

Writing Efficient Code

Optimizing Code

Before you consider a parallel algorithm:

- Make sure you have the best sequential algorithm
- Apply compiler optimization
- Identify performance bottlenecks
- Apply sequential optimization techniques

Timing of programs

Exists different ways to do timings:

- . Insert specific timing code
- . Use tools

```
#include <time.h>          // Contains headers for timing

time_t start, stop; // Timing variables
double avg_time = 0;
double cur_time;

for(i=0; i<100; i++) {
    start = clock();           // Get the current time when we start
    scalar_product_sse += ScalarSSE(s1, s2); // Do work
    stop = clock();           // Get the current time when we end

// Time is measured in "ticks", must calculate seconds
    cur_time = ((double) stop-start) / CLOCKS_PER_SEC;
    avg_time += cur_time;      // Accumulate time on all iterations
}

// Will get more accurate time if we perform several runs and take the average
printf("SIMD code used on average %f seconds.\n", avg_time / 100);
}
```

Timing of programs

```
#include <omp.h>

// Using OpenMP specific routines, must compile with -fopenmp

double start = omp_get_wtime();    // Timer start
    linear(&sum1,size,data);
double end = omp_get_wtime();      // Timer ends
double time1 = end - start;        // Calculate time

printf("Timing: %lf \n",time1);
```

Timing of programs

Use tools to time the program

gprof shows time spent in each routine

Compile with -pg

Run program

Creates binary file gmon.out

To get readable data execute:

Look at timing data:

```
gcc -pg myprog.c
```

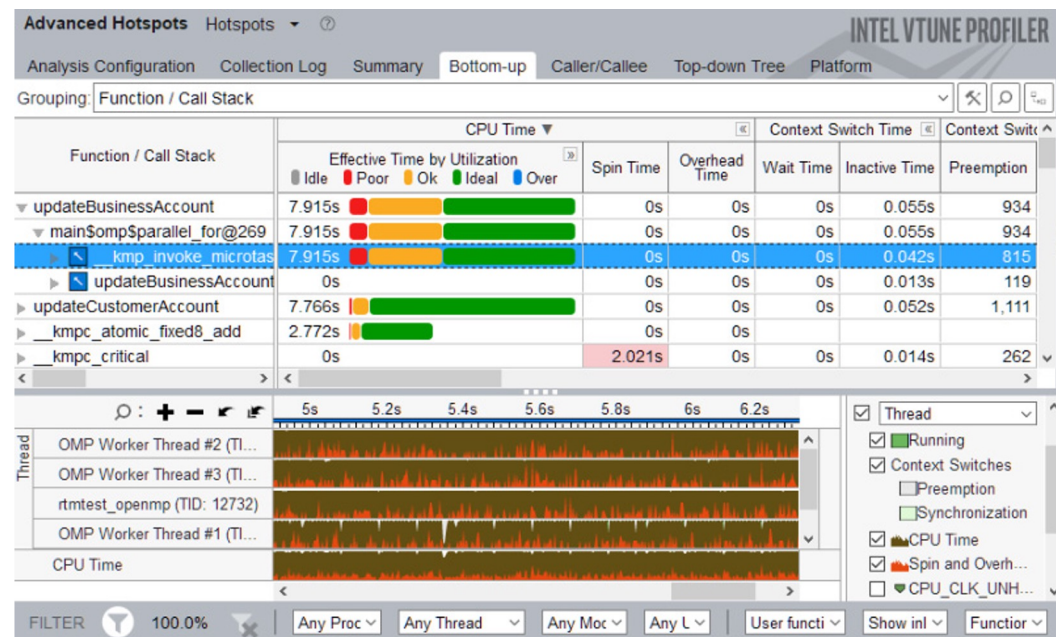
```
./a.out
```

```
ls gmon.out
```

```
gprof a.out gmon.out > myprog_output
```

```
more myprog_output
```

More sophisticated tools
such as Intel Vtune



The Traditional Model

- . A sequential program executes one instruction at a time, according to the logic of the program
- . Time to access any memory location is **uniform** and **short** (comparable to the time it takes to perform one flop)
- . Each flop takes constant time (hopefully one clock tick)
- . **Well suited for algorithm analysis (recall the O -, Ω - and Θ -notations)**

But in the real world, things are slightly different...

How it really works

```
#include <stdio.h>
int main()
{
    double a, b, result = 0;
    printf("Enter first number : ");
    scanf("%lf", &a);

    printf("Enter Second Number : ");
    scanf("%lf", &b);

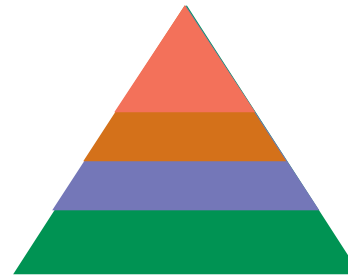
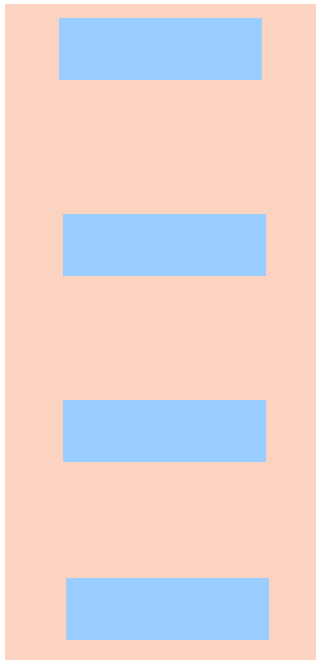
    result = a + b;
    printf("Addition of two numbers is : %lf", result);

    return 0;
}
```

Issues to consider

- Pipelining
- SIMD processing
- Memory hierarchy

CPU



Memory

SIMD

Single Instruction Multiple Data

- Setting: Performing the same operation on multiple data
- Processors have hardware support for performing multiple operations at the same time.
- Compiler (or user) must detect operations that fit

Scalar processing

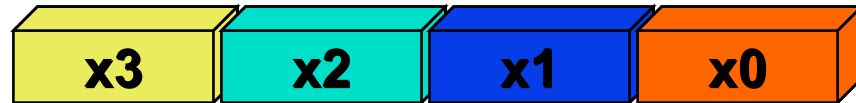


+



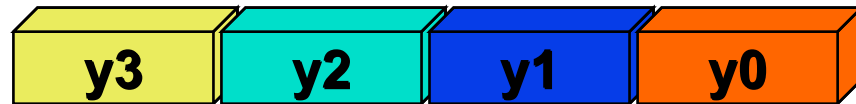
SIMD processing

X

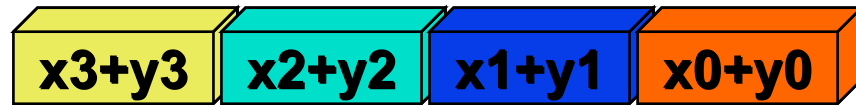


+

Y



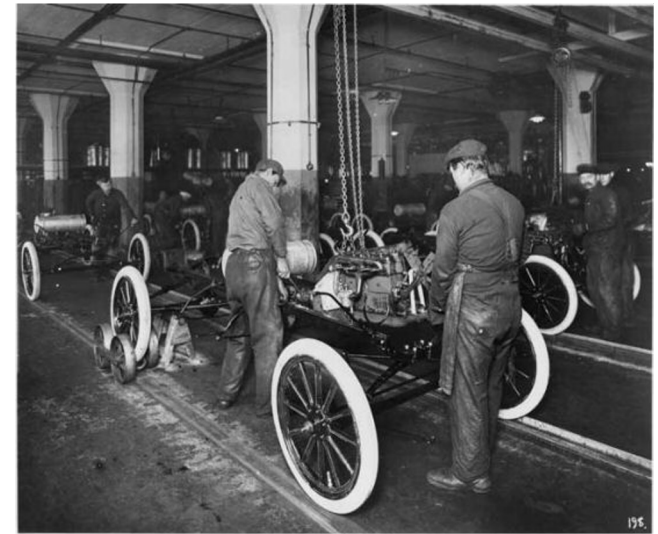
X + Y



Pipelines

Assembly line:

- Each worker performs a separate task
- Can produce one complete unit every time step



Example:

4 tasks, each requiring 4 sequential operations
Would take $4 \times 4 = 16$ time steps.

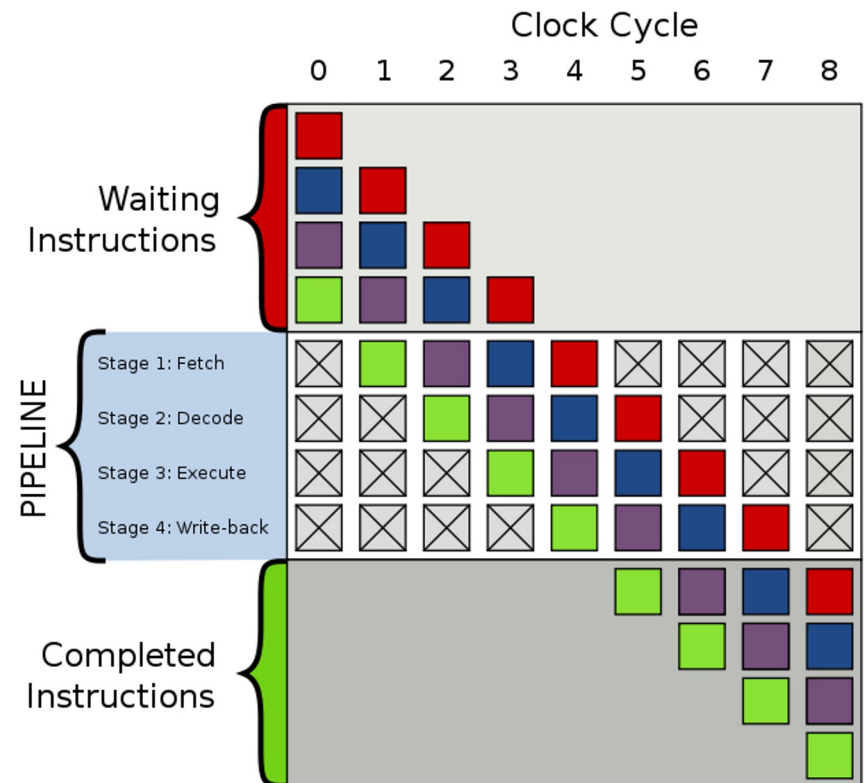
Modern CPUs can *pipeline* similar operations

Using a pipeline of depth 4:

Takes 4 steps to fill the pipeline
Takes 4 more steps to finish
Total 8 steps.

Requires enough similar operations to pay off

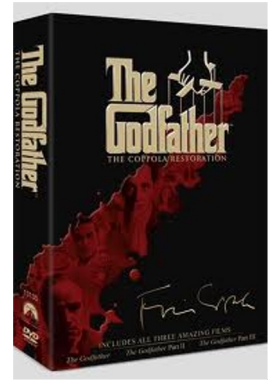
The program must lend itself to using SIMD and pipelines



Latency

You want to buy a DVD. Two options:
Download or buy at the store. Which is faster ?

Assume a DVD holds 4.7Gbyte, 4 discs = 18.8 Gbyte = 150 Gbit.
With a 25 Mbit/s connection this takes $150000/25 \text{ sec} = 6000 \text{ sec} = 1 \text{ hour } 40 \text{ minutes}$
Going to the store takes max 30 minutes...



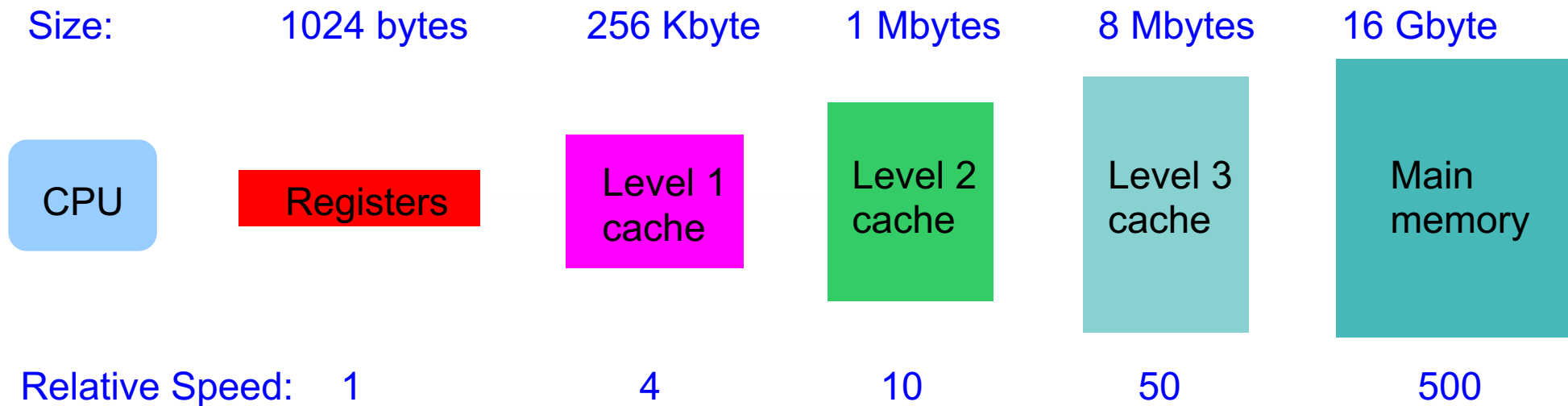
Latency: The time you have to wait before data starts arriving.

- . Going to the store: 30 minutes
- . Downloading: a few milliseconds (10^{-3})

```
[nmidd@ii122078 ~]$ ping brake.ii.uib.no
PING brake.ii.uib.no (129.177.120.119) 56(84) bytes of data.
64 bytes from brake.ii.uib.no (129.177.120.119): icmp_req=1 ttl=63 time=0.313 ms
64 bytes from brake.bccs.uib.no (129.177.120.119): icmp_req=2 ttl=63 time=0.311 ms
```

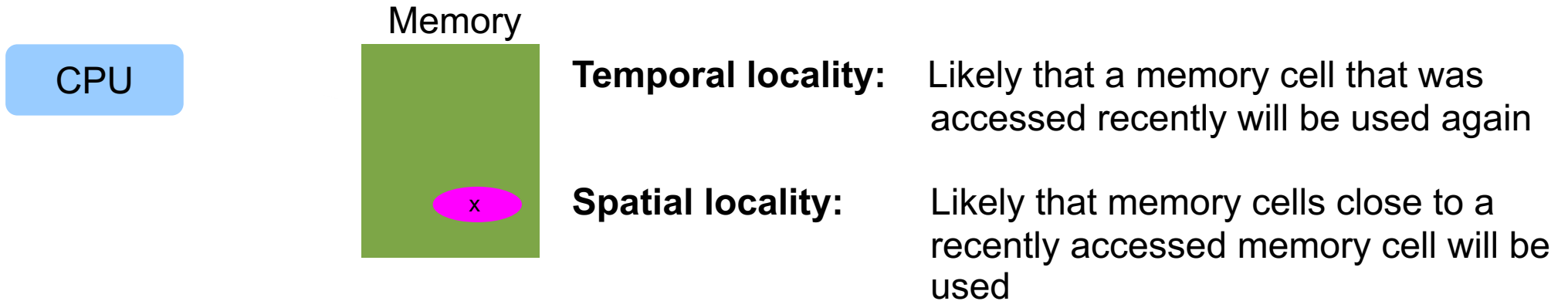
Latency between two computers is $\approx 0.3 \text{ ms} = 0.0003 \text{ seconds} = 300\,000 \times 10^{-9} \text{ seconds}$
A 1GHz computer can perform 300 000 operations in the same time!

Memory Latency



- Larger cache size costs more and increases the latency.
- Bandwidth (bits per second) improves faster than latency.
- Computer looks for data as high up in the cache hierarchy as possible.
- Different strategies for deciding what to keep and what to throw away.

Exploiting Locality



Memory will store what you are likely to need closer to the CPU:

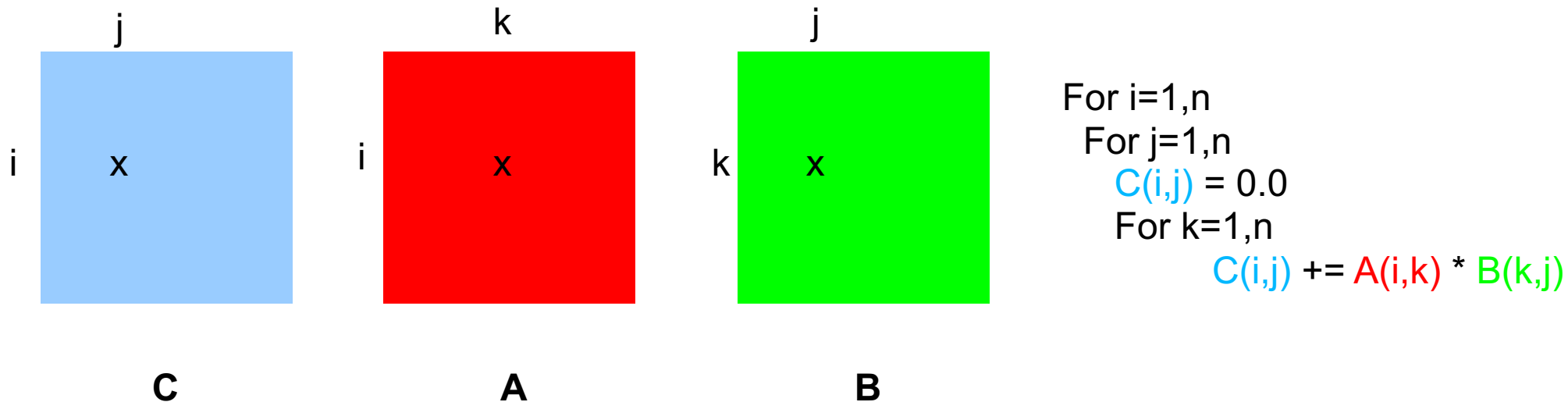
Temporal locality: Will store recently used data points high in the cache hierarchy

Spatial locality: Will simultaneously fetch memory cells close to requested cell

Cache line : Smallest unit of memory that the cache system operates on
Typically 64 bytes

Case study: Matrix multiplication

Compute $C = A \times B$ where A, B , and C are $n \times n$ real valued matrices




Fundamental operation in linear algebra, exists highly optimized codes

Requires $3n^2$ data elements and $n^2(2n - 1)$ flops

Mapping Data to Memory

- A matrix is a 2D data structure, while computer memory addresses are arranged in 1D (one long sequence of memory cells)
- Conventions for matrix layout
 - by column, or “column major”; $A(i,j)$ at $A+i+j*n$
 - by row, or “row major” $A(i,j)$ at $A+i*n+j$

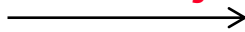
Column major



0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

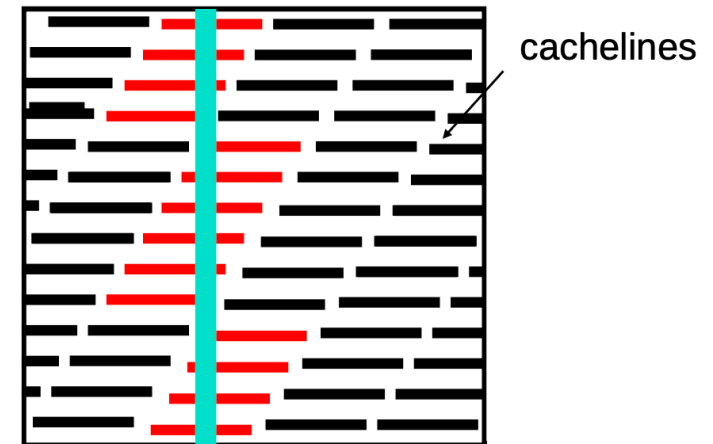
Fortran, Matlab

Row major



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

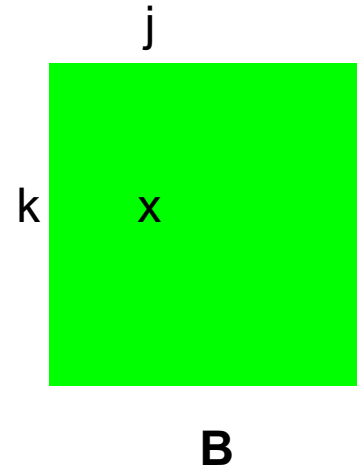
C, Java



Column of matrix is stored in red cache lines

Layout of B matters

Each access to **B** requires a new cache line to be loaded



```
For i=1,n
  For j=1,n
    C(i,j) = 0.0
    For k=1,n
      C(i,j) += A(i,k) * B(k,j)
```

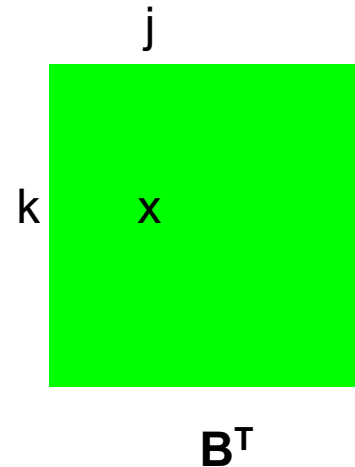
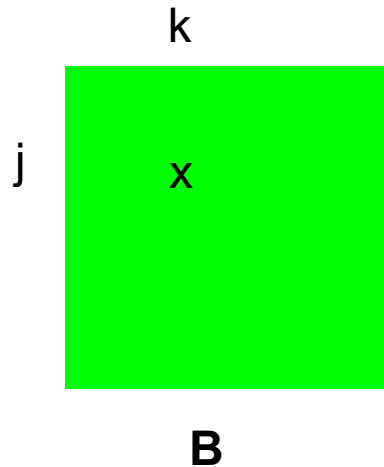


brake.ii.uib.no

Computer with 8 Gflop peak rate

Naive algorithm using $n=1000$ gave 162 Mflop; 2% of theoretical peak...

Transposing B



```

For i = 1,n
  For j = i+1,n
    temp = B(i,j)
    B(i,j) = B(j,i)
    B(j,i) = temp
  
```

```

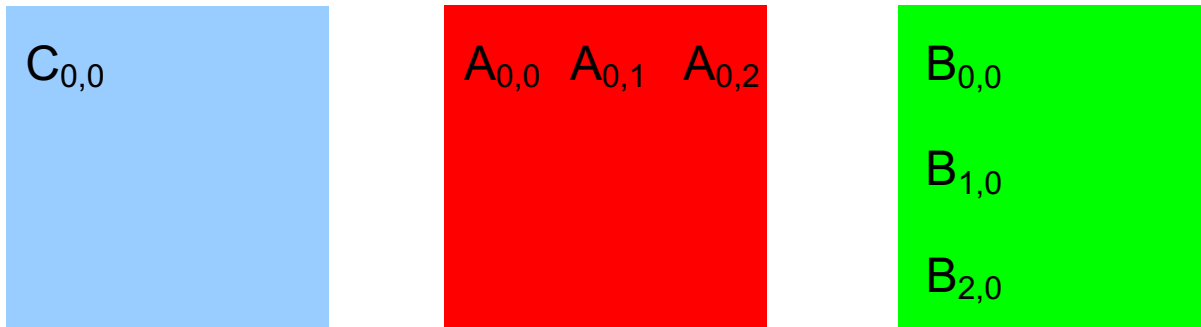
For i=1,n
  For j=1,n
    C(i,j) = 0.0
    For k=1,n
      C(i,j) += A(i,k) * B(k,j)
    
```

```

For i=1,n
  For j=1,n
    C(i,j) = 0.0
    For k=1,n
      C(i,j) += A(i,k) * B(j,k)
    
```

Will use contiguous memory cells in B
 Gave 1189 Mflop; 15% of theoretical peak,

Multiplication by Blocks



$$\text{Then } C_{0,0} = (A_{0,0} * B_{0,0}) + (A_{0,1} * B_{1,0}) + (A_{0,2} * B_{2,0})$$

Advantage:

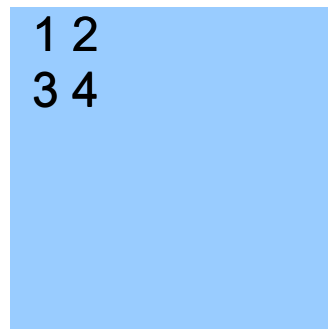
```

For i = 0, nb-1
  For j = 0, nb-1
     $C_{i,j} = 0.0$ 
    For k = 0, nb-1
       $C_{i,j} += A_{i,k} * B_{k,j}$ 
  
```

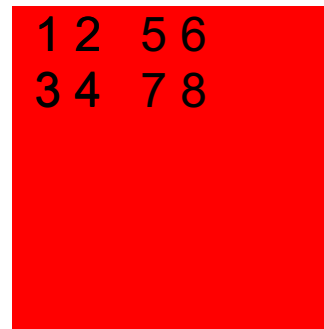
Both $A_{i,k}$ and $B_{k,j}$ can fit in cache
 With $2 \times x$ elements in the cache we can
 perform $x * \sqrt{x}$ operations.



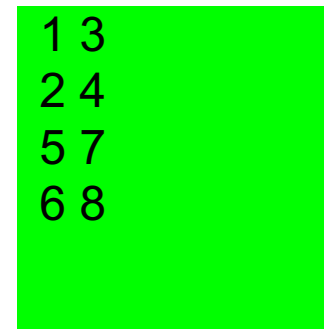
Blocked Storage



C



A



B

- Make sure data elements **within each block** are stored in consecutive memory cells.
- Can use regular “transposed” matrix multiply on each block pair:

$$\begin{array}{c} \text{bs} \\ \text{bs} \end{array} \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} * \begin{array}{|c|c|} \hline 5 & 7 \\ \hline 6 & 8 \\ \hline \end{array}$$

```

For i=1,bs
  For j=1,bs
    For k=1,bs
      C(i,j) += A(i,k) * B(j,k)
    
```

A, B, and C now points to the start of each block

- Must know cache size in advance.
- Other schemes try to be cache friendly independently of the cache size.

Storing Graphs

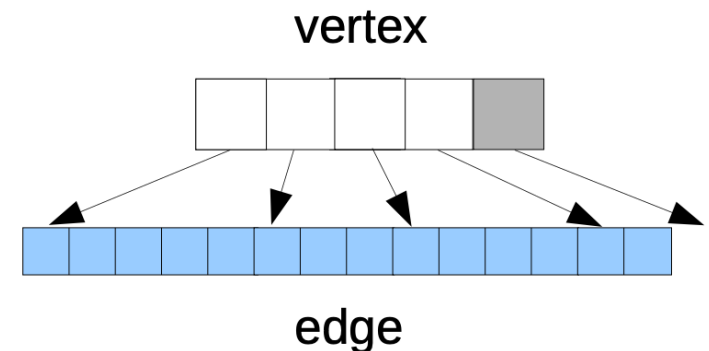
Individual lists:

- One vertex-indexed array
- Each entry points to a neighbor list
- Neighbor lists might not be consecutive in memory!



Compressed lists:

- Edges are consecutive
- Use extra dummy pointer at the end
Points to last element + 1
- Use integer values in both arrays



Traverse the graph:

```
for v = 0, n-1
  for e = vertex[v], vertex[v+1]-1
    do something with edge[e]
```

Computing Connected Components

Algorithms courses

- Traditionally taught to use either DFS or BFS
- Linear running time
- Will move around in memory depending on the structure of the graph and how it is stored.
- Not good in terms of memory use!

Disjoint set data structure

- (partial) components are represented by trees
- Find(v) returns root of component containing vertex v
- Union(u, v) merges trees containing vertices u and v
- Running time $O(m\alpha(m, n)) \approx O(m)$ Almost linear.
- Traverses the graph according to how it is stored!

Union-Find(G)

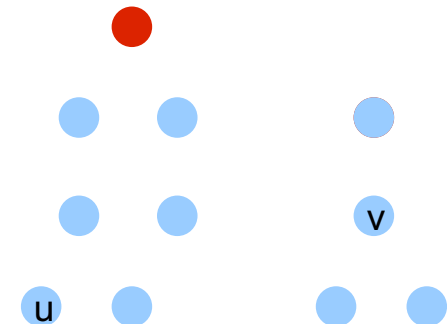
for each vertex v

$$p(v) = v$$

for each edge (u, v)

if Find(u) \neq Find(v)

Union(u, v)



Experimental Results

From [Manne and Patwary 2009]

- Comparing DFS with Disjoint Set algorithm (ZZ)
- Real world graphs from the Florida sparse matrix collection

Name	$ V $	$ E $	Max Deg	Avg Deg	DFS	ZZ
m_t1	97578	4827996	236	98.95	0.12	0.06
cranksg2	63838	7042510	3422	220.64	0.15	0.03
inline_1	503712	18156315	842	72.09	0.57	0.26
ldoor	952203	22785136	76	47.86	0.71	0.47
af_shell10	1508065	25582130	34	33.93	1.04	0.37
boneS10	914898	27276762	80	59.63	0.86	0.38
bone010	986703	35339811	80	71.63	1.05	0.47
audi	943695	38354076	344	81.28	1.20	0.33
spal_004	321696	45429789	6140	282.44	1.33	0.66
rmat1	377823	30696982	8109	162.49	2.07	1.34
rmat2	504817	40870608	10468	161.92	2.71	1.81

Things to remember

Before going parallel, always optimize single CPU performance first

- Access data elements as sequentially as possible
- Do as much as possible with the data in cache
 - Reuse data elements
- Check for compiler options that might help you
- To use full optimization, compile with -O3
- If possible, use optimized software
 - Linear algebra software
 - Graph algorithms

For more on sequential performance optimization, see the online book:

Algorithms for Modern Hardware

<https://en.algorithmica.org/hpc/>

Contains specific advice and examples