HPC & Parallel Programming

Start from An Example

Kun Suo

Computer Science, Kennesaw State University

https://kevinsuo.github.io/

An example of HPC & Parallel Programming: Matrix Multiplication

$$\begin{matrix} & & \vec{b_1} & \vec{b_2} \\ & & \downarrow & \downarrow \\ \vec{a_1}\rightarrow & \begin{bmatrix} 1 & 7 \\ 2 & 4 \end{bmatrix} & \cdot \begin{bmatrix} 3 & 3 \\ 5 & 2 \end{bmatrix} & = \begin{bmatrix} \vec{a_1} \cdot \vec{b_1} & \vec{a_1} \cdot \vec{b_2} \\ \vec{a_2} \cdot \vec{b_1} & \vec{a_2} \cdot \vec{b_2} \end{bmatrix} \\ \vec{a_2}\rightarrow & & & \end{matrix}$$

$A \qquad B \qquad C$

Matrix multiply

https://github.com/kevinsuo/CS7172/blob/master/matrix.c

```c
int main()
{
        initMatrix();

        double time_spent = 0.0;
        clock_t begin = clock();

        matrixMultiply();

        clock_t end = clock();
        time_spent += (double)(end - begin) / CLOCKS_PER_SEC;
        printf("Time elpased is %f seconds", time_spent);

        return 0;
}
```

Matrix multiply

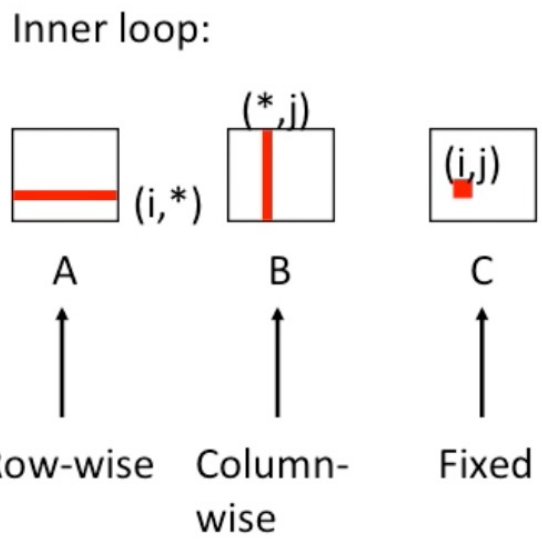
https://github.com/kevinsuo/CS7172/blob/master/matrix.c

```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#define N 1000

double A[N][N], B[N][N], C[N][N];

void initMatrix()
{
        int i, j = 0;
        for (i = 0; i < N; i++) {
                for (j = 0; j < N; j++) {
                        A[i][j] = rand() % 100 + 1;  //generate a number between [1, 100]
                        B[i][j] = rand() % 100 + 1;  //generate a number between [1, 100]
                }
        }
}
```

Matrix multiply

https://github.com/kevinsuo/CS7172/blob/master/matrix.c

```
void matrixMultiply() {
        int i, j, k = 0;
        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];
                        }
                }
        }
}
```

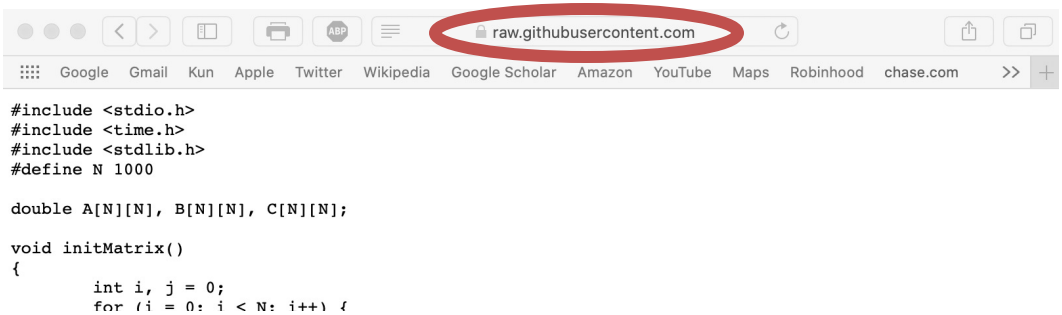


$$\Theta\left(n^3\right)$$

Matrix multiply

https://github.com/kevinsuo/CS7172/blob/master/matrix.c

2. Copy the URL



3.

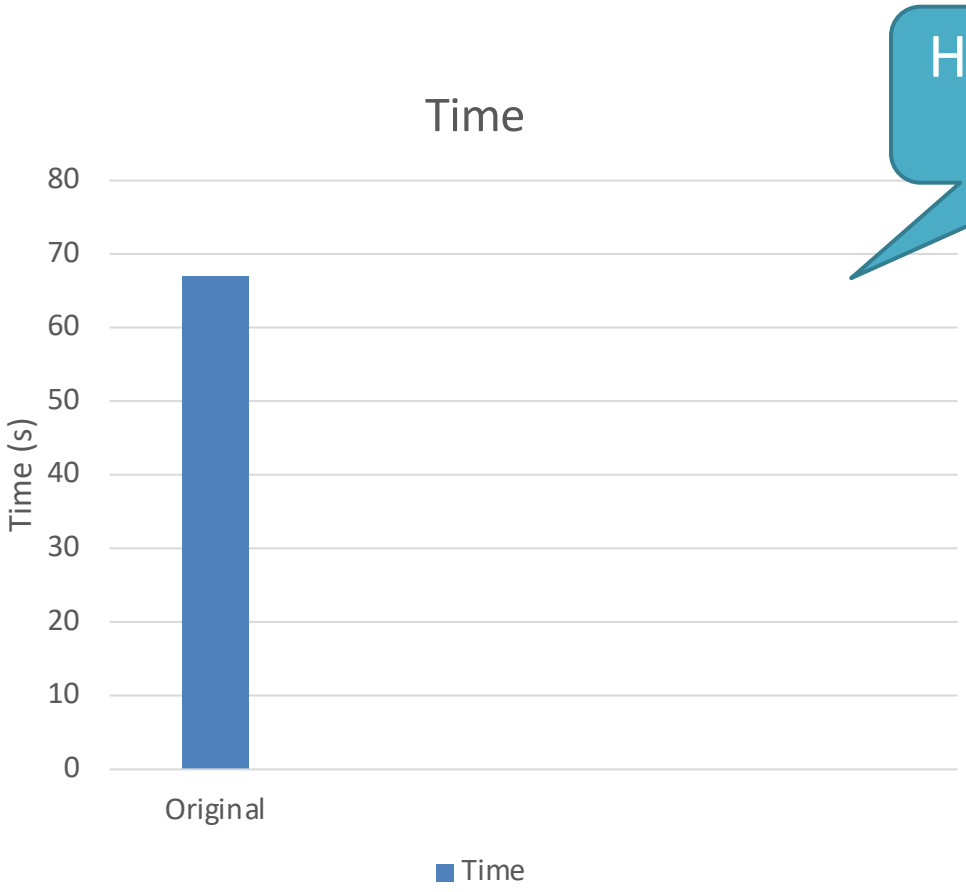
$ wget URL

$ gcc filename.c –o filename.o

$ ./filename.o

(if no wget/gcc,
$ sudo apt install wget, gcc)

Matrix multiply



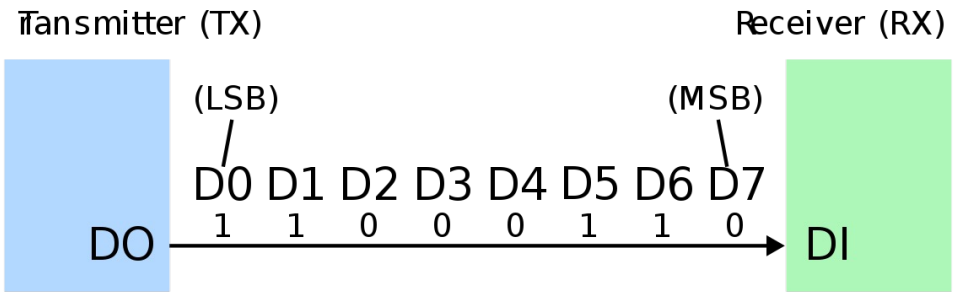
How to run it faster?

1. Accelerate serial execution

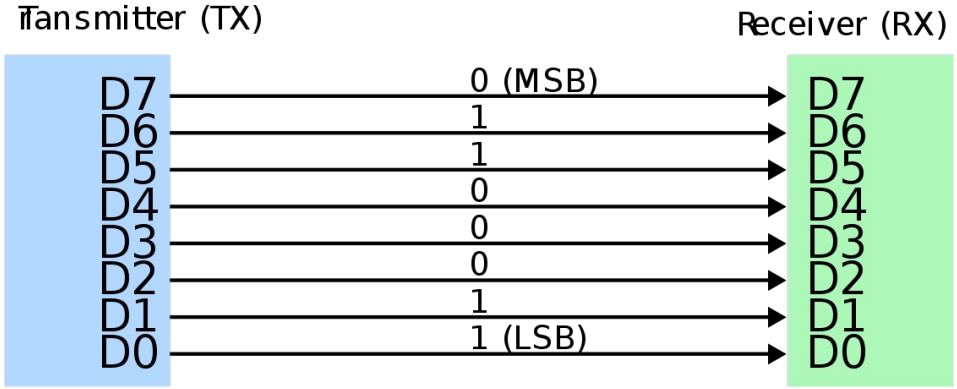
2. Accelerate in parallel

How to run it faster?

1. Accelerate serial execution
Reduce unnecessary steps



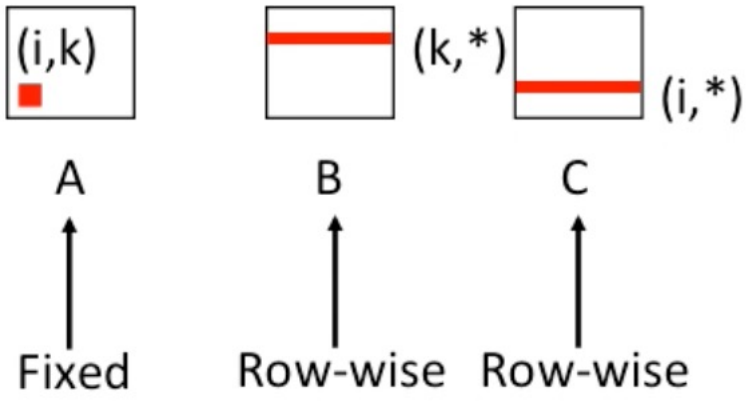
2. Accelerate in parallel



Option 1: Optimization using locality

```
void matrixMultiply() {
        int i, j, k = 0;
        for (k = 0; k < N; k++) {
                for (i = 0; i < N; i++) {
                        for (j = 0; j < N; j++) {
                                C[i][j] += A[i][k] * B[k][j];
                        }
                }
        }
}
```

Inner loop:



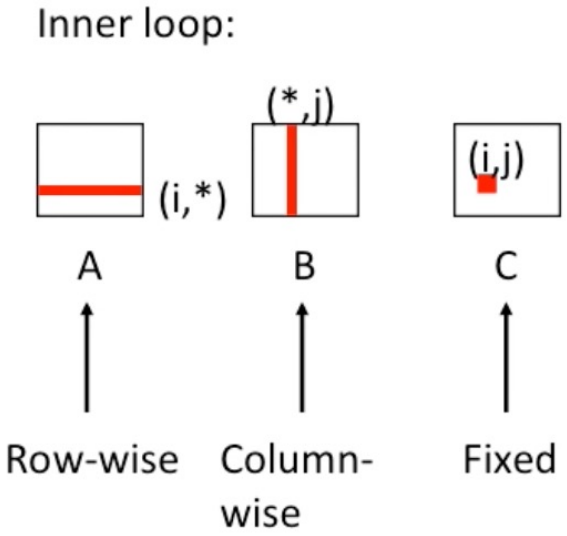
https://github.com/kevinsuo/CS7172/blob/master/matrix-opt.c

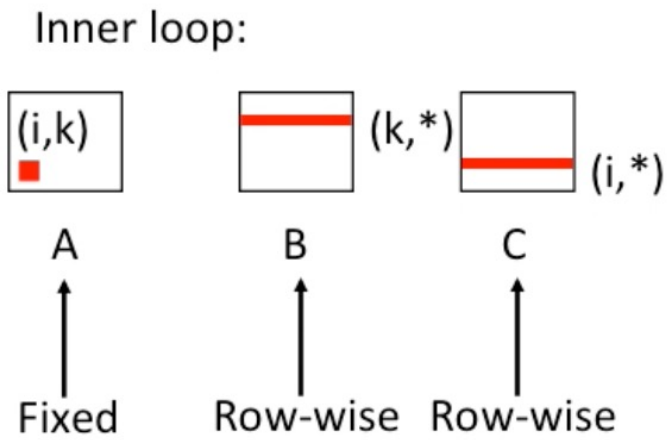
Option 1: Optimization using locality

```
ksuo@ksuo-VirtualBox ~/cs7172> ./a.o
Time elpased is 67.452589 seconds
ksuo@ksuo-VirtualBox ~/cs7172>
ksuo@ksuo-VirtualBox ~/cs7172>
ksuo@ksuo-VirtualBox ~/cs7172>
ksuo@ksuo-VirtualBox ~/cs7172>
ksuo@ksuo-VirtualBox ~/cs7172>
ksuo@ksuo-VirtualBox ~/cs7172> ./a2.o
Time elpased is 18.149353 seconds
```

N=2000

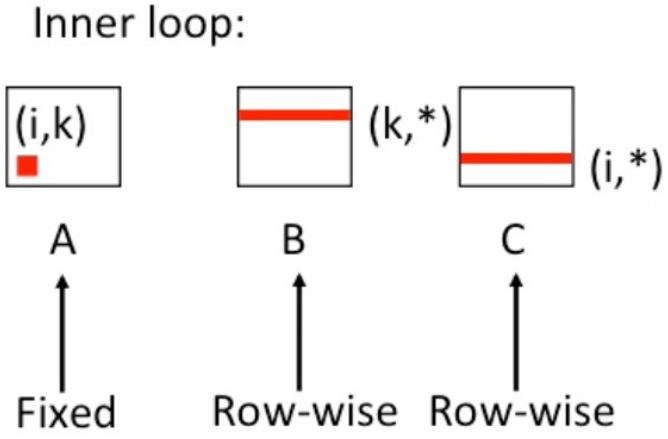
3.7x



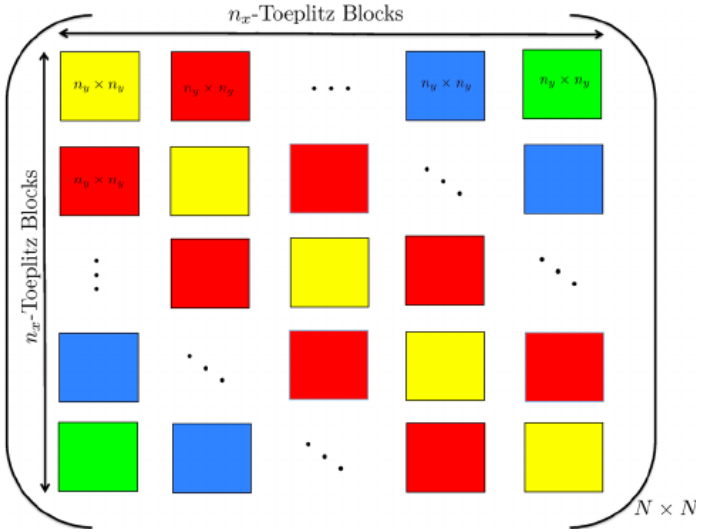


Option 1: Optimization using locality

- Temporal locality
  - Every inner loop reuse the value of A[i, k]



- Spatial locality
  - Divide the large matrix into smaller ones and put it inside the cache during calculation

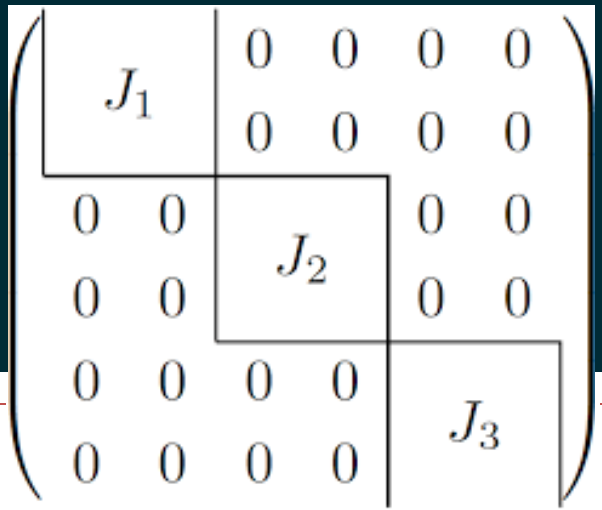


Option 1: Optimization using locality

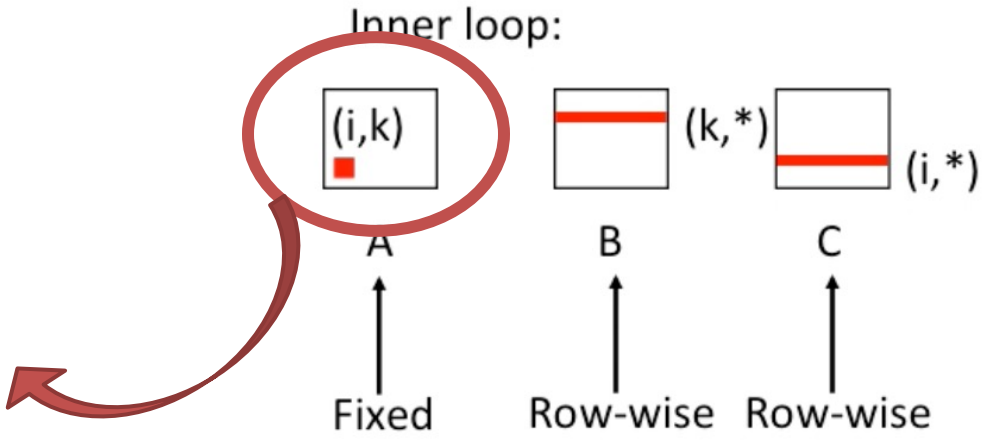
```
void matrixMultiply() {
        int i, j, k = 0;
        int i2, j2, k2 = 0;

        for (k2 = 0; k2 < N; k2+=BLOCK_SIZE) {
                for (i2 = 0; i2 < N; i2+=BLOCK_SIZE) {
                        for (j2 = 0; j2 < N; j2+=BLOCK_SIZE) {
                                //inside each block
                                for (k = k2; k < k2+BLOCK_SIZE; k++) {
                                        for (i = i2; i < i2+BLOCK_SIZE; i++) {
                                                for (j = j2; j < j2+BLOCK_SIZE; j++) {
                                                        C[i][j]+= A[i][k] * B[k][j];
                                                }
                                        }
                                }

                        }
                }
        }
}
```

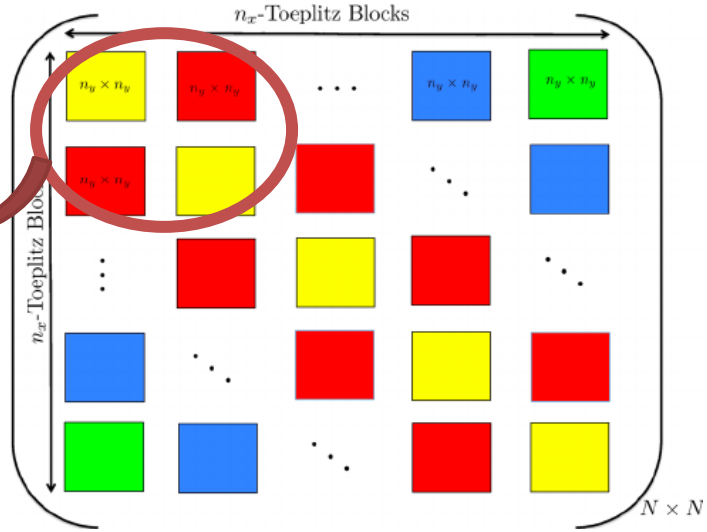
https://github.com/kevinsuo/CS7172/blob/master/matrix-opt2.c

N = 2000

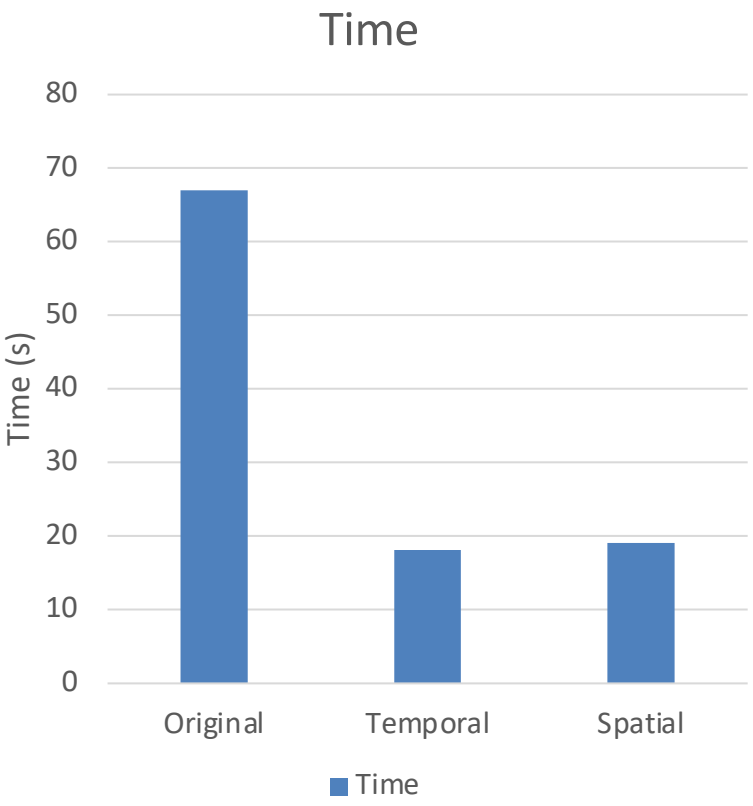


Option 1: Optimization using locality





Option 1: Optimization using locality




```
ksuo@ksuo-VirtualBox ~/cs7172> ./a.o
Time elpased is 67.845517 seconds
ksuo@ksuo-VirtualBox ~/cs7172>
ksuo@ksuo-VirtualBox ~/cs7172>
ksuo@ksuo-VirtualBox ~/cs7172>
ksuo@ksuo-VirtualBox ~/cs7172>
ksuo@ksuo-VirtualBox ~/cs7172> ./a3.o
Time elpased is 19.115410 seconds
```

Optimal 2: Optimization using parallel

- The algorithm uses a divide-and-conquer approach. When the matrix order is very large, a recursive formula is used for calculation. The relevant recursive formula is described as follows:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Strassen algorithm

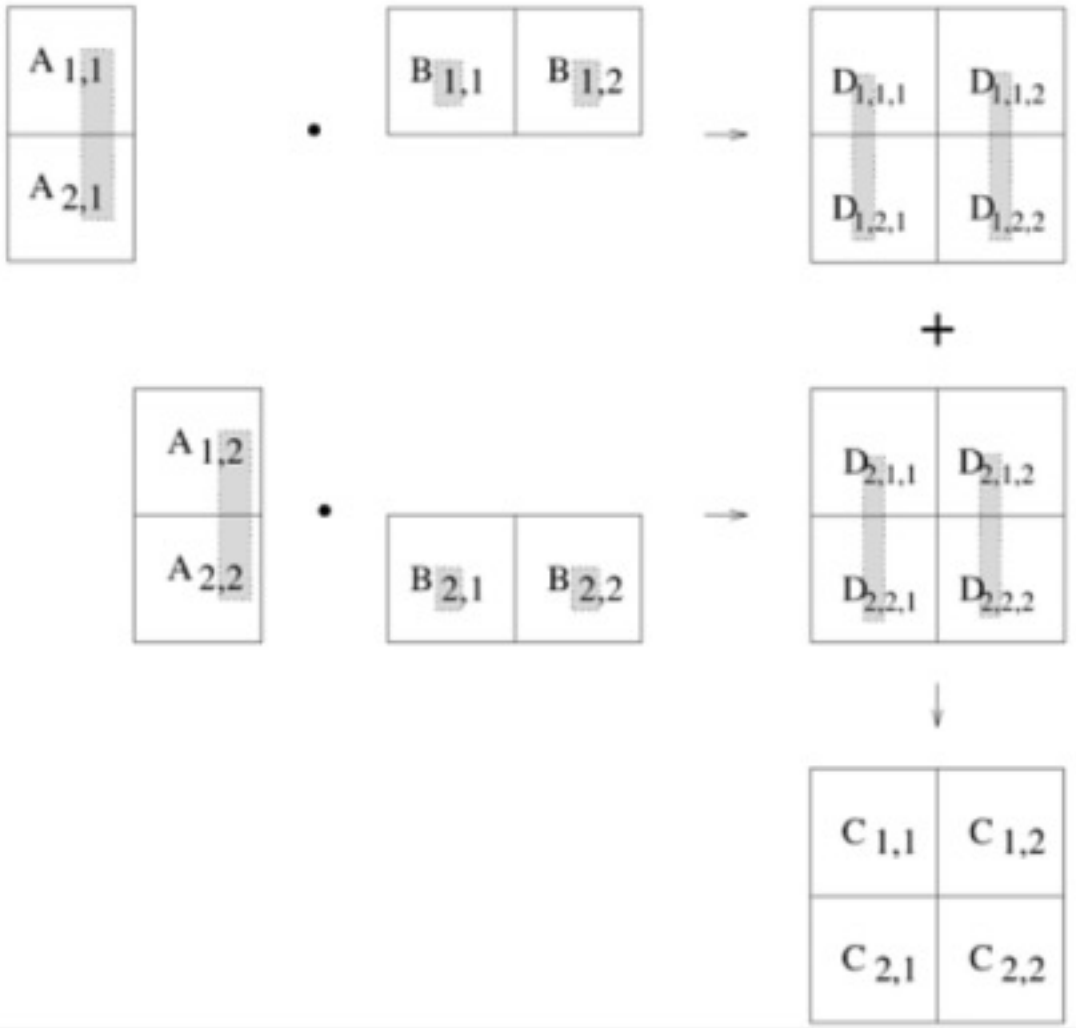
$$\begin{aligned} &\text{Task 1: } C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ &\text{Task 2: } C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ &\text{Task 3: } C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ &\text{Task 4: } C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

$$\Theta\left(n^{lg7}\right) \approx \Theta\left(n^{2.81}\right)$$

Optimal 2: Optimization using parallel

Thread 1:

Thread 2:



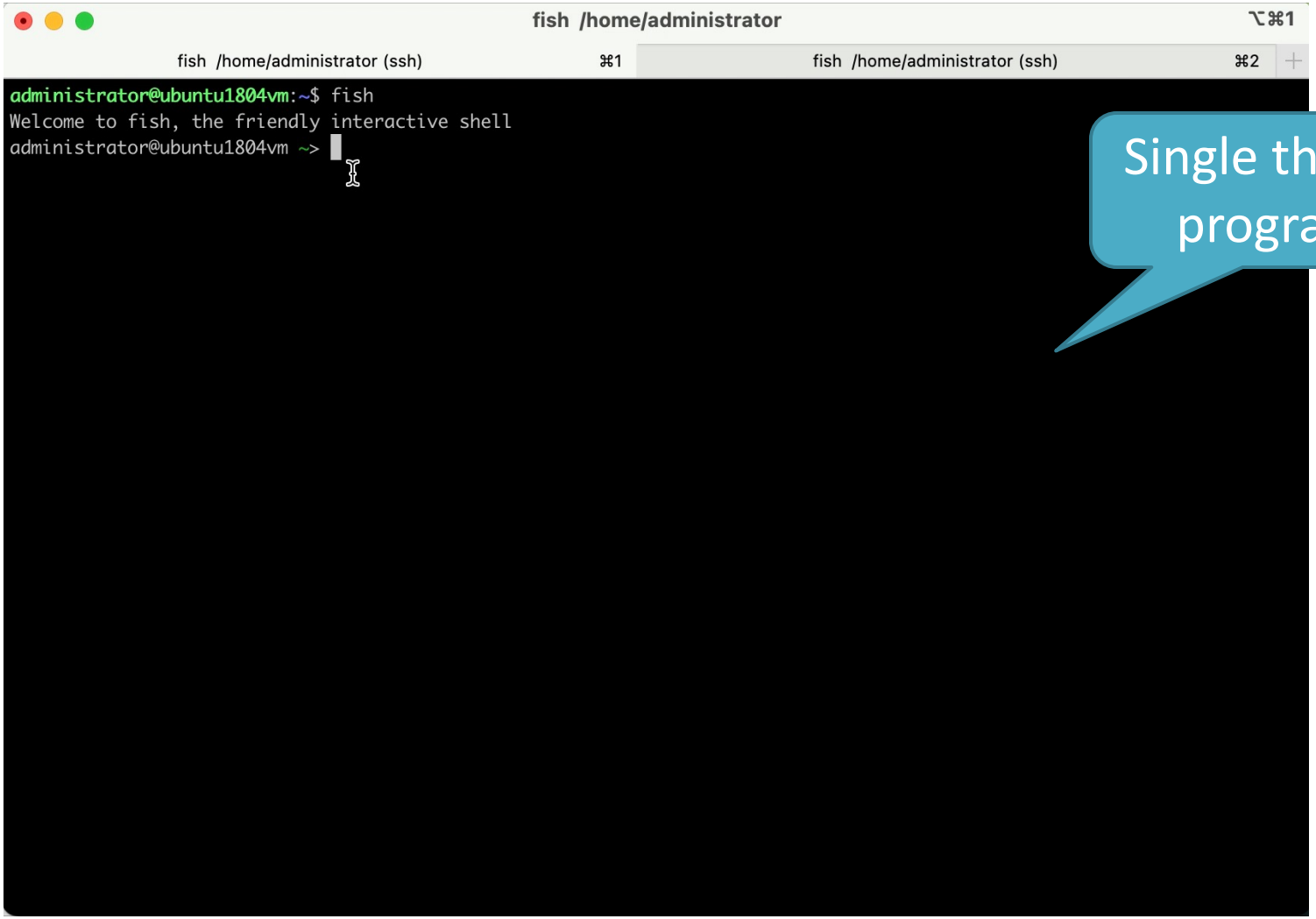
Optimization and Speedup

	N=200	N=400	N=800	N=1600
matrix				
matrix-opt1				
matrix-opt2				

	N=200	N=400	N=800	N=1600
matrix				
matrix-opt1				
matrix-opt2				

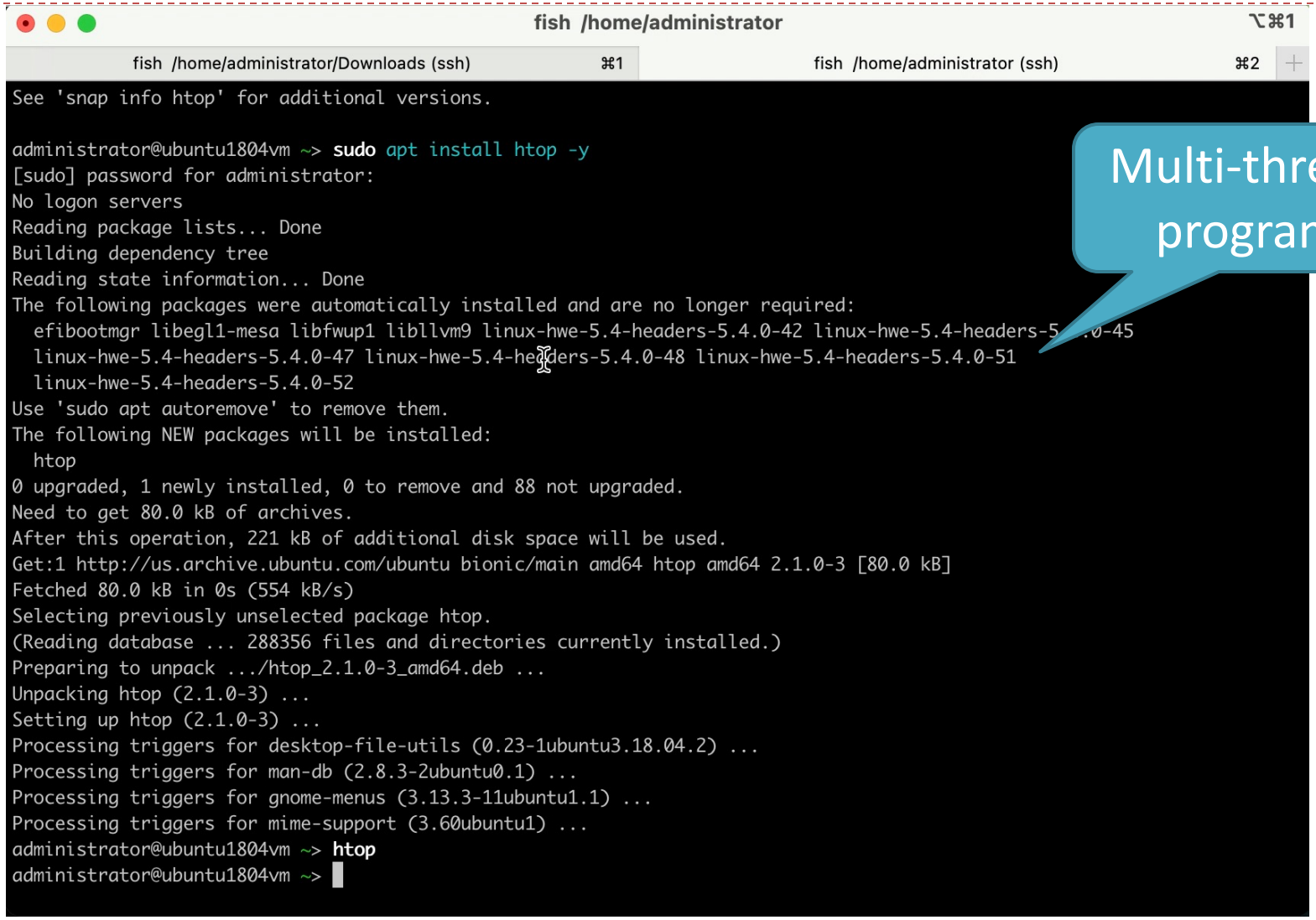
Single thread app demo

https://youtu.be/dIsBhvhQ9mA

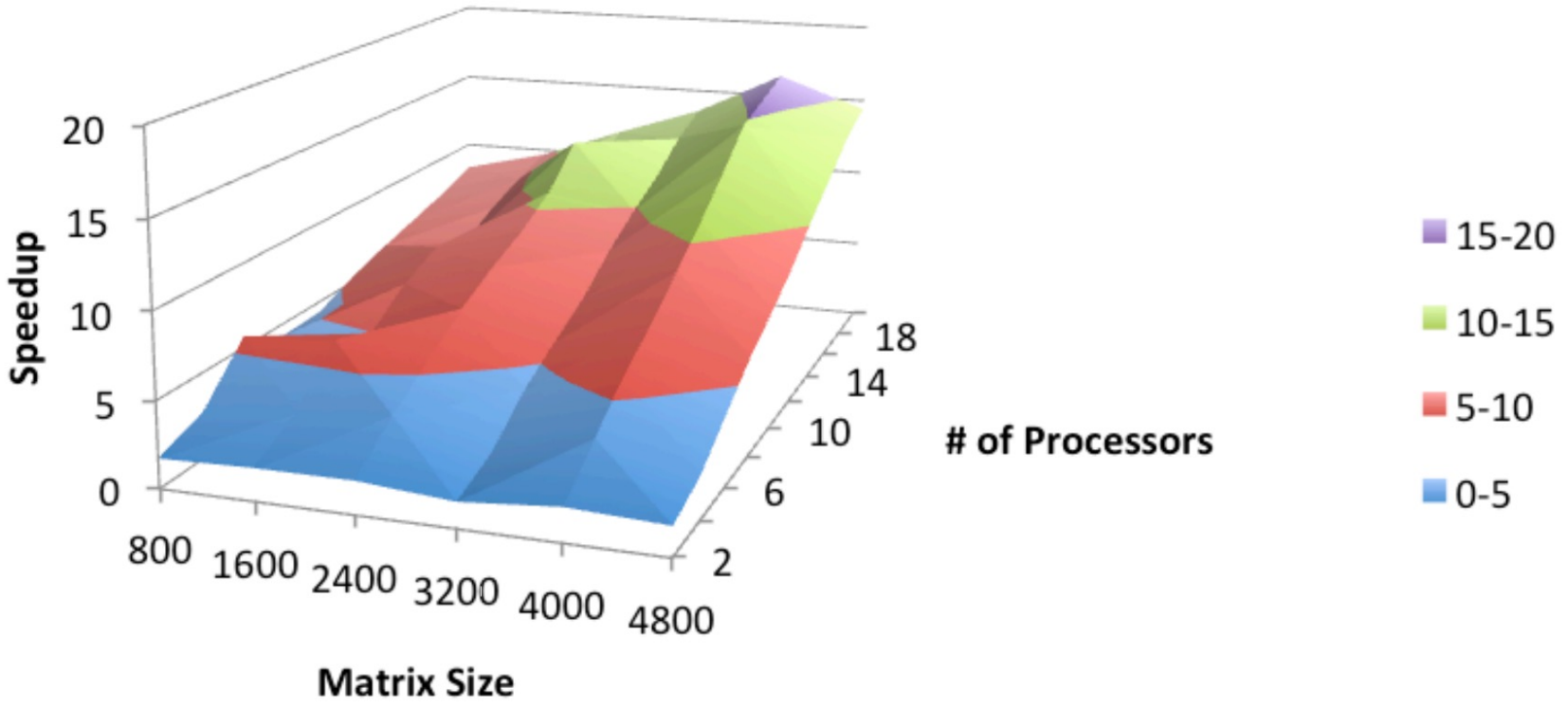


Multi-thread app demo

https://youtu.be/ubLB2fb8cdc

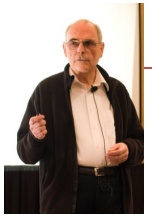


Optimal 2: Optimization using parallel



https://www.cse.unr.edu/~fredh/class/415/Nolan/matrix_multiplication/writeup.pdf

Reading Materials



- Strassen algorithm
  - In 1969, Volker Strassen proposed a matrix multiplication algorithm with a complexity of $O(n^{2.807})$: Link. This was the first time in history that computational complexity of matrix multiplication was reduced below $O(n^3)$.

- Coppersmith–Winograd algorithm
  - In 1990, Don Coppersmith and Shmuel Winograd made a groundbreaking achievement by reducing the complexity to $O(n^{2.3727})$. Paper: Link

Without Strassen algorithm

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

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

8 submatrix multiplication

4 matrix additions

Each equation requires two matrix multiplications and one matrix addition. Using T(n) to represent the multiplication between two matrices, the above equation can be expressed as the following recursive formula:

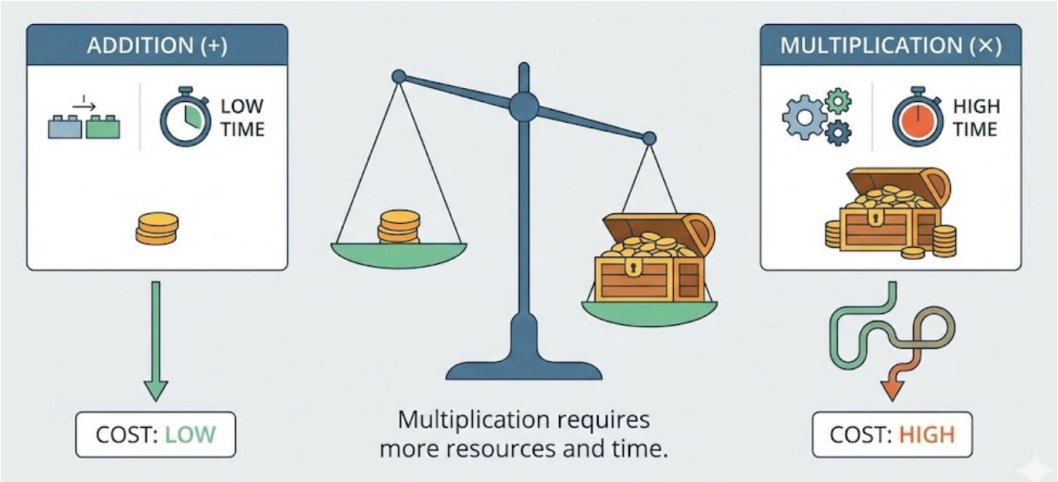
$$T(n) = 8T(n/2) + \Theta(n^2) \Rightarrow T(n) = \Theta(n^3)$$

8 submatrix multiplication

4 matrix additions, with submatrices of size n/2 x n/2.

Strassen algorithm basic idea

- Multiplication is expensive than additions



- Reduce the number of matrix multiplications by adding the cost of more matrix additions

Strassen algorithm basic idea

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \rightarrow \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$\begin{aligned} S_1 &= B_{12} - B_{22} \\ S_2 &= A_{11} + A_{12} \\ S_3 &= A_{21} + A_{22} \\ S_4 &= B_{21} - B_{11} \\ S_5 &= A_{11} + A_{22} \\ S_6 &= B_{11} + B_{22} \\ S_7 &= A_{12} - A_{22} \\ S_8 &= B_{21} + B_{22} \\ S_9 &= A_{11} - A_{21} \\ S_{10} &= B_{11} + B_{12} \end{aligned}$$

10 intermediate matrices

$$\begin{aligned} P_1 &= A_{11} \times S_1 \\ P_2 &= S_2 \times B_{22} \\ P_3 &= S_3 \times B_{11} \\ P_4 &= A_{22} \times S_4 \\ P_5 &= S_5 \times S_6 \\ P_6 &= S_7 \times S_8 \\ P_7 &= S_9 \times S_{10} \end{aligned}$$

7 submatrix multiplication

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

18 matrix additions

Strassen algorithm time complexity

$$
\begin{aligned}
P_1 &= A_{11} \times S_1 \\
P_2 &= S_2 \times B_{22} \\
P_3 &= S_3 \times B_{11} \\
P_4 &= A_{22} \times S_4 \\
P_5 &= S_5 \times S_6 \\
P_6 &= S_7 \times S_8 \\
P_7 &= S_9 \times S_{10}
\end{aligned}
$$

7 submatrix multiplication

$$
\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_5 + P_1 - P_3 - P_7
\end{aligned}
$$

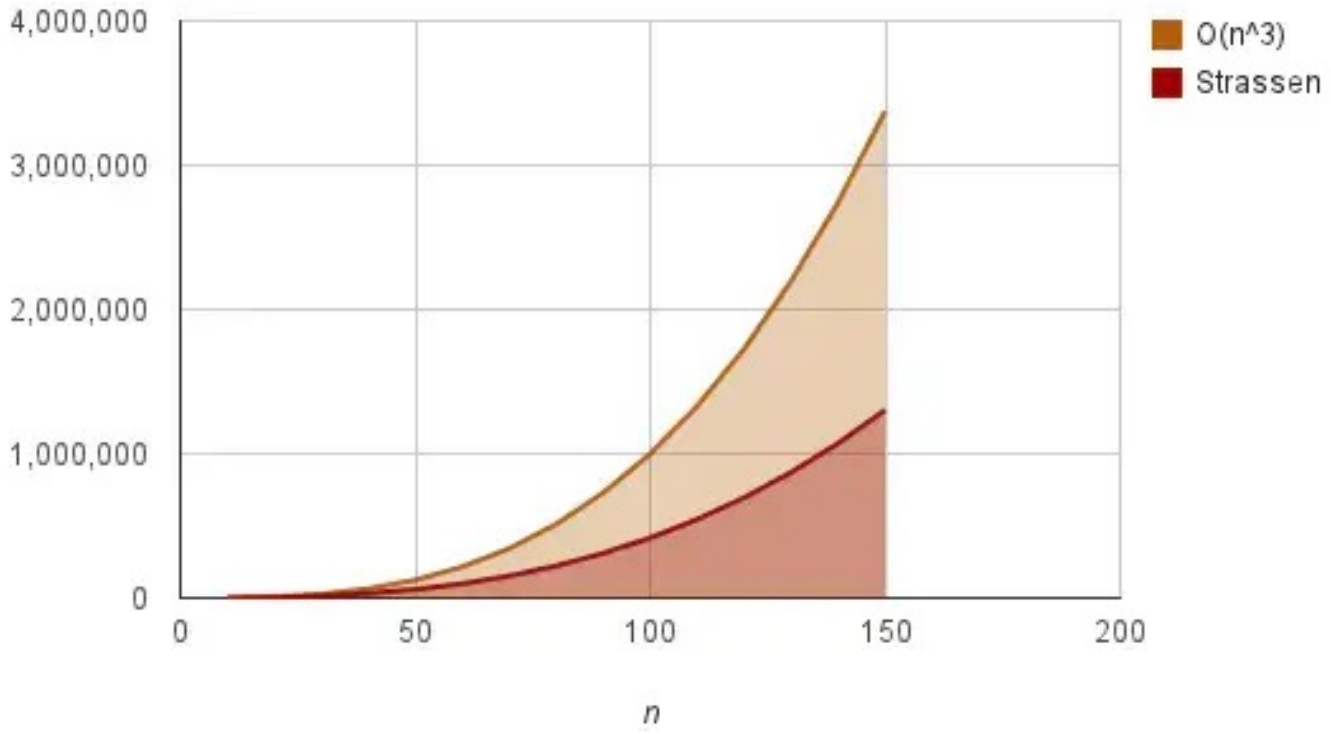
18 matrix additions

$$T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n^{lg7}) \approx \Theta(n^{2.81})$$

7 submatrix multiplication

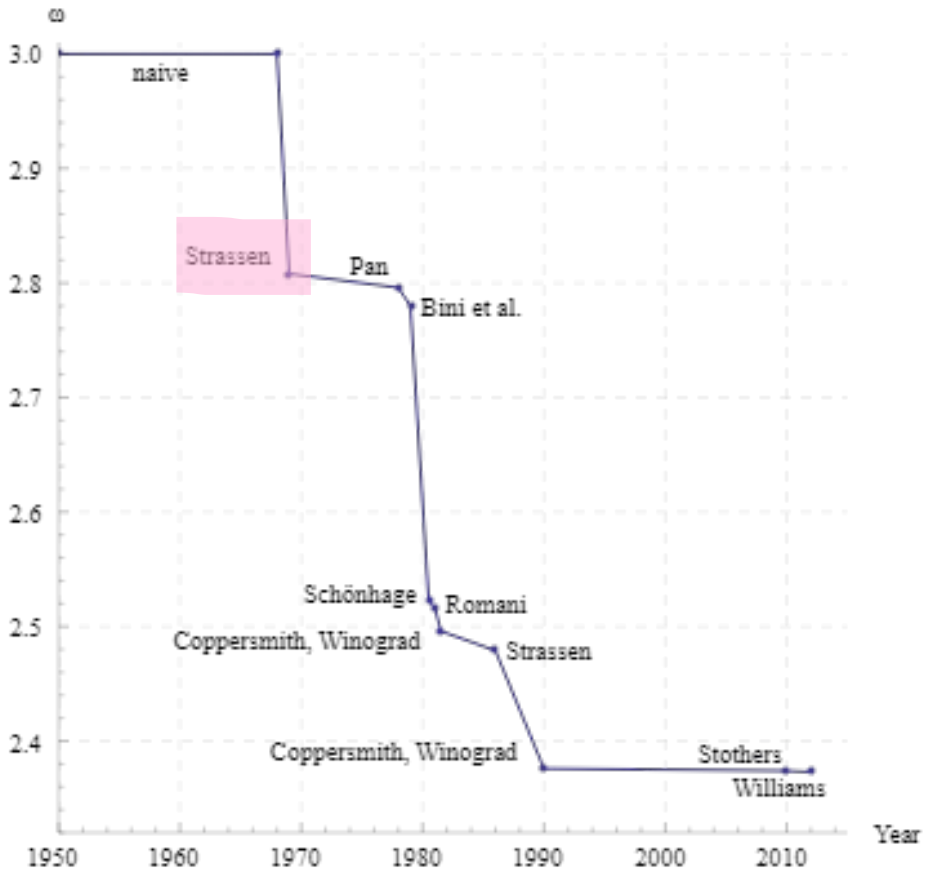
18 matrix additions, with submatrices of size n/2 x n/2.

Strassen algorithm time complexity



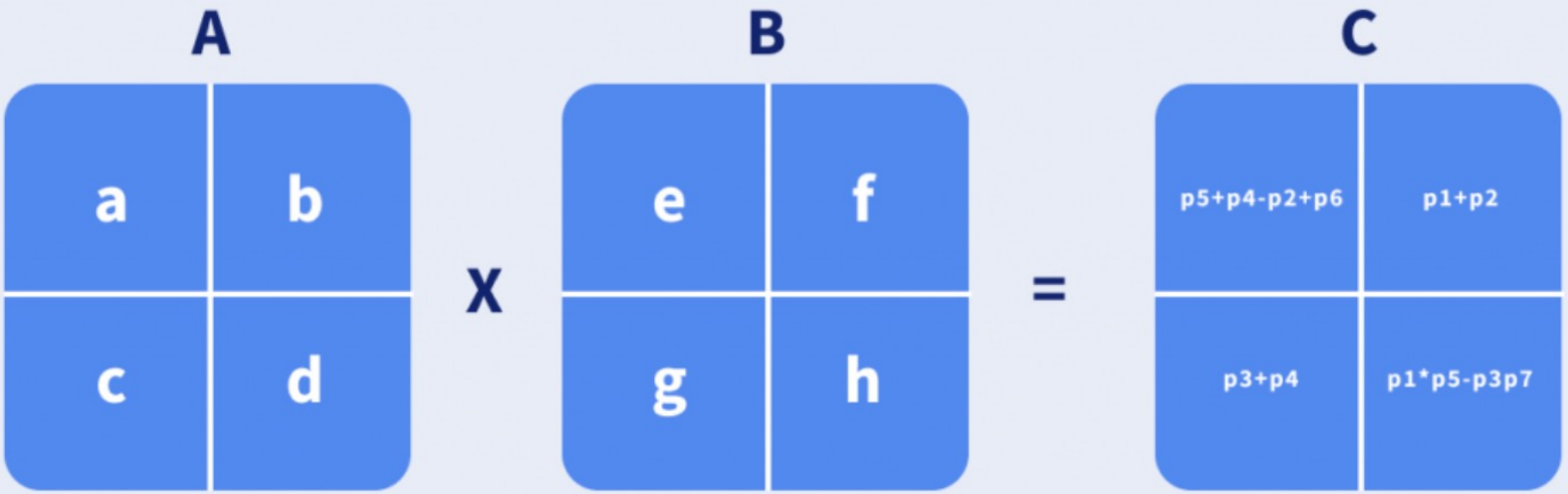
The larger the value of n, the more time the Strassen algorithm saves.

Strassen algorithm time complexity



Strassen algorithm

p1 = a(f-h)
p2 = (a+b)h
p3 = (c+d)e
p4 = d(g-e)
p5 = (a+d)(e+h)
p6 = (b-d)(g+h)
p7 = (a-c)(e+f)



Coppersmith–Winograd algorithm basic idea

The mathematician Carl Friedrich Gauss (1777–1855) once noticed that although the product of two complex numbers

$$(a+bi)(c+di) = ac - bd + (bc + ad)i$$

seems to involve *four* real-number multiplications, it can in fact be done with just *three:* ac, bd, and $(a+b)(c+d)$, since

$$bc + ad = (a+b)(c+d) - ac - bd.$$

- Winograd extended this principle to matrix multiplication.
- By decomposing matrix multiplication into smaller-scale computations, the number of multiplications is further reduced by increasing the number of additions

Coppersmith–Winograd algorithm basic idea

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \rightarrow \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$S1 = A21 + A22$$
$$S2 = S1 - A11$$
$$S3 = A11 - A21$$
$$S4 = A12 - S2$$

$$T1 = B12 - B11$$
$$T2 = B22 - T1$$
$$T3 = B22 - B12$$
$$T4 = T2 - B21$$

$$M1 = A11 * B11$$
$$M2 = A12 * B21$$
$$M3 = S4 * B22$$
$$M4 = A22 * T4$$
$$M5 = S1 * T1$$
$$M6 = S2 * T2$$
$$M7 = S3 * T3$$

7 smaller size multiplication

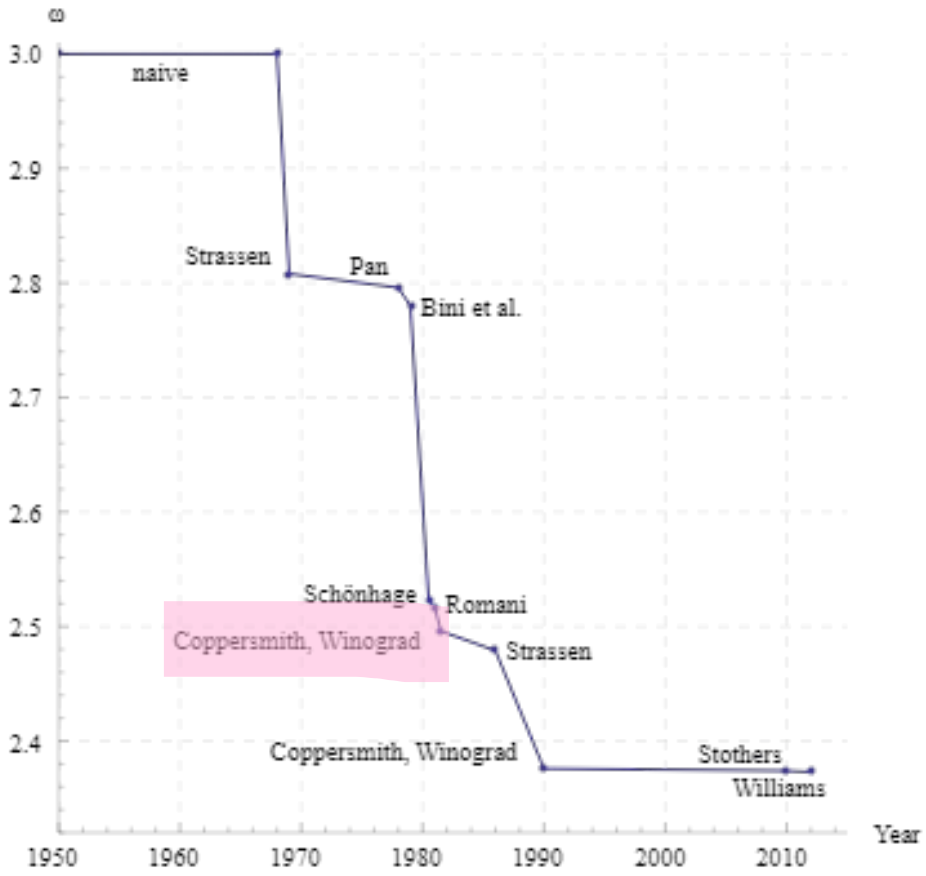
$$U1 = M1 + M2$$
$$U2 = M1 + M6$$
$$U3 = U2 + M7$$
$$U4 = U2 + M5$$
$$U5 = U4 + M3$$
$$U6 = U3 - M4$$
$$U7 = U3 + M5$$

more matrix additions

$$C11 = U1$$
$$C12 = U5$$
$$C21 = U6$$
$$C22 = U7$$

$$O(n^{2.376})$$

Strassen algorithm time complexity



Summary Comparison Table

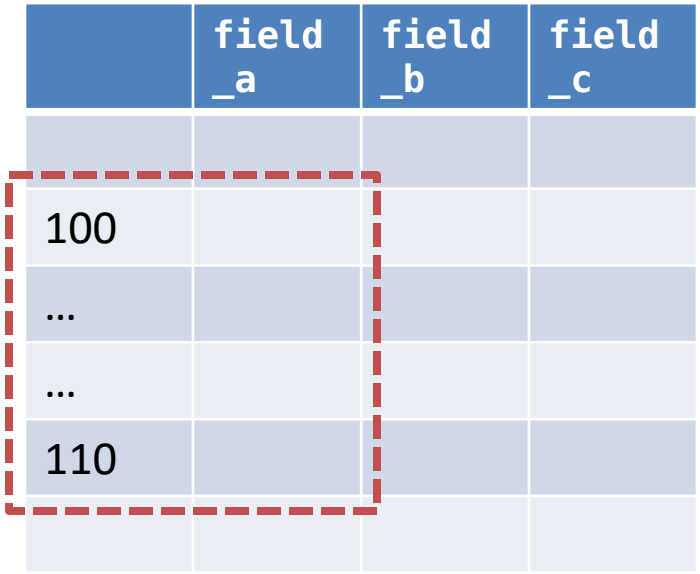
Feature	Strassen Algorithm	Coppersmith–Winograd (CW) Algorithm
Theoretical Complexity	$\approx O(n^{2.81})$	$\approx O(n^{2.37})$
Practicality	Widely used for large-scale matrices	Extremely low (Mainly used for theoretical research)
Constant Factors	Relatively small	Enormously large
Hardware Optimization	Easier to adapt for GPU/Parallel computing	Extremely difficult
Main Contribution	Proved matrix multiplication can be $< O(n^3)$	Pushed the theoretical lower bound of complexity

Example of distributed system: sorting

- Sorting on a single machine, e.g., Database

```
select field_a from table_b order by field_a limit 100, 10;
```

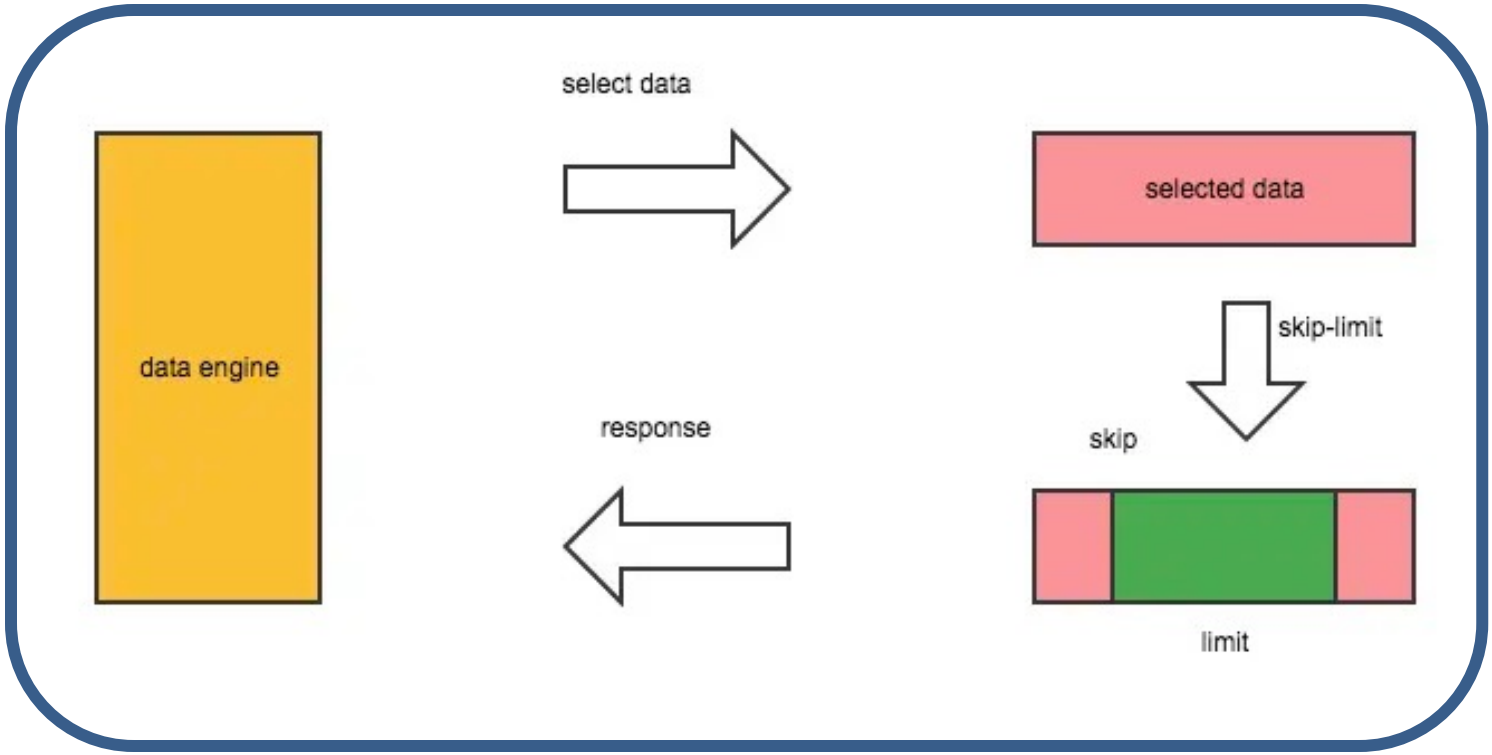
```
db.collection_b
.find()
.sort({"field_a":1})
.skip(100)
.limit(10);
```

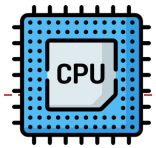


From line 100 to the next 10 lines of data

Example of distributed system: sorting

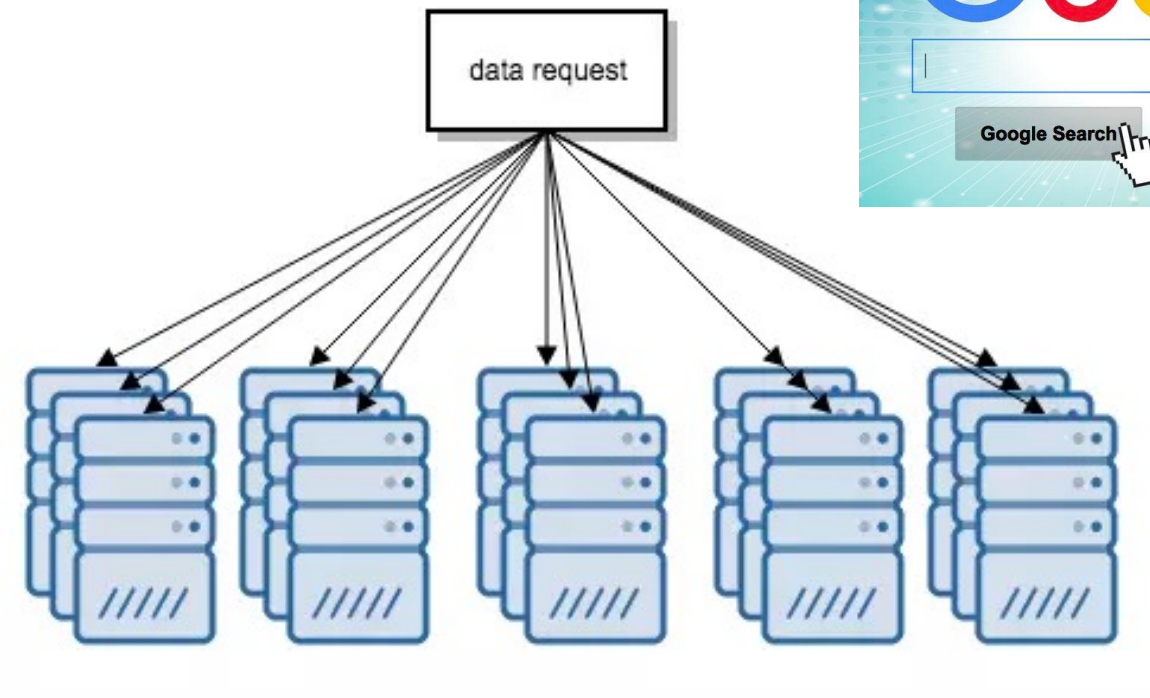
- Workflow on a single node



One server 

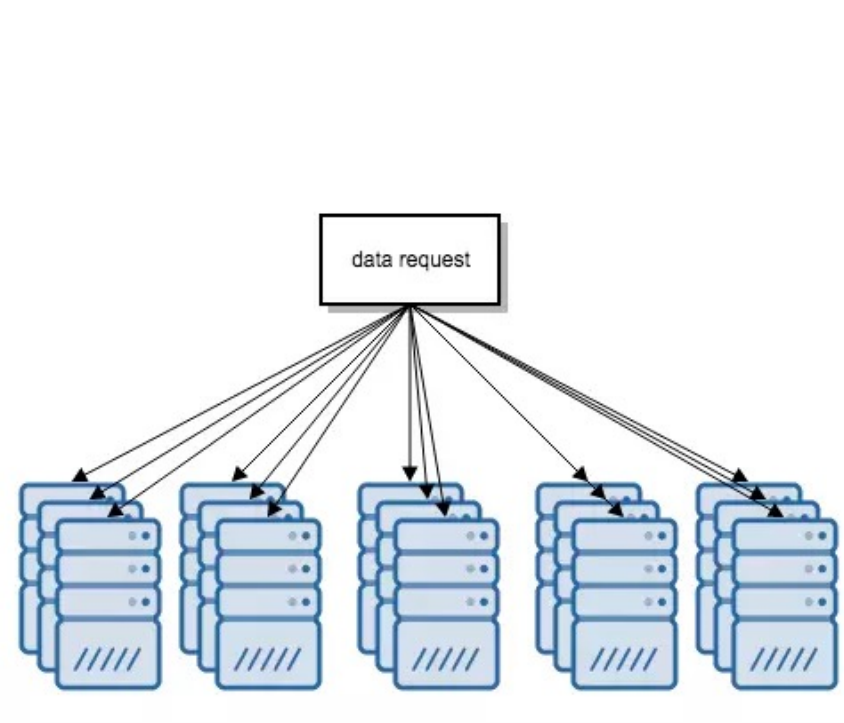
Example of distributed system: sorting

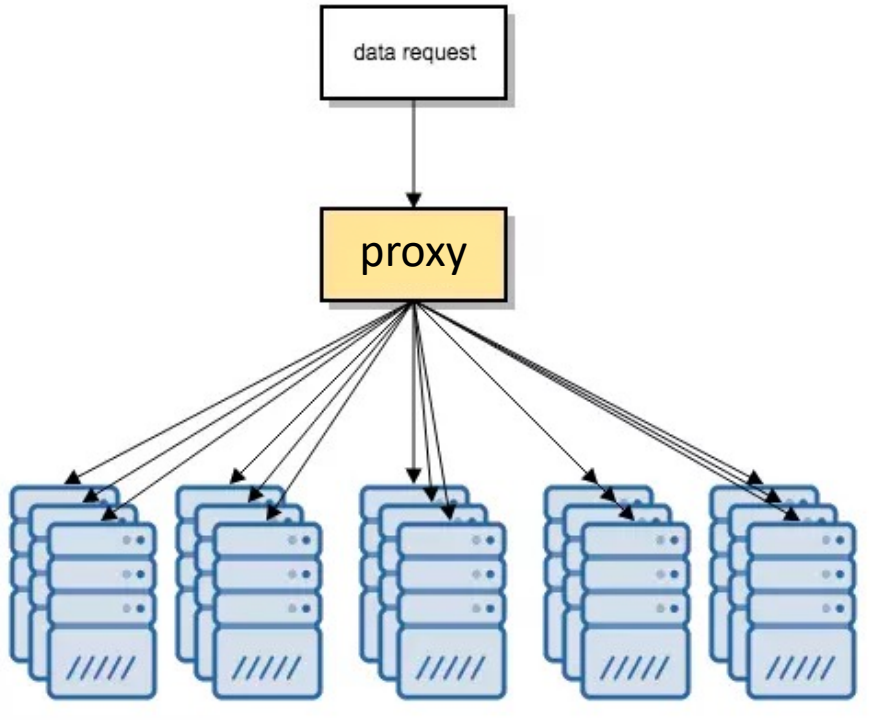
- If the data is too much and single node cannot hold



Example of distributed system: sorting

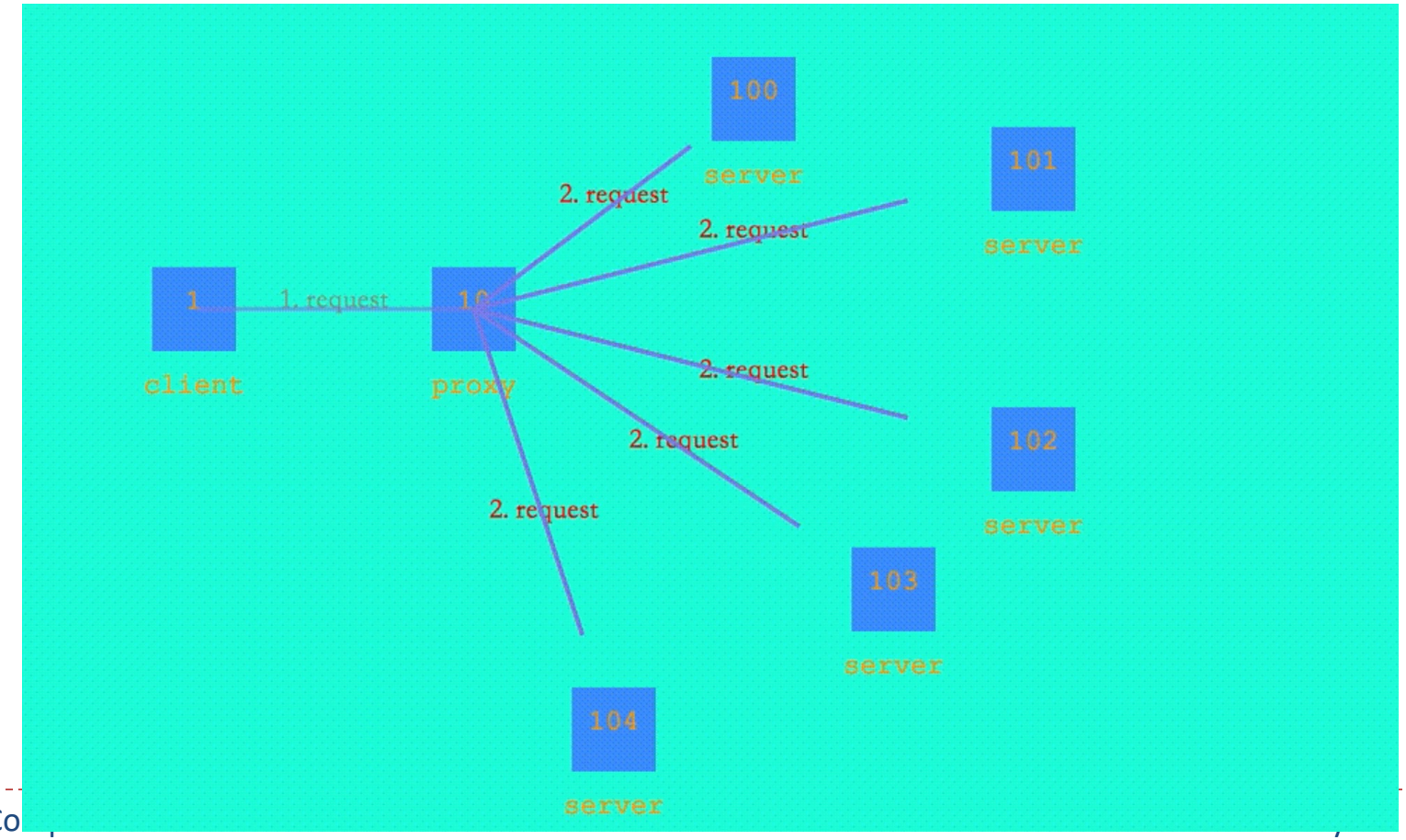
- Choose a node for merge processing





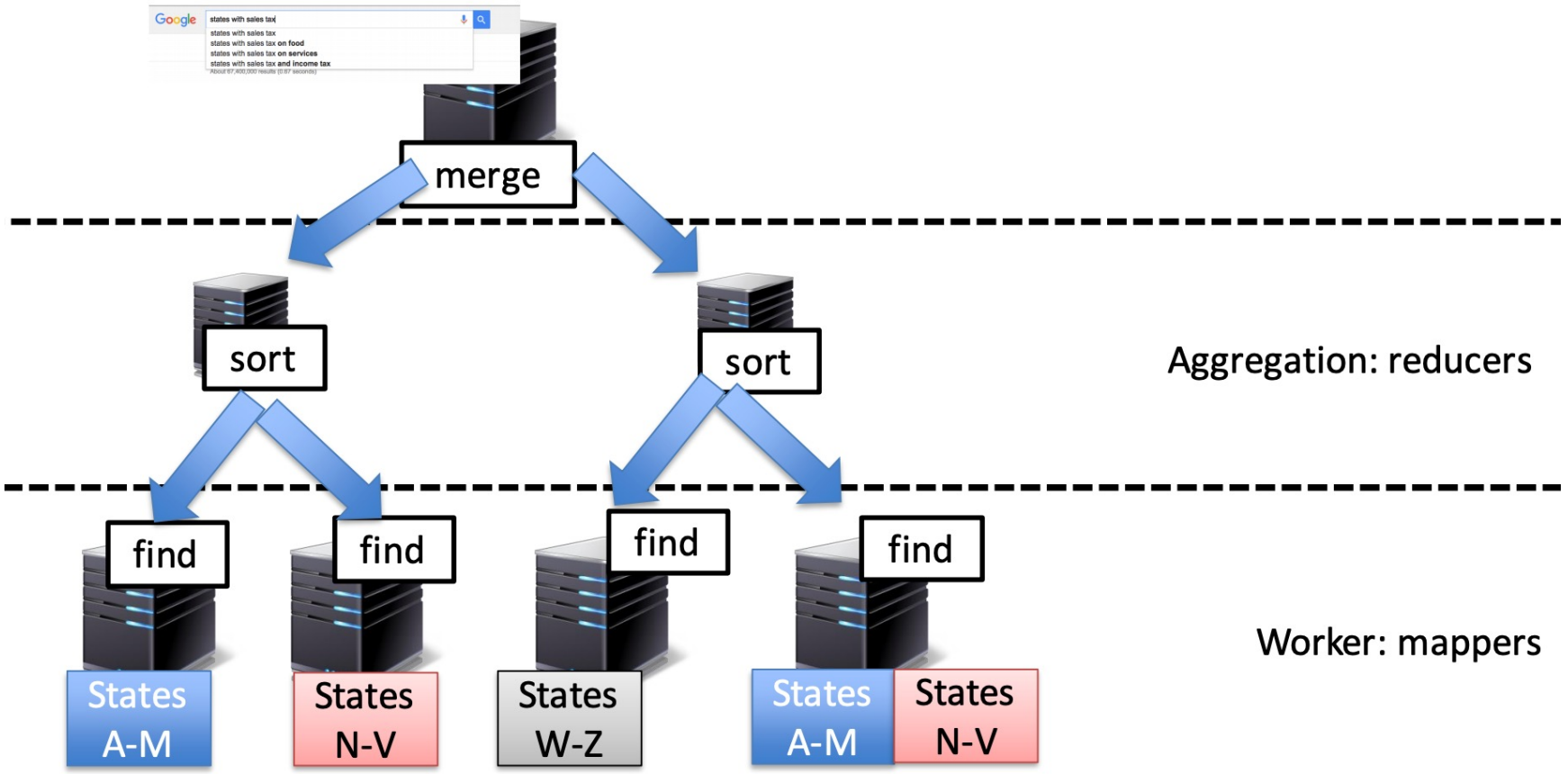
Example of distributed system: sorting

- Workflow



Example of distributed system: sorting

- How Do Request Get Processed in a Data Center



Example of distributed system: sorting

- How Google Search Works
- https://www.youtube.com/watch?v=0eKVizvYSUQ

Conclusion

- Why study HPC & parallel programming?
- What to learn?
- Course structure
- Course policy
- An example of HPC & parallel programming