Parallel & Distributed Computing: Lecture 9

Alberto Paoluzzi

November 6, 2018

Parallel Algorithm Design

Preliminaries

Source: Introduction to Parallel Computing, Second Edition, By Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Addison Wesley, 2004

Specifying a nontrivial parallel algorithm may include:

- Identifying portions of the work that can be performed concurrently.
- Mapping the concurrent pieces of work onto multiple processes running in parallel.
- Distributing the input, output, and the code associated with the program.
- Managing accesses to data shared by multiple processors.
- Synchronizing the processors at various stages of the parallel program execution.

Decomposition, Tasks, and Dependency Graphs

- Tasks are programmer-defined units of computation into which the main computation is subdivided by means of decomposition
- Simultaneous execution of multiple tasks is the key to reducing the time required to solve the entire problem
- Tasks can be of arbitrary size, but once defined, they are regarded as indivisible units of computation

Dependency Graphs

Dependency Graphs abstract the mapping of tasks to processors, and specify where to put syncronization points (barriers)

Database query processing

Vehicle database example

ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	US$18,00
3476	Corolla	1999	White	IL	US$15,00
7623	Camry	2001	Green	NY	US$21,00
9834	Prius	2001	Green	CA	US$18,00
6734	Civic	2001	White	OR	US$17,00
5342	Altima	2001	Green	FL	US$19,00

Database query processing

Vehicle database example

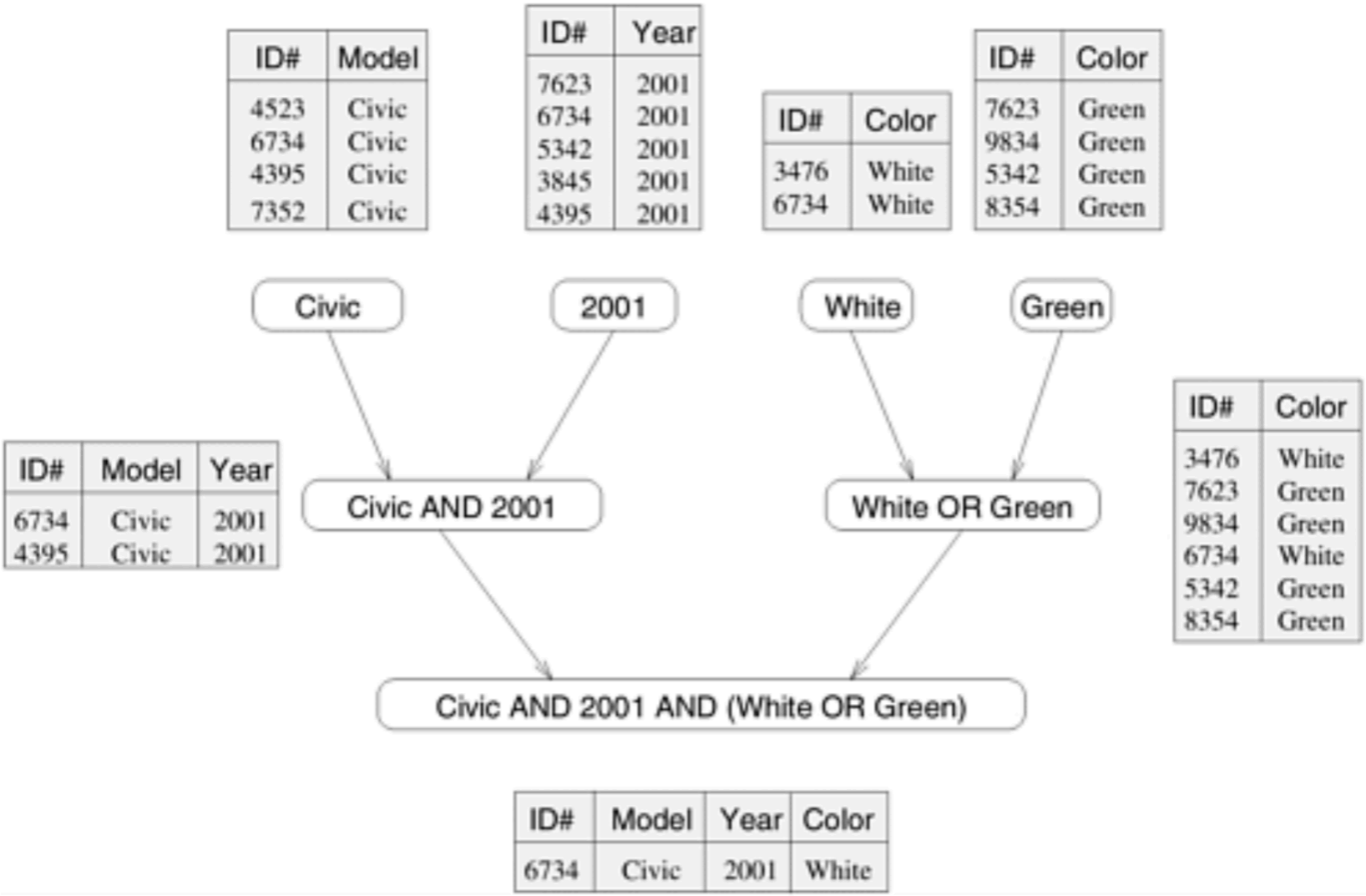
ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	US$18,00
3476	Corolla	1999	White	IL	US$15,00
7623	Camry	2001	Green	NY	US$21,00
9834	Prius	2001	Green	CA	US$18,00
6734	Civic	2001	White	OR	US$17,00
5342	Altima	2001	Green	FL	US$19,00

SQL query example:

MODEL=“Civic” AND YEAR=“2001” AND (COLOR=“Green” OR COLOR=“White”)

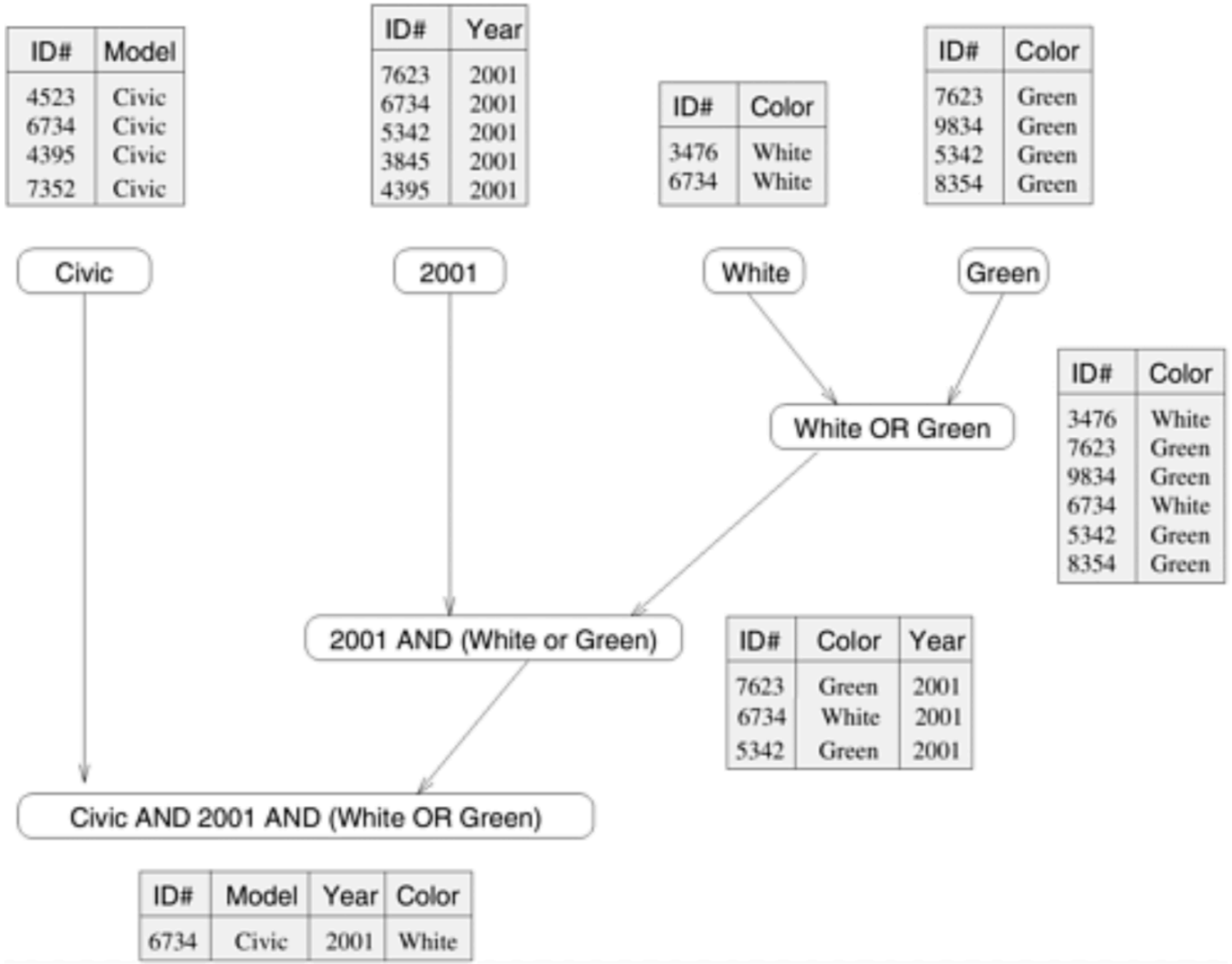
Database query processing

Data-dependency graph for the query processing operation

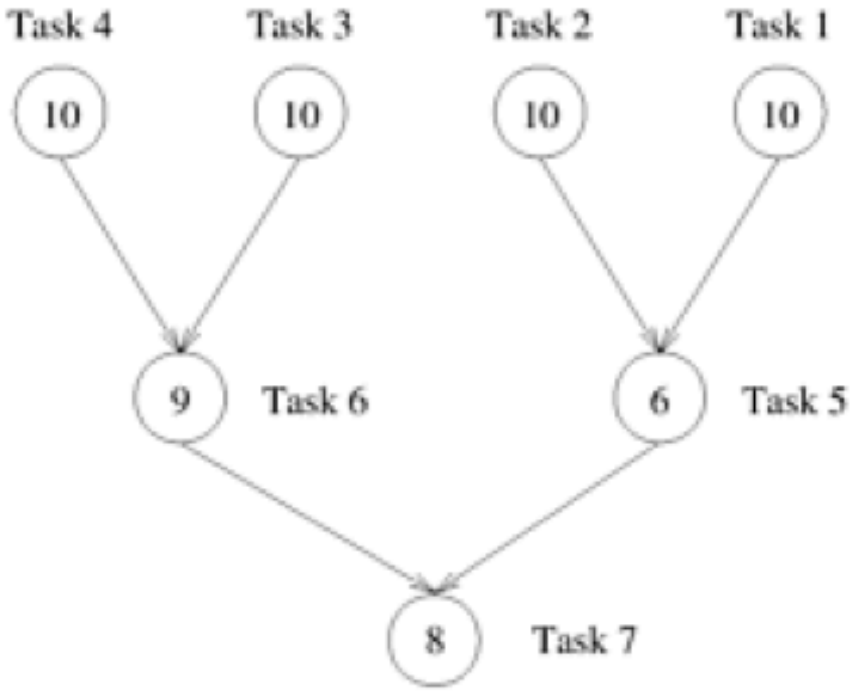


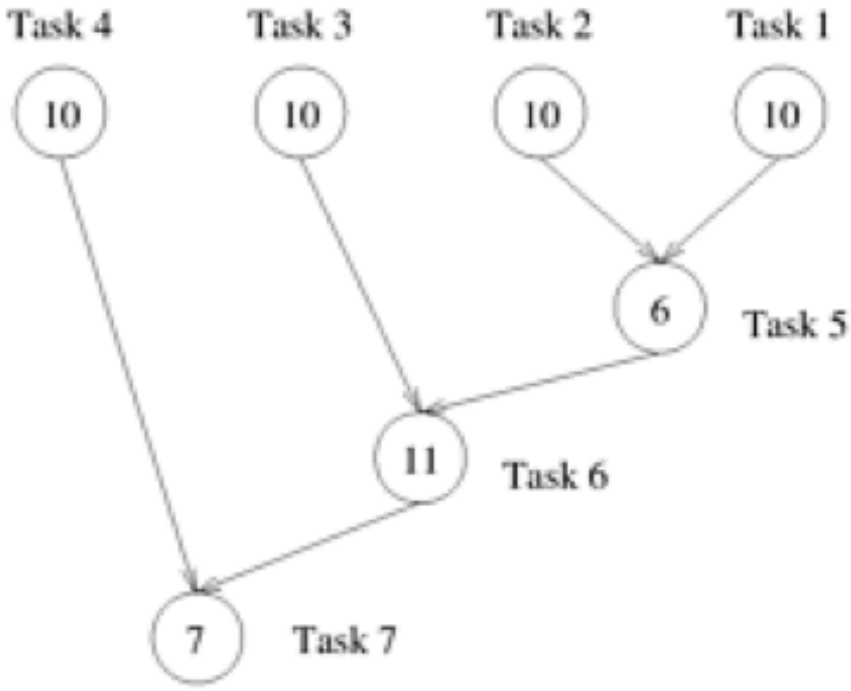
Database query processing

Another data-dependency graph for the query processing operation



Abstractions of the task graphs

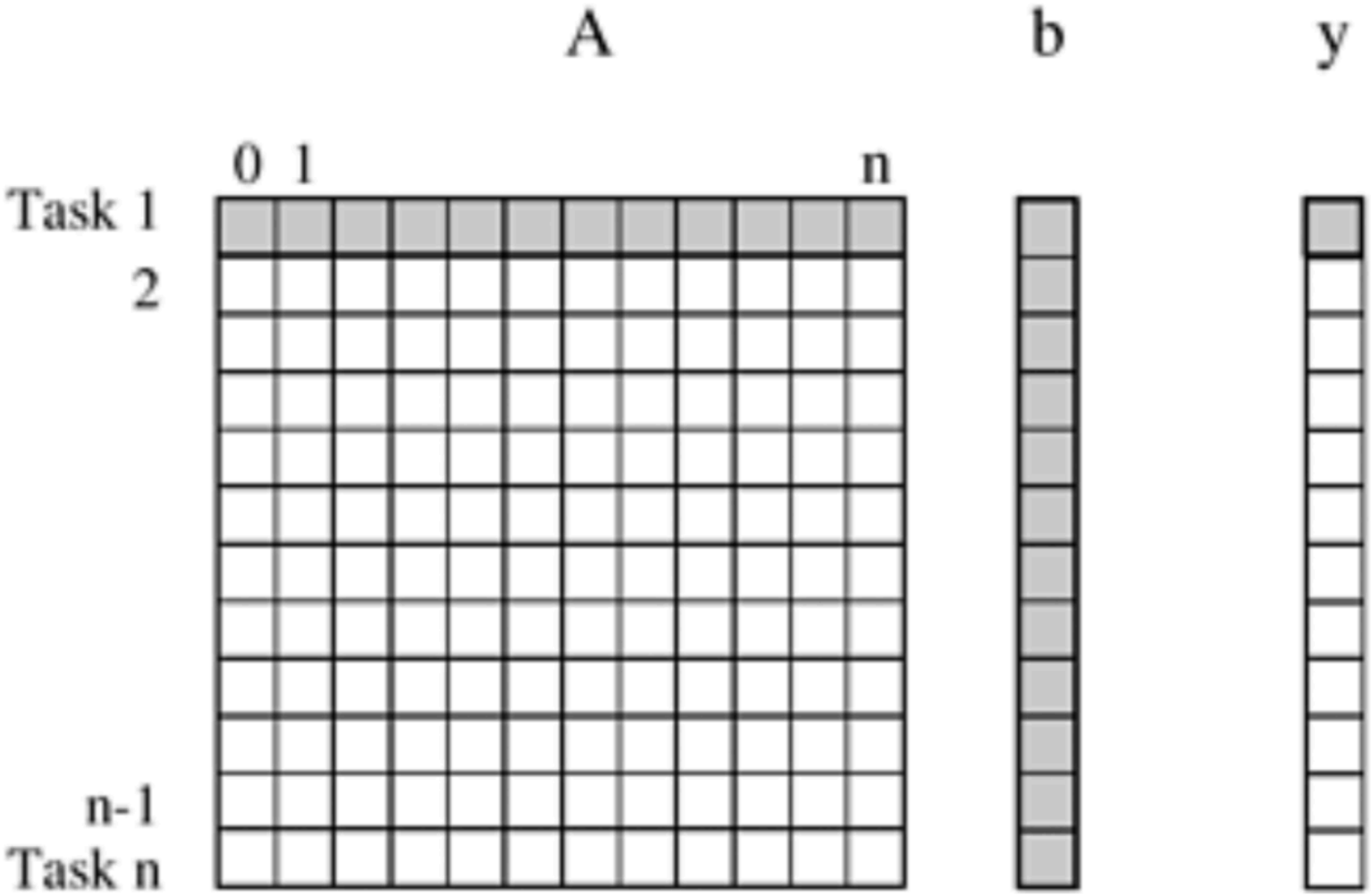




The longest directed path between any pair of start and finish nodes is the critical path

The sum of weights of nodes along this path is the critical path length

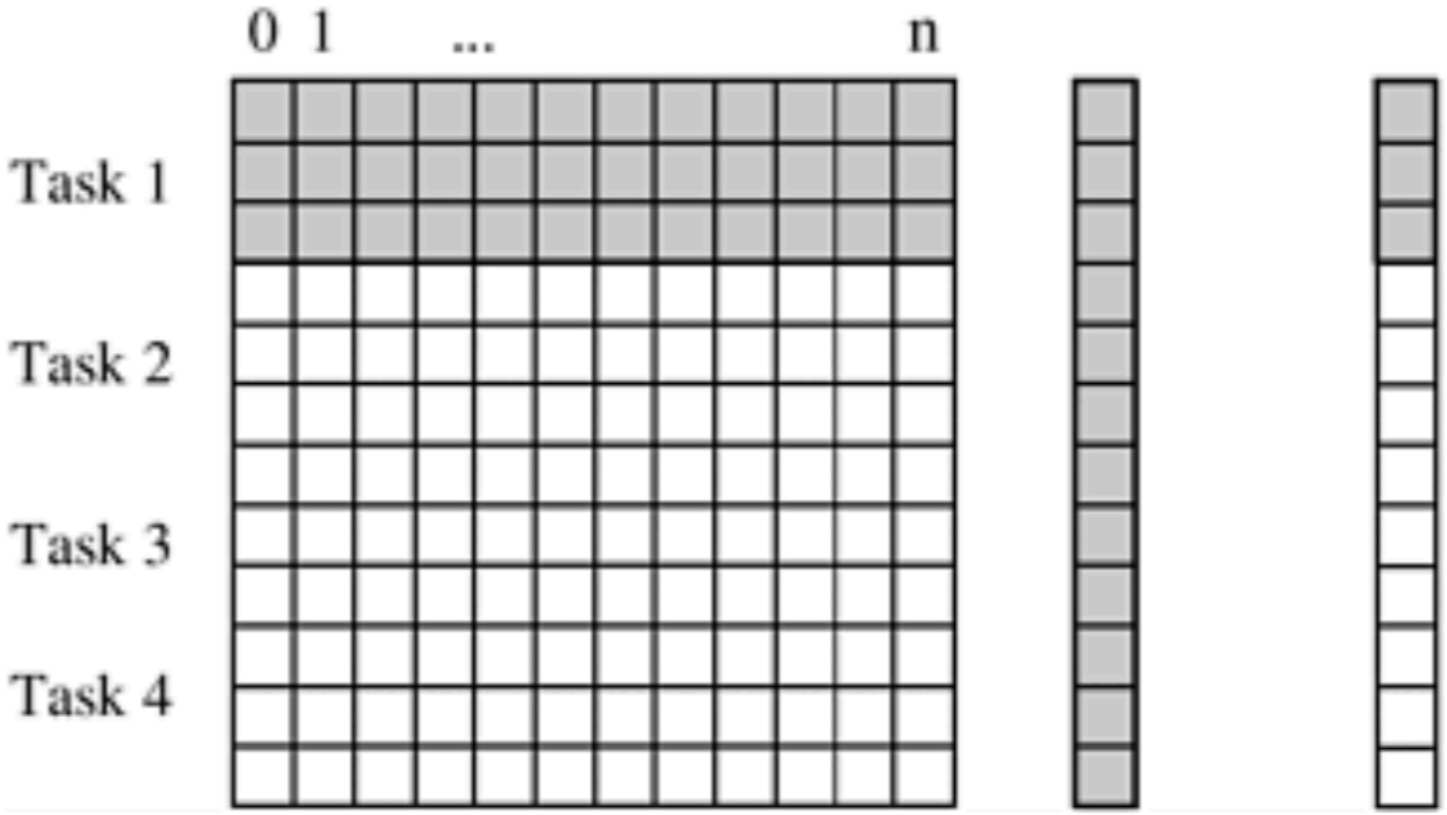
Dense matrix-vector multiplication $\mathbf{A\ b = y}$



Granularity

$$\mathbf{A\ b = y}$$

Decomposition of dense matrix-vector multiplication into four tasks



portion of matrix and input/output vectors accessed by Task 1 are highlighted

Sparse matrix-vector (SpMV) multiplication

Compute $\mathbf{y} = \mathbf{A}_{n \times n} \times \mathbf{b}$ with $\mathbf{A}$ sparse matrix

To compute the i-th entry of the product vector:

$$\mathbf{y}[i] = \sum_{j=1}^{n} \mathbf{A}[i,j]\mathbf{b}[j]$$

we need to compute the products $\mathbf{A}[i,j] \times \mathbf{b}[j]$ for only those values of j for which $A[i,j] = 0$

Sparse matrix-vector (SpMV) multiplication

Compute $\mathbf{y} = \mathbf{A}_{n \times n} \times \mathbf{b}$ with $\mathbf{A}$ sparse matrix

To compute the i-th entry of the product vector:

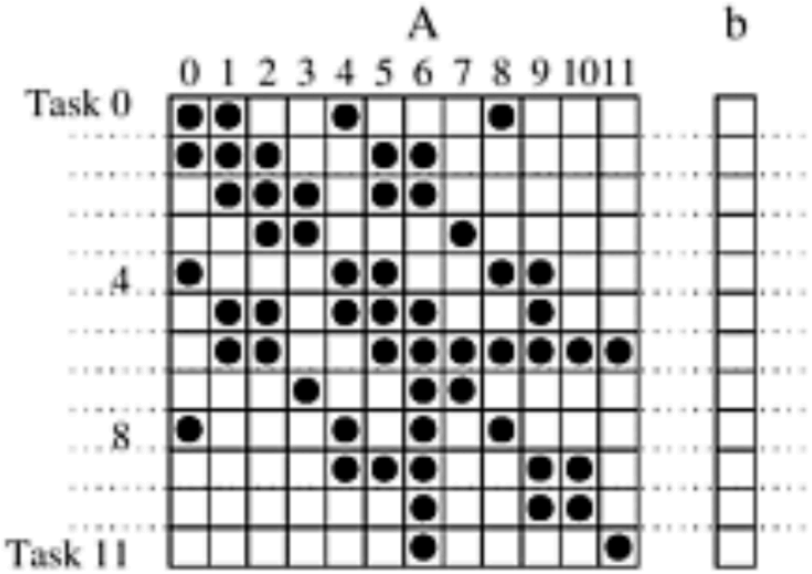
$$\mathbf{y}[i] = \sum_{j=1}^{n} \mathbf{A}[i,j]\mathbf{b}[j]$$

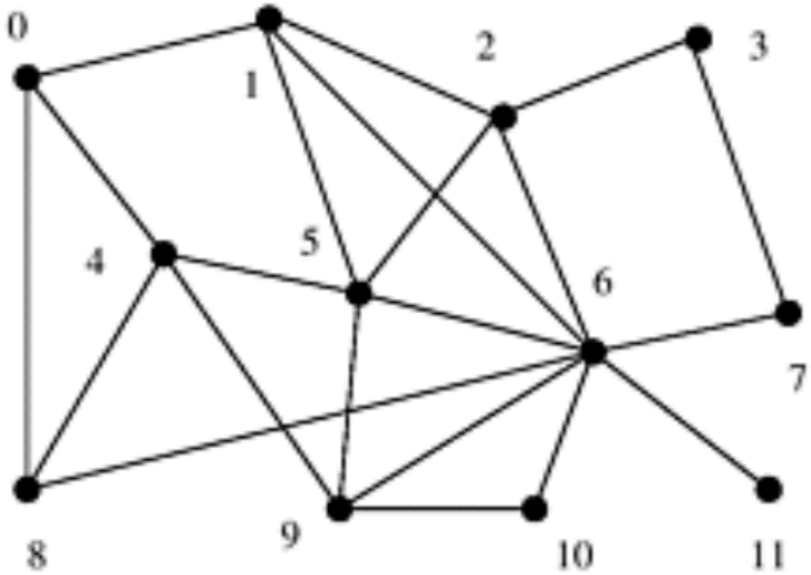
we need to compute the products $\mathbf{A}[i,j] \times \mathbf{b}[j]$ for only those values of j for which $A[i,j] = 0$

Protocol (message passing)

- Task i computes the element $y[i]$ of the output vector;
- Task i it the "owner" of row $A[i,*]$ and the element $b[i]$
- Task i gets the responsibility of sending $b[i]$ to nodes that need it

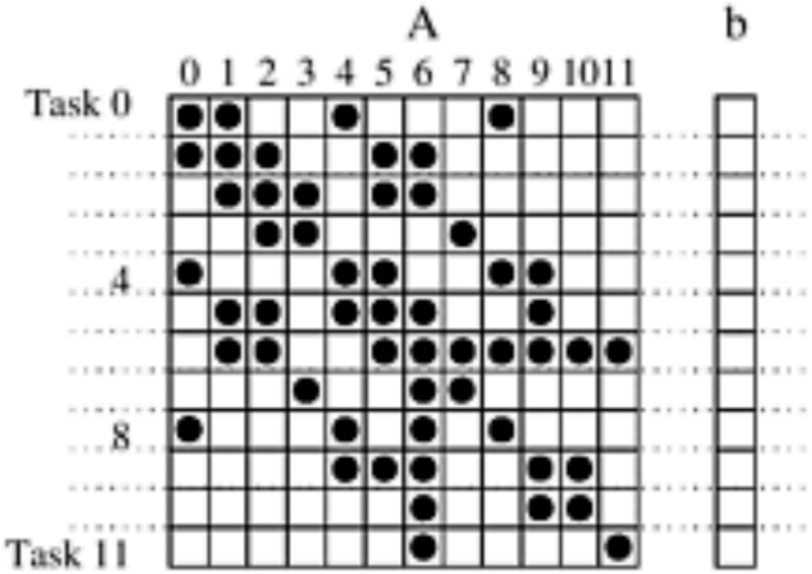
Sparse matrix-vector (SpMV) multiplication

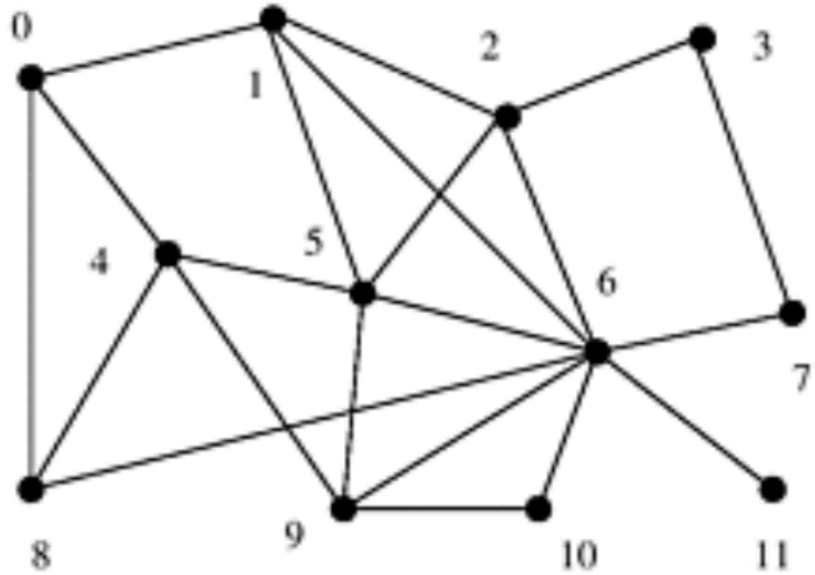




Task 4 must send b[4] to Tasks 0, 5, 8, and 9 and must get b[0], b[5], b[8], and b[9] to perform its own

Sparse matrix-vector (SpMV) multiplication





Task 4 must send b[4] to Tasks 0, 5, 8, and 9 and must get b[0], b[5], b[8], and b[9] to perform its own

Cohordination tasks

- Initial distribution of input data
- Final collection of output results

Memory and Matrices

A trivial problem?

Source: Steven G. Johnson, MIT Applied Math, 18.S096 January 2017

$$\underset{m\times p}{C} = \underset{m\times n}{A}\ \underset{n\times p}{B}$$

the "obvious" C code (rows · columns):

```
/* C = A B, where A is m x n, B is n x p,
            and C is m x p, in row-major order */
void matmul(const double *A, const double *B,
            double *C, int m, int n, int p)
{
     int i, j, k;
     for (i = 0; i < m; ++i)
          for (j = 0; j < p; ++j) {
               double sum = 0;
               for (k = 0; k < n; ++k)
                    sum += A[i*n + k] * B[k*p + j];
               C[i*p + j] = sum;
          }
}
```

for $i = 1$ **to** m

  for $j = 1$ **to** p

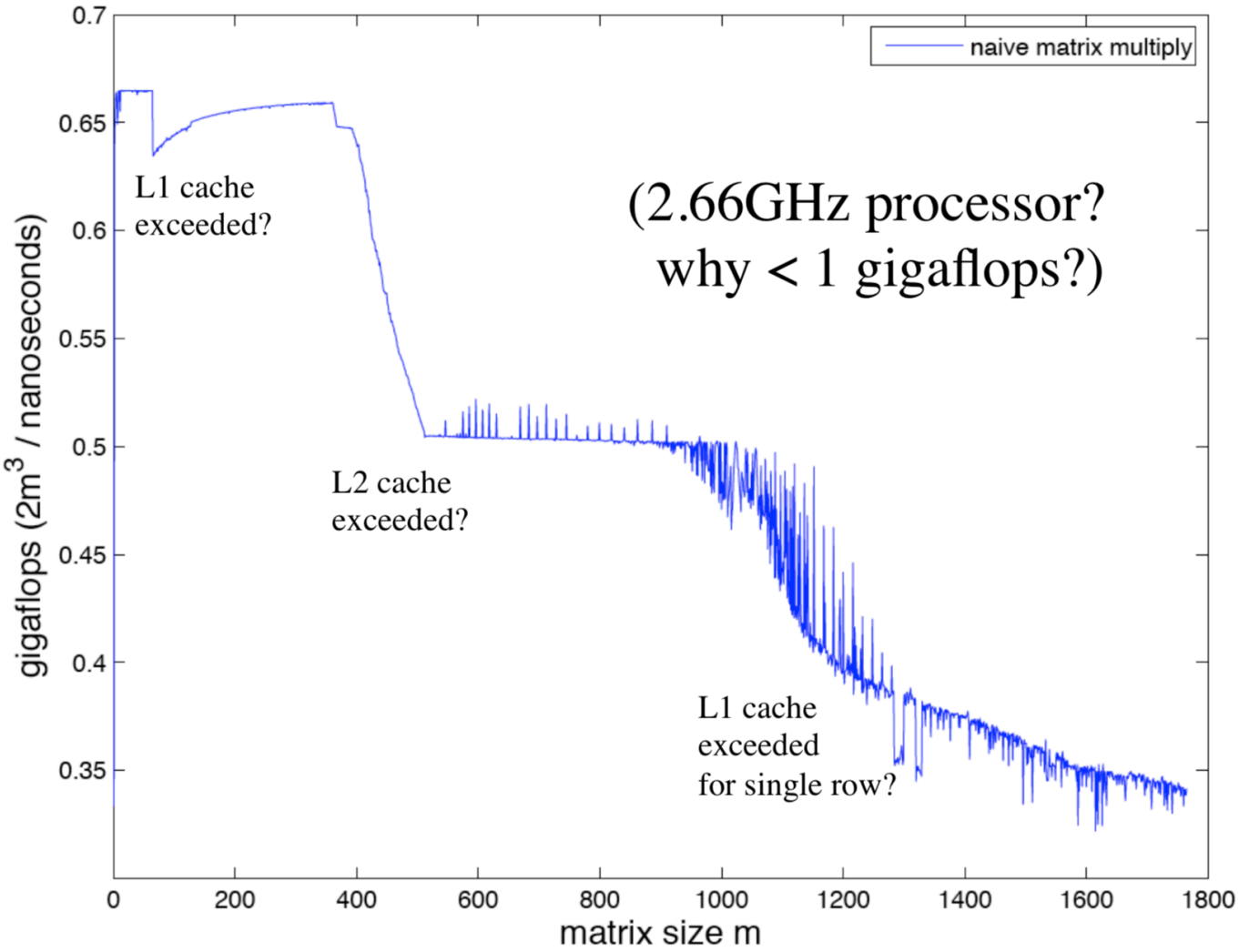
$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

$2mnp$ flops

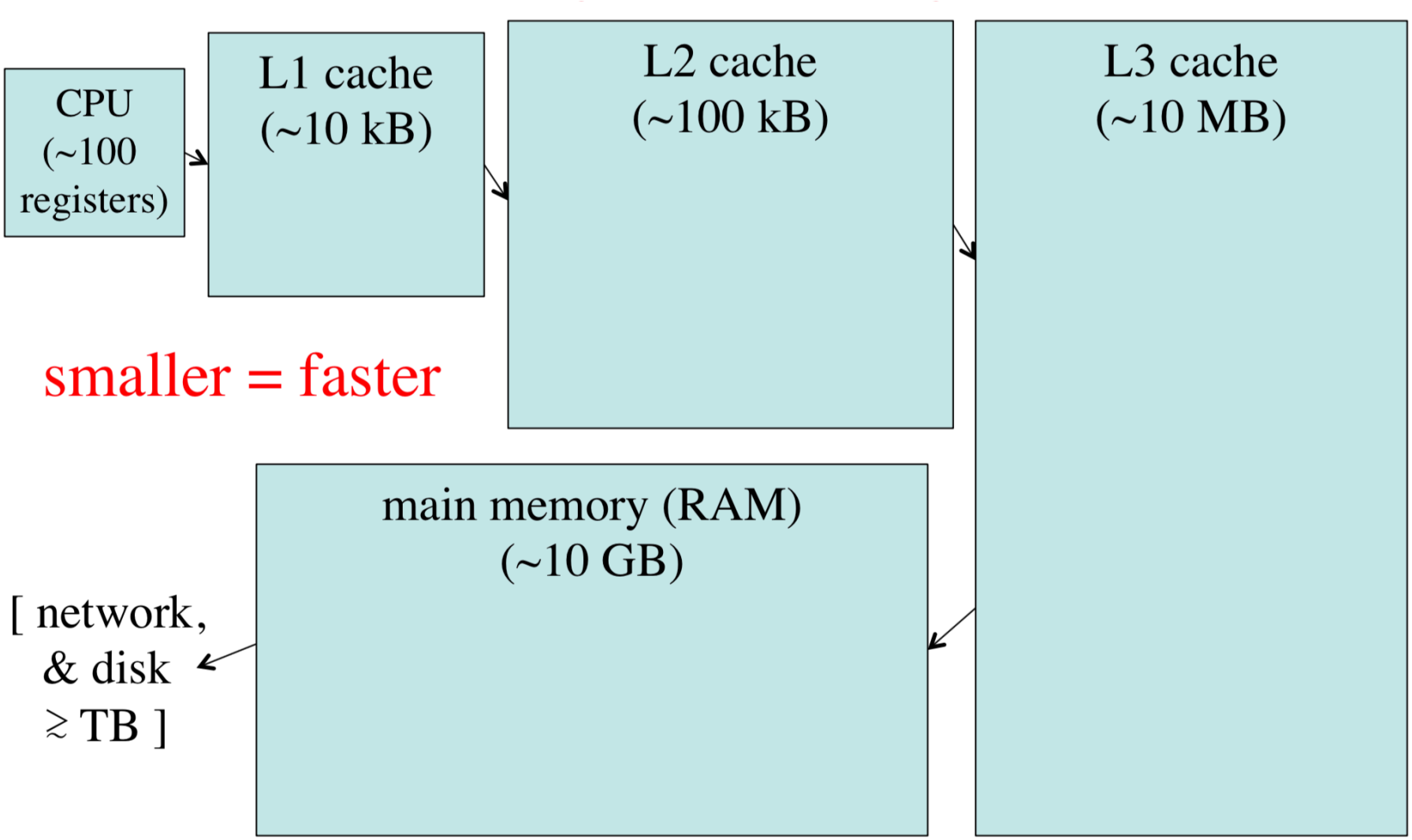
(adds+mults)

"floating-point operations"

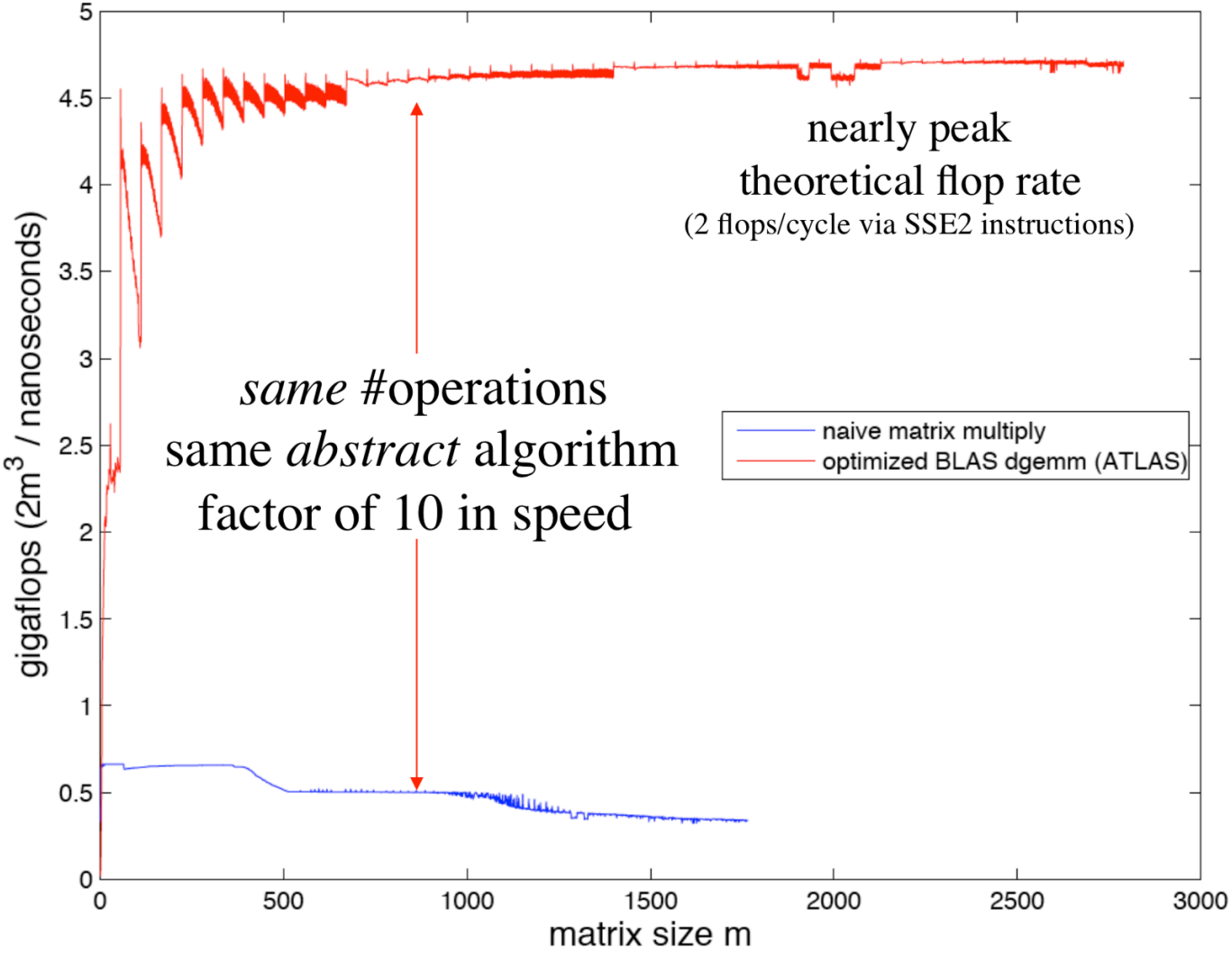
flops/time is not constant!



Speed is limited by access to the memory hierarchy



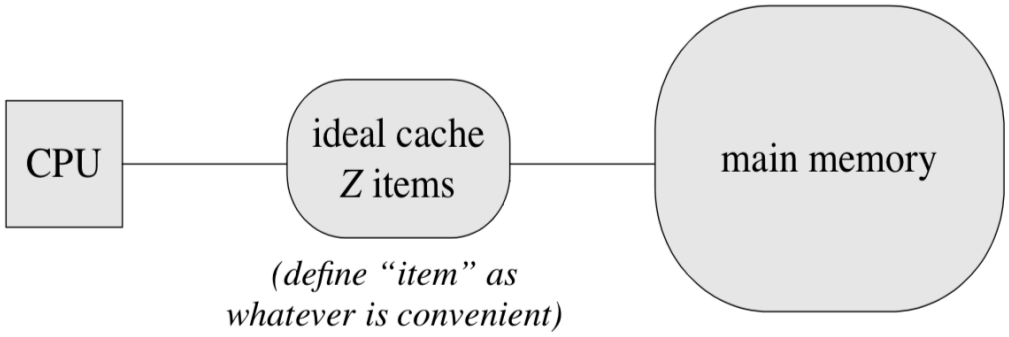
All flops are not created equal



Things to remember

- We often cannot understand performance without understanding memory efficiency (caches).
  - often ~10 times more important than arithmetic count when working with lots of data
- Computers are more complicated than you think.
- Even a trivial algorithm is nontrivial to implement *well*.
  - matrix multiplication: 10 lines of code $\longrightarrow$ 130,000+ (ATLAS)
  - getting the last factor of 2 in speed often requires wizardry (and is usually not worth it)
  - but factors of 10 are often worthwhile and not too hard…

Ideal Cache Model



when CPU needs an item, either:
- cache hit: already in cache (fast)
- cache miss: load into cache (slow)

goal: analyze # of cache misses

Simplification: cache is "ideal"
- fully associative: any item in memory can replace any item in cache
- optimal replacement: cache miss replaces "best" item
  (= item not needed for longest time in the future)
  … within a ~constant factor of more realistic caches

Cache complexity: for problem with n items, cache size Z,
want # misses for large n in "big O" or "big Θ" notation
(ignoring constant factors), e.g. $\Theta(n/Z)$

Strategy for efficient cache utilization: Maximize temporal locality

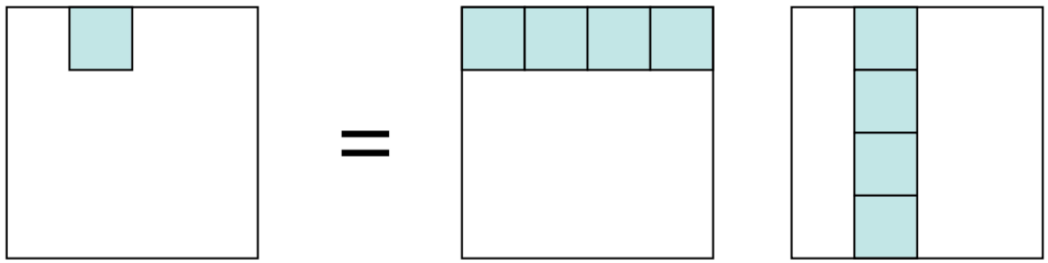
Strategy for efficient cache utilization:

Maximize temporal locality

Once we read an item from memory, we want as much computation as possible before reading the next item…

Re-arrange our algorithm so that computations on the same data occur at close to the same time.

(optimal) Blocked Matrix Multiply



C $n \times n$ = A $n \times n$ B $n \times n$

divide matrices into $b \times b$ blocks of $b = \sqrt{Z/3}$ numbers
load 3 $b \times b$ blocks into cache and *multiply blocks in-cache*

(provably optimal) cache misses: $\Theta(n^3/b^3) \times \Theta(b^2) = \Theta(n^3/\sqrt{Z})$

$\Theta(n^3/b^3)$: # block × block multiplications

$\Theta(b^2)$: # cache misses per block

(optimal) Cache-Oblivious Matrix Multiply

$$\begin{pmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathbf{0}_{1,1} & \mathbf{0}_{1,2} \\ \mathbf{0}_{2,1} & \mathbf{0}_{2,2} \end{pmatrix} + \begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{pmatrix} \times \begin{pmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{pmatrix}$$

(optimal) Cache-Oblivious Matrix Multiply

$$\begin{pmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathbf{0}_{1,1} & \mathbf{0}_{1,2} \\ \mathbf{0}_{2,1} & \mathbf{0}_{2,2} \end{pmatrix} + \begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{pmatrix} \times \begin{pmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{pmatrix}$$

$$\mathbf{C}_{1,1} = \mathbf{0}_{1,1} + \mathbf{A}_{1,1} \times \mathbf{B}_{1,1} + \mathbf{A}_{1,2} \times \mathbf{B}_{2,1}$$

$$\mathbf{C}_{1,2} = \mathbf{0}_{1,2} + \mathbf{A}_{1,1} \times \mathbf{B}_{1,2} + \mathbf{A}_{1,2} \times \mathbf{B}_{2,2}$$

$$\mathbf{C}_{2,1} = \mathbf{0}_{2,1} + \mathbf{A}_{2,1} \times \mathbf{B}_{1,1} + \mathbf{A}_{2,2} \times \mathbf{B}_{2,1}$$

$$\mathbf{C}_{2,2} = \mathbf{0}_{2,2} + \mathbf{A}_{2,1} \times \mathbf{B}_{1,2} + \mathbf{A}_{2,2} \times \mathbf{B}_{2,2}$$

(optimal) Cache-Oblivious Matrix Multiply

$$\begin{pmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathbf{0}_{1,1} & \mathbf{0}_{1,2} \\ \mathbf{0}_{2,1} & \mathbf{0}_{2,2} \end{pmatrix} + \begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{pmatrix} \times \begin{pmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{pmatrix}$$

$$\mathbf{C}_{1,1} = \mathbf{0}_{1,1} + \mathbf{A}_{1,1} \times \mathbf{B}_{1,1} + \mathbf{A}_{1,2} \times \mathbf{B}_{2,1}$$

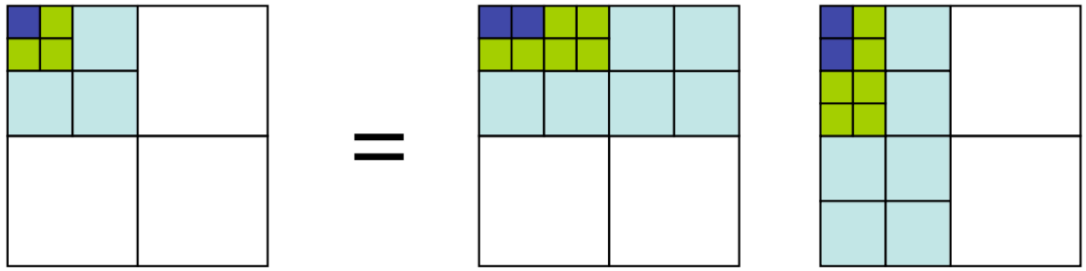
$$\mathbf{C}_{1,2} = \mathbf{0}_{1,2} + \mathbf{A}_{1,1} \times \mathbf{B}_{1,2} + \mathbf{A}_{1,2} \times \mathbf{B}_{2,2}$$

$$\mathbf{C}_{2,1} = \mathbf{0}_{2,1} + \mathbf{A}_{2,1} \times \mathbf{B}_{1,1} + \mathbf{A}_{2,2} \times \mathbf{B}_{2,1}$$

$$\mathbf{C}_{2,2} = \mathbf{0}_{2,2} + \mathbf{A}_{2,1} \times \mathbf{B}_{1,2} + \mathbf{A}_{2,2} \times \mathbf{B}_{2,2}$$

8 block-sums & 8 block-products

(optimal) Cache-Oblivious Matrix Multiply



C $n \times n$ = A $n \times n$ B $n \times n$

divide and conquer:

divide C into 4 blocks

compute block multiply recursively

achieves optimal $\Theta(n^3/\sqrt{Z})$ cache complexity

without knowing the Z, works for nested caches too

(optimal) Cache-Oblivious Matrix Multiply

Source: Steven G. Johnson, MIT Applied Math, 18.S096 January 2017

```
function add_matmul_rec!(m,n,p, i0,j0,k0, C,A,B)
    if m+n+p <= 64    # base case: naive matmult for sufficiently large matrices
        for i = 1:m
            for k = 1:p
                c = zero(eltype(C))
                for j = 1:n
                    @inbounds c += A[i0+i,j0+j] * B[j0+j,k0+k]
                end
                @inbounds C[i0+i,k0+k] += c
            end
        end
    else
        m2 = m ÷ 2; n2 = n ÷ 2; p2 = p ÷ 2
        add_matmul_rec!(m2, n2, p2, i0, j0, k0, C, A, B)

        add_matmul_rec!(m-m2, n2, p2, i0+m2, j0, k0, C, A, B)
        add_matmul_rec!(m2, n-n2, p2, i0, j0+n2, k0, C, A, B)
        add_matmul_rec!(m2, n2, p-p2, i0, j0, k0+p2, C, A, B)

        add_matmul_rec!(m-m2, n-n2, p2, i0+m2, j0+n2, k0, C, A, B)
        add_matmul_rec!(m2, n-n2, p-p2, i0, j0+n2, k0+p2, C, A, B)
        add_matmul_rec!(m-m2, n2, p-p2, i0+m2, j0, k0+p2, C, A, B)

        add_matmul_rec!(m-m2, n-n2, p-p2, i0+m2, j0+n2, k0+p2, C, A, B)
    end
    return C
end
```

Sources

Sources of lecture materials

1. Grama, Gupta, Karypis, Kumar Introduction to Parallel Computing
2. 18.S096, IAP 2018: Performance Computing in a High Level Language
3. Steven G. Johnson, More Dots: Syntactic Loop Fusion in Julia