frac{n}{l}\right)^3 = \frac{3\sqrt{3}}{\sqrt{C}}n^3$$

```
l = √(C/3);
for (ii = 0; ii < n; ii += l)
 for (jj = 0; jj < n; jj += l)
  for (kk = 0; kk < n; kk += l)
   /* working set fits in the cache below */
   for (i = ii; i < ii + l; i++)
    for (j = jj; j < jj + l; j++)
     for (k = kk; k < kk + l; k++)
      A(i,j) += B(i,k) * C(k,j);
```

Effect of cache blocking

condition	$R(n)/n^3$	range
$3n^2 \leq C$	$\dfrac{3}{n}$	~ 0
$2n + n^2 \leq C$	$\dfrac{1}{a} + \dfrac{2}{n}$	$0 \sim 1$
$1 + 2n \leq C$	$\dfrac{1}{b} + 1 + \dfrac{1}{n}$	$1 \sim 2$
$1 + 2n > C$	$2 + \dfrac{1}{c}$	$2 \sim 3$
blocking	$\dfrac{3\sqrt{3}}{\sqrt{C}}$	below

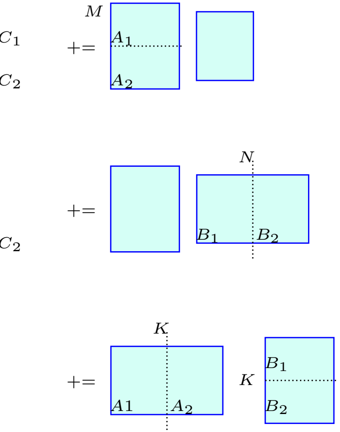
- assume a word = 4 bytes (`float`)

bytes	C	l	$R(n)/n^3$
32K	8K	52	0.72
256K	64K	147	0.43
3MB	768K	886	0.0059

Recursive blocking

- the tiling technique just mentioned targets a cache of a particular size (= level)
- need to do this at all levels (12 deep nested loop)?
- we also (for the sake of simplicity) assumed all matrices are square
- for generality, portability, simplicity, recursive blocking may apply

Recursively blocked matrix multiply




```
gemm(A, B, C) {
  if ((M, N, K) = (1, 1, 1)) {
    c11 += a11 * b11;
  } else if (max(M, N, K) = M) {
    gemm(A1, B, C1);
    gemm(A2, B, C2);
  } else if (max(M, N, K) = N) {
    gemm(A, B1, C1);
    gemm(A, B1, C2);
  } else { /* max(M, N, K) = K */
    gemm(A1, B1, C);
    gemm(A2, B2, C);
  }
}
```

- it divides flops into two
- it divides two of the three matrices, along *the longest* axis

Settings

- a single word = a single floating point number
- cache size $= C$ words

Settings

- a single word = a single floating point number
- cache size $= C$ words
- let $R(M, N, K)$ be the number of words transferred between cache and memory when multiplying $M \times K$ and $K \times N$ matrices (the cache is initially empty)

Settings

- a single word = a single floating point number
- cache size = C words
- let $R(M, N, K)$ be the number of words transferred between cache and memory when multiplying $M \times K$ and $K \times N$ matrices (the cache is initially empty)
- let $R(w)$ be the maximum number of words transferred for any matrix multiply of up to w words in total:

$$R(w) \equiv \max_{MK+KN+MN \leq w} R(M, N, K)$$

we want to bound $R(w)$ from the above

Settings

- a single word = a single floating point number
- cache size $= C$ words
- let $R(M, N, K)$ be the number of words transferred between cache and memory when multiplying $M \times K$ and $K \times N$ matrices (the cache is initially empty)
- let $R(w)$ be the maximum number of words transferred for any matrix multiply of up to w words in total:

$$R(w) \equiv \max_{MK+KN+MN \leq w} R(M, N, K)$$

  we want to bound $R(w)$ from the above
- to avoid making analysis tedious, assume all matrices are "nearly square"

$$\max(M, N, K) \leq 2 \min(M, N, K)$$

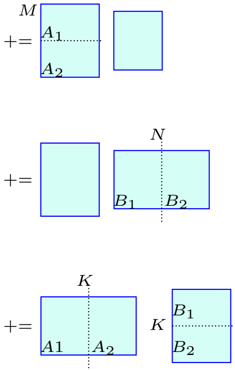
The largest subproblem that fits in the cache

- the working set of `gemm(A,B,C)` is $(MK + KN + MN)$ (words)
- it fits in the cache if this is $\leq C$
- thus we have:

$$\therefore R(w) \leq C \quad (w \leq C)$$

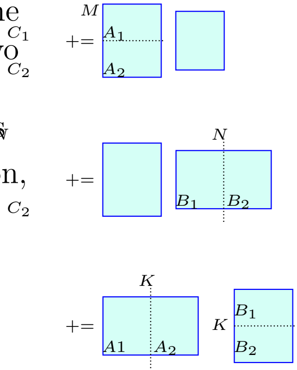
Analyzing cases that do not fit in the cache

- when $MK + KN + MN > C$, the interval doing `gemm(A,B,C)` is two subintervals, each of which does `gemm` for slightly smaller matrices



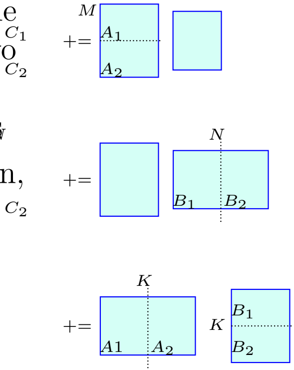
Analyzing cases that do not fit in the cache

- when $MK + KN + MN > C$, the interval doing `gemm(A,B,C)` is two subintervals, each of which does `gemm` for slightly smaller matrices
- in the "nearly square" assumption, the working set becomes $\leq 1/4$ when we divide 3 times



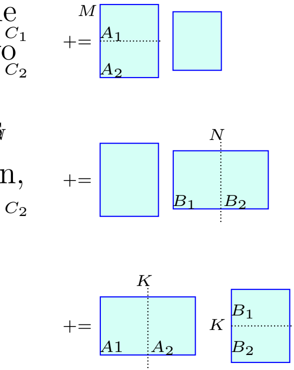
Analyzing cases that do not fit in the cache

- when $MK + KN + MN > C$, the interval doing `gemm(A,B,C)` is two subintervals, each of which does `gemm` for slightly smaller matrices
- in the "nearly square" assumption, the working set becomes $\leq 1/4$ when we divide 3 times
- to make math simpler, we take it that the working set becomes $\leq \dfrac{1}{\sqrt[3]{4}} (= 2^{-2/3})$ of the original size on each recursion. i.e.,



Analyzing cases that do not fit in the cache

- when $MK + KN + MN > C$, the interval doing `gemm(A,B,C)` is two subintervals, each of which does `gemm` for slightly smaller matrices
- in the "nearly square" assumption, the working set becomes $\leq 1/4$ when we divide 3 times
- to make math simpler, we take it that the working set becomes $\leq \dfrac{1}{\sqrt[3]{4}} (= 2^{-2/3})$ of the original size on each recursion. i.e.,



$$\therefore R(w) \leq 2R(w/\sqrt[3]{4}) \quad (w > C)$$

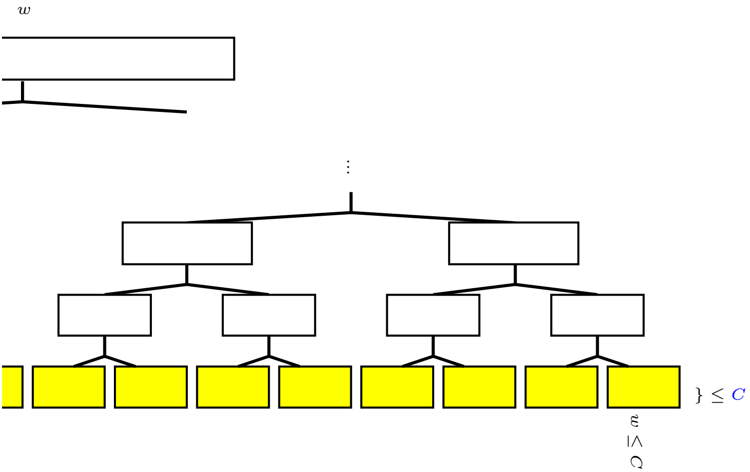
Combined

- we have:

$$R(w) \leq \begin{cases} w & (w \leq C) \\ 2R(w/\sqrt[3]{4}) & (w > C) \end{cases}$$

- when $w > C$, it takes up to $d \approx \log_{\sqrt[3]{4}}(w/C)$ recursion steps until the working set becomes $\leq C$
- the whole computation is essentially 2^d consecutive intervals, each transferring $\leq C$ words

Illustration



. . .

Result

- we have:

$$R(w) \leq \frac{1}{\sqrt{C}} w^{3/2}$$

- for square ($n \times n$) matrices ($w = 3n^2$),

$$\therefore R(n) = R(3n^2) = \frac{3\sqrt{3}}{\sqrt{C}} n^3$$

- the same as the blocking we have seen before (not surprising), but we achieved this for all cache levels

A practical remark

- in practice we stop recursion when the matrices become “small enough”

```
gemm(A, B, C) {
  if (A, B, C together fit in the cache) {
    for (i, j, k) ∈ [0..M] × [0..N] × [0..K]
      c_ij += a_ik * b_kj;
  } else if (max(M, N, K) = M) {
    gemm(A_1, B, C_1);
    gemm(A_2, B, C_2);
  } else if (max(M, N, K) = N) {
    gemm(A, B_1, C_1);
    gemm(A, B_1, C_2);
  } else { /* max(M, N, K) = K */
    gemm(A_1, B_1, C);
    gemm(A_2, B_2, C);
  }
}
```

A practical remark

- in practice we stop recursion when the matrices become "small enough"
- but how small is small enough?

```
gemm(A, B, C) {
  if (A, B, C together fit in the cache) {
    for (i, j, k) ∈ [0..M] × [0..N] × [0..K]
      c_ij += a_ik * b_kj;
  } else if (max(M, N, K) = M) {
    gemm(A_1, B, C_1);
    gemm(A_2, B, C_2);
  } else if (max(M, N, K) = N) {
    gemm(A, B_1, C_1);
    gemm(A, B_1, C_2);
  } else { /* max(M, N, K) = K */
    gemm(A_1, B_1, C);
    gemm(A_2, B_2, C);
  }
}
```

A practical remark

- in practice we stop recursion when the matrices become "small enough"
- but how small is small enough?
- when the threshold ≤ level x cache, the analysis holds for all levels x and lower

```
gemm(A, B, C) {
  if (A, B, C together fit in the cache) {
    for (i, j, k) ∈ [0..M] × [0..N] × [0..K]
      c_ij += a_ik * b_kj;
  } else if (max(M, N, K) = M) {
    gemm(A_1, B, C_1);
    gemm(A_2, B, C_2);
  } else if (max(M, N, K) = N) {
    gemm(A, B_1, C_1);
    gemm(A, B_1, C_2);
  } else { /* max(M, N, K) = K */
    gemm(A_1, B_1, C);
    gemm(A_2, B_2, C);
  }
}
```

A practical remark

- in practice we stop recursion when the matrices become "small enough"
- but how small is small enough?
- when the threshold $\leq$ level x cache, the analysis holds for all levels x and lower

```
gemm(A, B, C) {
  if (A, B, C together fit in the cache) {
    for (i, j, k) ∈ [0..M] × [0..N] × [0..K]
      c_ij += a_ik * b_kj;
  } else if (max(M, N, K) = M) {
    gemm(A_1, B, C_1);
    gemm(A_2, B, C_2);
  } else if (max(M, N, K) = N) {
    gemm(A, B_1, C_1);
    gemm(A, B_1, C_2);
  } else { /* max(M, N, K) = K */
    gemm(A_1, B_1, C);
    gemm(A_2, B_2, C);
  }
}
```

- on the other hand, we like to make it large, to reduce control overhead

Contents

Tools to measure cache/memory traffic

- analyzing data access performance is harder than analyzing computational efficiency (ignoring caches)
  - the code reflects how much computation you do
  - you can experimentally confirm your understanding by counting cycles (or wall-clock time)
- caches are complex and subtle
  - the same data access expression (e.g., `a[i]`) may or may not count as the traffic
  - gaps are larger between our model and the real machines (associativity, prefetches, local variables and stacks we often ignore, etc.)
- we like to have a tool to measure what happened on the machine $\rightarrow$ performance counters

Performance counters

- recent CPUs equip with *performance counters*, which count the number of times various events happen in the processor
- OS exposes it to users (e.g., Linux `perf_event_open` system call)
- there are tools to access them more conveniently
  - command: Linux perf (`man perf`)
  - library: PAPI `http://icl.cs.utk.edu/papi/`
  - GUI: hpctoolkit `http://hpctoolkit.org/`, VTunes, ...

Contents

`perf` command

- `perf` command is particularly easy to use

```
perf stat command line
```

will show you cycles, instructions, and some other info

- to access performance counters of your interest (e.g., cache misses), specify them with `-e`

```
perf stat -e counter -e counter ...  command line
```

- to know the list of available counters

```
perf list
```

perf command

Table 19-1. Architectural Performance Events

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
3CH	UnHalted Core Cycles	00H	Unhalted core cycles	
3CH	UnHalted Reference Cycles	01H	Unhalted reference cycles	Measures bus cycle[1]
C0H	Instruction Retired	00H	Instruction retired	

- many interesting counters are not listed by `perf list`
- we often need to resort to "raw" events (defined on each CPU model)
- consult intel document [1]
- if the table says Event Num = `2EH`, Umask Value = `41H`, then you can access it via perf by `-e r412e` (umask; event num)

[1] Intel 64 and IA-32 Architectures Developer's Manual: Volume 3B: System Programming Guide, Part 2. Chapter 19 "Performance Monitoring Events" `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html`

Contents

PAPI library

- library for accessing performance counters
- `http://icl.cs.utk.edu/papi/index.html`
- basic concepts
    - create an empty "event set"
    - add events of interest to the event set
    - start counting
    - do whatever you want to measure
    - stop counting
- visit `http://icl.cs.utk.edu/papi/docs/index.html` and see "Low Level Functions"

PAPI minimum example (single thread)

- A minimum example with a single thread and no error checks

```
#include <papi.h>
int main() {
  { do_whatever(); }
  return 0;
}
```

PAPI minimum example (single thread)

- A minimum example with a single thread and no error checks

```
#include <papi.h>
int main() {
  PAPI_library_init(PAPI_VER_CURRENT);
  int es = PAPI_NULL;
  PAPI_create_eventset(&es);
  PAPI_add_named_event(es, "ix86arch::LLC_MISSES");
  PAPI_start(es);
  long long values[1];
  { do_whatever(); }
  PAPI_stop(es, values);
  printf("%lld\n", values[0]);
  return 0;
}
```

Compiling and running PAPI programs

- compile and run

```
$ gcc ex.c -lpapi
$ ./a.out
33
```

- `papi_avail` and `papi_native_avail` list available event names (to pass to `PAPI_add_named_event`)
  - `perf_raw::r`*nnnn* for raw counters (same as perf command)

Error checks

- be prepared to handle errors (never assume you know what works)!
- many routines return PAPI_OK on success and a return code on error, which can then be passed to PAPI_strerror(*return_code*) to convert it into an error message
- encapsulate such function calls with this

```
void check_(int ret, const char * fun) {
  if (ret != PAPI_OK) {
    fprintf(stderr, "%s failed (%s)\n", fun, PAPI_strerror(ret));
    exit(1);
  }
}

#define check(call) check_(call, #call)
```

A complete example with error checks

```
#include <stdio.h>
#include <stdlib.h>
#include <papi.h>

void check_(int ret, const char * fun) {
  if (ret != PAPI_OK) {
    fprintf(stderr, "%s failed (%s)\n", fun, PAPI_strerror(ret));
    exit(1);
  }
}
#define check(f) check_(f, #f)

int main() {
  int ver = PAPI_library_init(PAPI_VER_CURRENT);
  if (ver != PAPI_VER_CURRENT) {
    fprintf(stderr, "PAPI_library_init(%d) failed (returned %d)\n",
            PAPI_VER_CURRENT, ver);
    exit(1);
  }
  int es = PAPI_NULL;
  check(PAPI_create_eventset(&es));
  check(PAPI_add_named_event(es, "ix86arch::LLC_MISSES"));
  check(PAPI_start(es));
  { do_whatever(); }
  long long values[1];
  check(PAPI_stop(es, values));
  printf("%lld\n", values[0]);
  return 0;
}
```

Multithreaded programs

- must call `PAPI_thread_init(`*id_fun*`)` in addition to `PAPI_library_init(PAPI_VER_CURRENT)`
  - *id_fun* is a function that returns identity of a thread (e.g., `pthread_self`, `omp_get_thread_num`)
- each thread must call `PAPI_register_thread`
- event set is private to a thread (each thread must call `PAPI_create_eventset()`, `PAPI_start()`, `PAPI_stop()`)

Multithreaded example

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <papi.h>
/* check_ and check omitted (same as single thread) */
int main() {
  /* error check for PAPI_library_init omitted (same as single thread) */
  PAPI_library_init(PAPI_VER_CURRENT);
  check(PAPI_thread_init((unsigned long(*)()) omp_get_thread_num));
#pragma omp parallel
  {
    check(PAPI_register_thread()); /* each thread must do this */
    int es = PAPI_NULL;
    check(PAPI_create_eventset(&es)); /* each thread must create its own set */
    check(PAPI_add_named_event(es, "ix86arch::LLC_MISSES"));
    check(PAPI_start(es));
    { do_whatever(); }
    long long values[1];
    check(PAPI_stop(es, values));
    printf("thread %d: %lld\n", omp_get_thread_num(), values[0]);
  }
  return 0;
}
```

Several ways to obtain counter values

- `PAPI_stop(`*es*`, `*values*`)`: get current values and stop counting
- `PAPI_read(`*es*`, `*values*`)`: get current values and continue counting
- `PAPI_accum(`*es*`, `*values*`)`: accumulate current values, reset counters, and continue counting

Useful PAPI commands

- papi_avail, papi_native_avail: list event counter names
- papi_mem_info: report information about caches and TLB (size, line size, associativity, etc.)

Contents

Matching the model and measurements (measurements)

- warnings:
  - counters are highly CPU model-specific
  - do not expect portability too much; always check `perf list`, `perf_native_avail`, and the Intel manual
  - some counters or combination thereof cannot be monitored even if listed in `perf_native_avail` (you fail to add it to event set; never forget to check return code)
  - virtualized environments have none or limited support of performance counters; *Amazon EC2 environment shows no counters available* `(; ;)` (I don't know if there is a workaround)
- the following experiments were conducted on my Haswell (Core i7-4500U) laptop
  - L1 : 32KB, L2 : 256KB, L3 : 4MB

Matching the model and measurements (measurements)

- relevant counters
  - `L1D:REPLACEMENT`
  - `L2_TRANS:L2_FILL`
  - `MEM_LOAD_UOPS_RETIRED:L3_MISS`
- cache miss counts do not include line transfers hit thanks to prefetches
- `L1D:REPLACEMENT` and `L2_TRANS:L2_FILL` seem closer to what we want to match our model against
- I could not find good counters for L3 caches, so measure `ix86arch::LLC_MISSES`

Matching the model and measurements

- counters give the number of *cache lines* transferred
- a line is 64 bytes and a word is 4 bytes, so we assume: *words* transferred $\approx 16\times$ cache lines transferred
- recall we have:

$$R(w) \leq \frac{1}{\sqrt{C}} w^{3/2}$$

  and $R(w)$ is the number of *words* transferred
- so we calculate:

$$\frac{\text{the number of cache lines transferred}}{w^{3/2}}$$

  (and expect it to be close to $\dfrac{1}{\sqrt{C}}$ for $w > C$)

$1/\sqrt{C}$

level	C	$\frac{1}{\sqrt{C}}$
L1	8K	$0.011048\cdots$
L2	64K	$0.003906\cdots$
L3	1M	$0.000976\cdots$

L1 (recursive blocking)

out/tex/data/mm/mm_blocking_l1rep_10000

- they are not constant as we expected

What are the spikes?

- it spikes when $M =$ a large power of two ($128|M$, to be more specific)
- analyzing why it's happening is a good exercise for you
- whatever it is, I told you avoid it!
- let's remove M's that are multiple of 128

L1 (remove multiples of 128)

out/tex/data/mm/mm_blocking_l1rep_128

M	value
1808	0.0187
1856	0.0178
1872	0.0177
1936	0.0170
1984	0.0159
2000	0.0167

$$\frac{1}{\sqrt{C}} \approx 0.011048\cdots$$

L1 (compare w/ and w/o recursive blocking)

out/tex/data/mm/mm_compare_l1rep_128

L1 (remove multiples of 64)

out/tex/data/mm/mm_compare_l1rep_64

L1 (remove multiples of 32)

out/tex/data/mm/mm_compare_l1rep_32

L2

out/tex/data/mm/mm_blocking_l2fill_10000

L2 (remove multiples of 128)

out/tex/data/mm/mm_blocking_l2fill_128

M	value
1808	0.00499
1856	0.00481
1872	0.00511
1936	0.00470
1984	0.00476
2000	0.00505

$$\frac{1}{\sqrt{C}} \approx 0.003906 \cdots$$

L2 (compare w/ and w/o recursive blocking)

out/tex/data/mm/mm_compare_l2fill_128

L2 (no multiples of 64)

out/tex/data/mm/mm_compare_l2fill_64

L2 (no multiples of 32)

out/tex/data/mm/mm_compare_l2fill_32

Contents

Review: (serial) merge sort

```
/* sort a..a_end and put the result into
   (i)  a (if dest = 0)
   (ii) t (if dest = 1) */
void ms(elem * a, elem * a_end,
        elem * t, int dest) {
  long n = a_end - a;
  if (n == 1) {
    if (dest) t[0] = a[0];
  } else {
    /* split the array into two */
    long nh = n / 2;
    elem * c = a + nh;
    /* sort 1st half */
    ms(a, c,     t,      1 - dest);
    /* sort 2nd half */
    ms(c, a_end, t + nh, 1 - dest);
    elem * s = (dest ? a : t);
    elem * d = (dest ? t : a);
    /* merge them */
    merge(s,      s + nh,
          s + nh, s + n, d);
  }
}
```

```
/* merge a_beg ... a_end
     and b_beg ... b_end
    into c */
void
merge(elem * a, elem * a_end,
      elem * b, elem * b_end,
      elem * c) {
  elem * p = a, * q = b, * r = c;
  while (p < a_end && q < b_end) {
    if (*p < *q) { *r++ = *p++; }
    else { *r++ = *q++; }
  }
  while (p < a_end) *r++ = *p++;
  while (q < b_end) *r++ = *q++;
}
```

note: as always, actually switch to serial sort below a threshold (not shown in the code above)

Memory $\leftrightarrow$ cache transfer in merge sort (1) base case

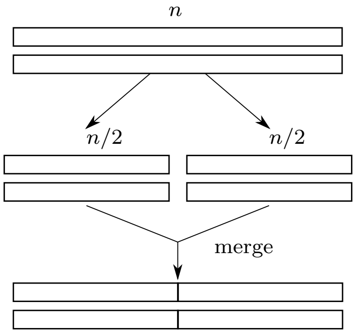
- merge sorting n elements takes *two* arrays of n elements each, and touch all elements of them $\Rightarrow$ *the working set is $2n$ words*
- thus, it fits in the cache when $2n \leq C$

$$\therefore R(n) \leq 2n \quad (2n \leq C)$$

Memory $\leftrightarrow$ cache transfer in merge sort (2)

- when $n > C/2$, the whole computation is two recursive calls + merging two results

```
long nh = n / 2;
/* sort 1st half */
ms(a, c,      t,       1 - dest);
/* sort 2nd half */
ms(c, a_end, t + nh, 1 - dest);
   ...
/* merge them */
merge(s,      s + nh,
      s + nh, s + n, d);
```

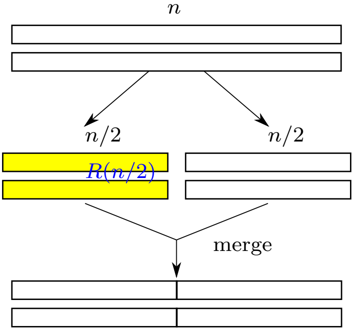


$$\therefore R(n) \leq 2R(n/2) + 2n \quad (n > C/2)$$

Memory ↔ cache transfer in merge sort (2)

- when $n > C/2$, the whole computation is two recursive calls + merging two results

```
long nh = n / 2;
/* sort 1st half */
ms(a, c,     t,      1 - dest);
/* sort 2nd half */
ms(c, a_end, t + nh, 1 - dest);
   ...
/* merge them */
merge(s,      s + nh,
      s + nh, s + n, d);
```

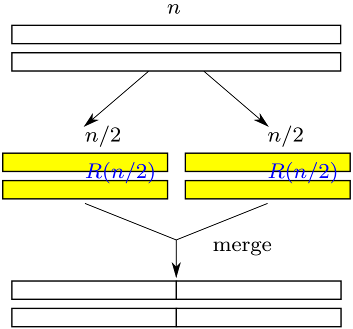


$$\therefore R(n) \leq 2R(n/2) + 2n \quad (n > C/2)$$

Memory ↔ cache transfer in merge sort (2)

- when $n > C/2$, the whole computation is two recursive calls + merging two results

```
long nh = n / 2;
/* sort 1st half */
ms(a, c,     t,       1 - dest);
/* sort 2nd half */
ms(c, a_end, t + nh, 1 - dest);
   ...
/* merge them */
merge(s,      s + nh,
      s + nh, s + n, d);
```

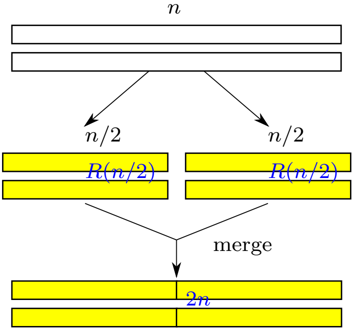


$$\therefore R(n) \leq 2R(n/2) + 2n \quad (n > C/2)$$

Memory $\leftrightarrow$ cache transfer in merge sort (2)

- when $n > C/2$, the whole computation is two recursive calls + merging two results

```
long nh = n / 2;
/* sort 1st half */
ms(a, c,      t,      1 - dest);
/* sort 2nd half */
ms(c, a_end, t + nh, 1 - dest);
   ...
/* merge them */
merge(s,      s + nh,
      s + nh, s + n, d);
```



$$\therefore R(n) \leq 2R(n/2) + 2n \quad (n > C/2)$$

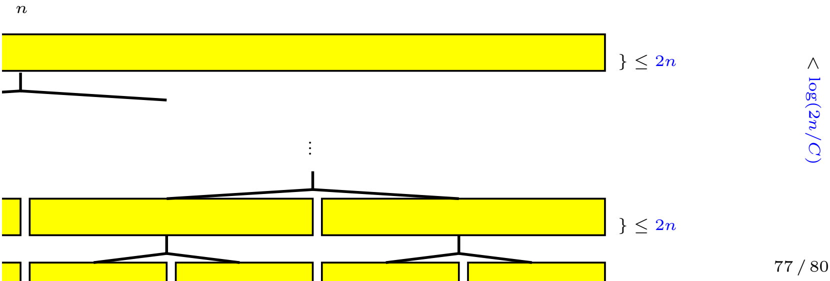
Combined

- so far we have:

$$R(n) \le \begin{cases} 2n & (n \le C/2) \\ 2R(n/2) + 2n & (n > C/2) \end{cases}$$

- for $n > C/2$, it takes at most $d \approx \log \frac{2n}{C}$ divide steps until it becomes $\le C/2$
- thus,

$$R(n) \le 2n \cdot d = 2n \log \frac{2n}{C}$$



Improving merge sort

- so what can we do to improve this?

$$R(n) \leq 2n \log \frac{2n}{C}$$

Improving merge sort

- so what can we do to improve this?

$$R(n) \leq 2n \log \frac{2n}{C}$$

- there are not much we can do to improve a single merge ($\because$ *each element of arrays is accessed only once*)

Improving merge sort

- so what can we do to improve this?

$$R(n) \le 2n \log \frac{2n}{C}$$

- there are not much we can do to improve a single merge ($\because$ *each element of arrays is accessed only once*)
- so the hope is to reduce the number of steps $\left(\log \frac{2n}{C}\right) \Rightarrow$ *multi-way merge*

Summary

- understanding and assessing data access performance (e.g., cache misses) is important

Summary

- understanding and assessing data access performance (e.g., cache misses) is important
- I hope I have taught that it's a subject of a rigid analysis, *not a black art*

Summary

- understanding and assessing data access performance (e.g., cache misses) is important
- I hope I have taught that it's a subject of a rigid analysis, *not a black art*
- the key for the assessment/analysis is *to identify a unit of computation that fits in the cache*, not to microscopically follow the state of the cache

Summary

- understanding and assessing data access performance (e.g., cache misses) is important
- I hope I have taught that it's a subject of a rigid analysis, *not a black art*
- the key for the assessment/analysis is *to identify a unit of computation that fits in the cache*, not to microscopically follow the state of the cache
- the key to achieve good cache performance is to keep *the compute intensity of cache-fitting computation* high

Next step

- our next goal is to understand data access performance of *parallel* algorithms
- we are particularly interested in performance of dynamically scheduled task parallel algorithms
- to this end, we first describe schedulers of task parallel systems