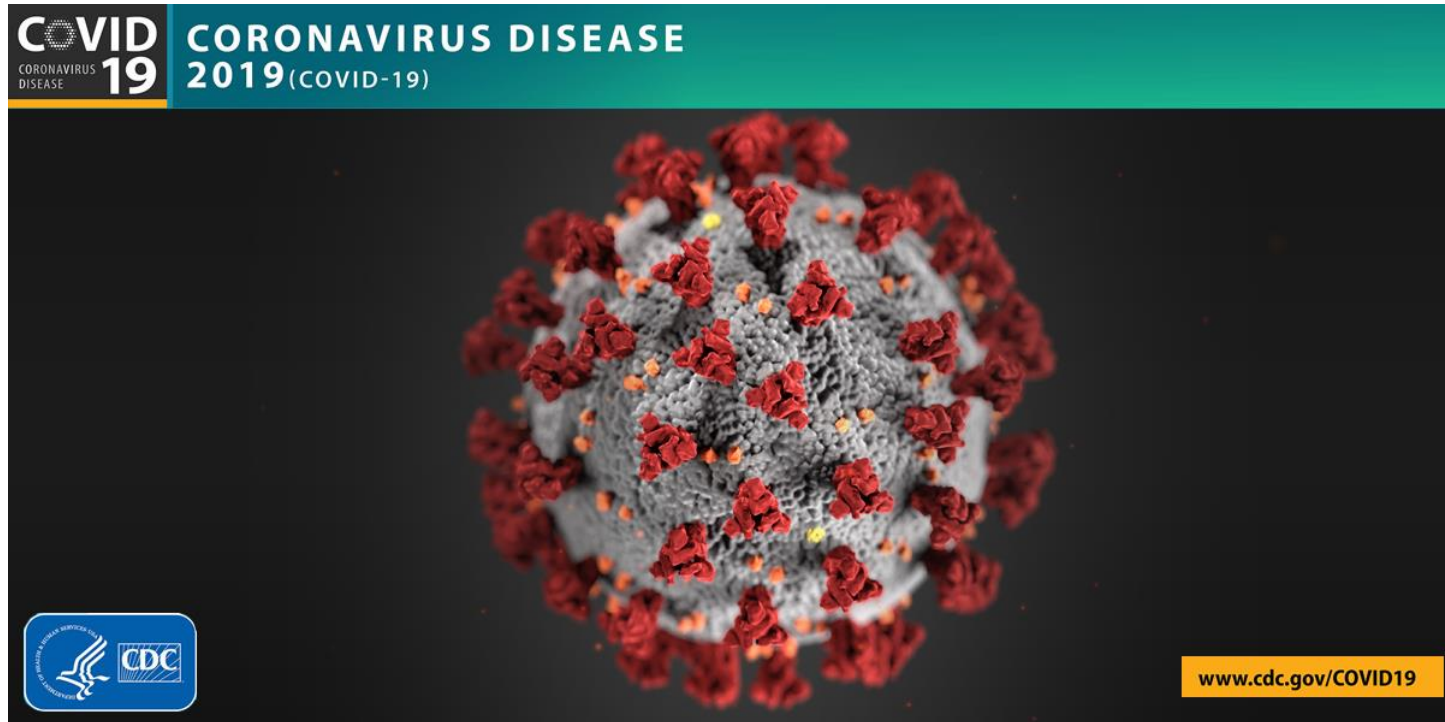


# Paralelní a distribuované výpočty (B4B36PDV)



- Přihlašujte se pod Google účtem
- Pokud je to možné, používejte sluchátka
- Pokud nemluvíte, vypněte si mikrofon

# Paralelní a distribuované výpočty (B4B36PDV)

**Branislav Bošanský, Michal Jakob**

[bosansky@fel.cvut.cz](mailto:bosansky@fel.cvut.cz)

Artificial Intelligence Center  
Department of Computer Science  
Faculty of Electrical Engineering  
Czech Technical University in Prague

# Dnešní přednáška

Motivace



# Dnešní přednáška

## Techniky paralelizace 3

Chci paralelizovat maticový algoritmus



Jak na to?

# Maticové operace

## Násobení matice vektorem

a11	a12	a13	a14	a15		x1	y1
a21	...					x2	y2
...					x	x3	y3
						x4	y4
				a55		x5	y5

$$y_1 = \sum_{\{i=1,\dots,5\}} a_{1i} \cdot x_i$$

# Maticové operace

## Násobení matice vektorem

- Jak paralelizovat?

a11	a12	a13	a14	a15		x1	y1
a21	...					x2	y2
...					x	x3	y3
						x4	y4
				a55		x5	y5

$$y_1 = \sum_{\{i=1,\dots,5\}} a_{1i} \cdot x_i$$

# Maticové operace

## Násobení matice vektorem

- Zřejmá paralelizace – výpočet každé složky  $y$  je nezávislý a všechny mohou být spočteny paralelně

P1	→	a11	a12	a13	a14	a15		x1	y1
P2	→	a21	...					x2	y2
		...						x3	y3
...								x4	y4
						a55		x5	y5

X =

$$y_1 = \sum_{\{i=1,\dots,5\}} a_{1i} \cdot x_i$$

# Maticové operace

## Násobení matice vektorem

- Zřejmá paralelizace – výpočet každé složky  $y$  je nezávislý a všechny mohou být spočteny paralelně

```
void multiply(std::vector<int>& A, std::vector<int>& x, std::vector<int>& y) {
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \
    std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>())) \
    initializer(omp_priv = omp_orig)

#pragma omp parallel for num_threads(thread_count) reduction(vec_int_plus : y)
    for (int i=0; i<ROWS; i++) {
        for (int j=0; j<COLS; j++) {
            y[i] += A[i * COLS + j]*x[j];
        }
    }
}
```



# Maticové operace

## Násobení matice vektorem

- Zřejmá paralelizace – výpočet každé složky  $y$  je nezávislý a všechny mohou být spočteny paralelně

```
void multiply(std::vector<int>& A, std::vector<int>& x, std::vector<int>& y) {
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \
    std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>())) \
    initializer(omp_priv = omp_orig)

#pragma omp parallel for num_threads(thread_count) reduction(vec_int_plus : y)
    for (int i=0; i<ROWS; i++) {
        for (int j=0; j<COLS; j++) {
            y[i] += A[i * COLS + j]*x[j];
        }
    }
}
```

- Co můžeme zlepšit?

# Maticové operace

## Násobení matice vektorem

- Zřejmá paralelizace – výpočet každé složky  $y$  je nezávislý a všechny mohou být spočteny paralelně

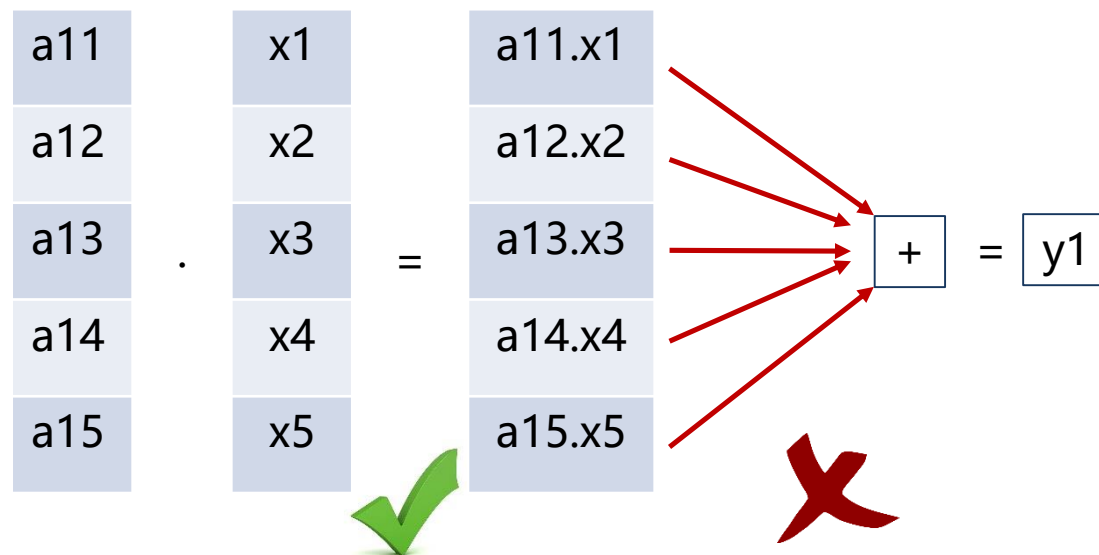
```
void multiply(std::vector<int>& A, std::vector<int>& x, std::vector<int>& y) {  
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \  
    std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>())) \  
    initializer(omp_priv = omp_orig)  
  
#pragma omp parallel for num_threads(thread_count) reduction(vec_int_plus : y)  
    for (int i=0; i<ROWS; i++) {  
        for (int j=0; j<COLS; j++) {  
            y[i] += A[i * COLS + j]*x[j];  
        }  
    }  
}
```

- Co můžeme zlepšit?
- Lze využít vektorizaci?

# Maticové operace

## Násobení matice vektorem

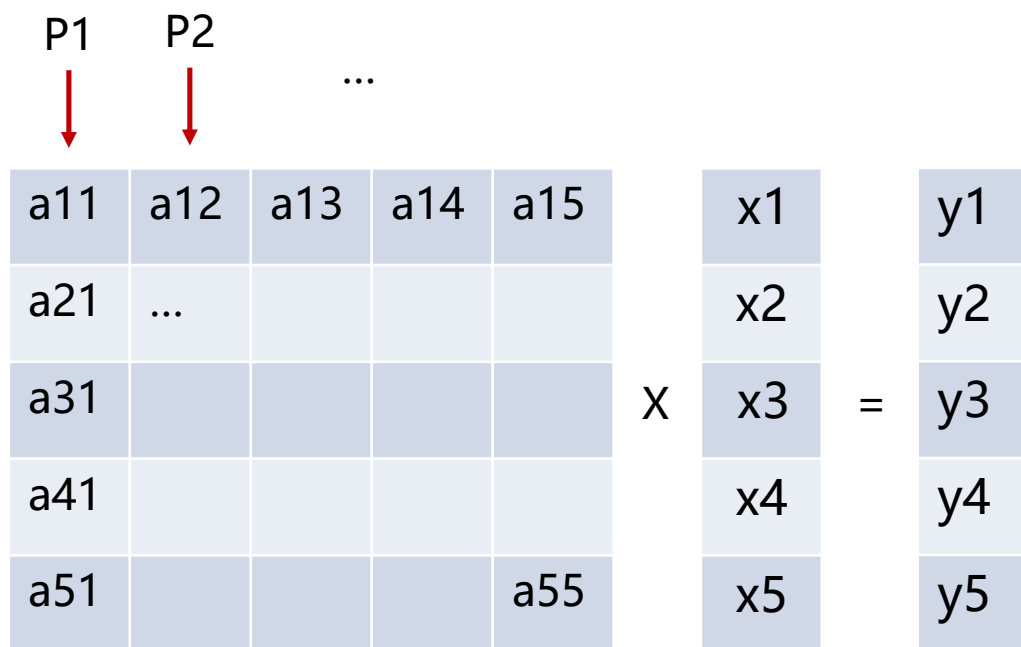
```
void multiply(std::vector<int>& A, std::vector<int>& x, std::vector<int>& y) {  
    #pragma omp declare reduction(vec_int_plus : std::vector<int> : \  
        std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>())) \  
        initializer(omp_priv = omp_orig)  
  
    #pragma omp parallel for num_threads(thread_count) reduction(vec_int_plus : y)  
    for (int i=0; i<ROWS; i++) {  
        for (int j=0; j<COLS; j++) {  
            y[i] += A[i * COLS + j]*x[j];  
        }  
    }  
}
```



# Maticové operace

## Násobení matice vektorem

- Co když budeme násobit po sloupcích?





$$z_{ij} = a_{ij} \cdot x_j$$


$$y_i = \sum_{\{j=1, \dots, 5\}} z_{ij}$$

# Maticové operace

## Násobení matice vektorem

$$\begin{array}{c} a_{11} \\ a_{21} \\ a_{31} \\ a_{41} \\ a_{51} \end{array} \cdot \begin{array}{c} x_1 \\ x_1 \\ x_1 \\ x_1 \\ x_1 \end{array} = \begin{array}{c} a_{11}.x_1 \\ a_{21}.x_1 \\ a_{31}.x_1 \\ a_{41}.x_1 \\ a_{51}.x_1 \end{array} \quad z_1$$


$$\begin{array}{c} a_{12} \\ a_{22} \\ a_{32} \\ a_{42} \\ a_{52} \end{array} \cdot \begin{array}{c} x_2 \\ x_2 \\ x_2 \\ x_2 \\ x_2 \end{array} = \begin{array}{c} a_{12}.x_2 \\ a_{22}.x_2 \\ a_{32}.x_2 \\ a_{42}.x_2 \\ a_{52}.x_2 \end{array} \quad z_2$$


$$\begin{array}{c} a_{11}.x_1 \\ a_{21}.x_1 \\ a_{31}.x_1 \\ a_{41}.x_1 \\ a_{51}.x_1 \end{array} + \begin{array}{c} a_{12}.x_2 \\ a_{22}.x_2 \\ a_{32}.x_2 \\ a_{42}.x_2 \\ a_{52}.x_2 \end{array} + \dots + = \begin{array}{c} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{array}$$


# Maticové operace

## Násobení matice vektorem

- Bude to fungovat?

# Maticové operace

## Násobení matice vektorem

- Bude to fungovat?
  - Operace sice mohou být vektorizovány, nicméně přístup není vhodný pro cache (mnoho dotazů)
  - Můžeme data v matici uspořádat po sloupcích

a11	a12	a13	a14	a15
a21	...			
a31				
a41				
a51				a55



a11	a21	a31	a41	a51	...		
-----	-----	-----	-----	-----	-----	--	--

# Maticové operace

## Násobení matice vektorem

- Bude to teď fungovat?

```
...
// data has to be ordered by columns in memory
for (int i = 0; i < COLS; i++) {
    x[i] = rand() % 1000;
    for (int j = 0; j < ROWS; j++) {
        A[i * ROWS + j] = rand() % 1000;
    }
}
...

void multiply_column(std::vector<int> &A, std::vector<int> &x, std::vector<int> &y) {
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \
    std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>())) \
    initializer(omp_priv = omp_orig)

#pragma omp parallel for num_threads(thread_count) reduction(vec_int_plus : y)
    for (int i = 0; i < COLS; i++) {
        for (int j = 0; j < ROWS; j++) {
            y[j] += A[i * ROWS + j]*x[i];
        }
    }
}
```



# Maticové operace

## Násobení matice vektorem

- Lze dále zefektivnit původní přístup?
  - Vzpomeňte si na falsesharing ...

```
void multiply(std::vector<int>& A, std::vector<int>& x, std::vector<int>& y) {
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \
    std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>())) \
    initializer(omp_priv = omp_orig)

#pragma omp parallel for num_threads(thread_count) reduction(vec_int_plus : y)
    for (int i=0; i<ROWS; i++) {
        for (int j=0; j<COLS; j++) {
            y[i] += A[i * COLS + j]*x[j];
        }
    }
}
```

# Maticové operace

## Násobení matice vektorem

- Lze dále zefektivnit původní přístup?
  - Vzpomeňte si na false sharing ...

```
void multiply(std::vector<int>& A, std::vector<int>& x, std::vector<int>& y) {  
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \  
    std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>())) \  
    initializer(omp_priv = omp_orig)  
  
#pragma omp parallel for num_threads(thread_count) reduction(vec_int_plus : y)  
    for (int i=0; i<ROWS; i++) {  
        for (int j=0; j<COLS; j++) {  
            y[i] += A[i * COLS + j]*x[j];  
        }  
    }  
}
```

- Nahradíme pole lokální proměnnou

# Maticové operace

## Násobení matice vektorem

- Lokální proměnná

```
void multiply(std::vector<int> &A, std::vector<int> &x, std::vector<int> &y) {  
    #pragma omp declare reduction(vec_int_plus : std::vector<int> : \  
        std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>())) \  
        initializer(omp_priv = omp_orig)  
  
    int tmp;  
    #pragma omp parallel for num_threads(thread_count) reduction(vec_int_plus : y)  
    for (int i=0; i<ROWS; i++) {  
        tmp = 0;  
        for (int j=0; j<COLS; j++) {  
            tmp += A[i * COLS + j]*x[j];  
        }  
        y[i] += tmp;  
    }  
}
```

# Maticové operace

## Násobení dvou matic

a11	a12	a13		b11	b12	b13		c11	c12	c13
a21	...		x	b21	...		=	c21	...	
...				...				...		

$$c_{ij} = \sum_{\{k=1, \dots, n\}} a_{ik} \cdot b_{kj}$$

# Maticové operace

## Násobení dvou matic

a11	a12	a13		b11	b12	b13		c11	c12	c13
a21	...		x	b21	...		=	c21	...	
...				...				...		

Výpočet prvků  $c$  je opět nezávislý a lze paralelizovat

$$c_{ij} = \sum_{\{k=1, \dots, n\}} a_{ik} \cdot b_{kj}$$

Nevýhody?

Velké množství úloh, malé úlohy

# Maticové operace

## Násobení dvou matic

a11	a12	a13		b11	b12	b13		c11	c12	c13
a21	...		×	b21	...		=	c21	...	
...				...				...		

Můžeme zvětšit úkoly spojením několika řádků

```
void multiply(std::vector<int>& A, std::vector<int>& B, std::vector<int>& C) {
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \
    std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>())) \
    initializer(omp_priv = omp_orig)
    int tmp;
#pragma omp parallel for collapse(2) num_threads(thread_count) reduction(vec_int_plus : C)
    for (int i=0; i<ROWS; i++) {
        for (int j=0; j<COLS; j++) {
            tmp = 0;
            for (int k=0; k<ROWS; k++) {
                tmp += A[i * COLS + k] * B[k * COLS + j];
            }
            C[i * COLS + j] += tmp;
        }
    }
}
```

# Maticové operace

## Násobení dvou matic

- Rozdělení na bloky
- 1 úkol odpovídá  
částečnému výsledku  
submatice

# Maticové operace

## Násobení dvou matic

- Rozdělení na bloky
- 1 úkol odpovídá  
částečnému výsledku  
submatice ( např.  
 $c_{11}, c_{12}, c_{21}, c_{22}$ )

b11	b12	b13	b14
b21	...		
...			

a11	a12	a13	a14
a21	...		
...			

c11	c12	c13	c14
c21	...		
...			

$$c_{11} += a_{11} \cdot b_{11} + a_{12} \cdot b_{21}$$

$$c_{12} += a_{11} \cdot b_{12} + a_{12} \cdot b_{22}$$

...



# Maticové operace

## Násobení dvou matic

```
void multiply_blocks(std::vector<int>& A, std::vector<int>& B, std::vector<int>& C) {
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \
    std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>())) \
    initializer(omp_priv = omp_orig)

    const int ROWS_IN_BLOCK = 10;
    const int BLOCKS_IN_ROW = ROWS/ROWS_IN_BLOCK;
    int tmp;

#pragma omp parallel for collapse(2) num_threads(thread_count) reduction(vec_int_plus : C) private(tmp)
    for (int br1=0; br1<BLOCKS_IN_ROW; br1++) {
        for (int bb=0; bb<BLOCKS_IN_ROW; bb++) {
            for (int bc2 = 0; bc2 < BLOCKS_IN_ROW; bc2++) {
                for (int r = br1 * ROWS_IN_BLOCK; r < (br1 + 1) * ROWS_IN_BLOCK; r++) {
                    for (int c = bc2 * ROWS_IN_BLOCK; c < (bc2 + 1) * ROWS_IN_BLOCK; c++) {
                        tmp = 0;
                        for (int k = 0; k < ROWS_IN_BLOCK; k++) {
                            tmp += A[r * COLS + (k + bb*ROWS_IN_BLOCK)] * B[(bb*ROWS_IN_BLOCK + k) * COLS + c];
                        }
                        C[r * COLS + c] += tmp;
                    }
                }
            }
        }
    }
}
```

# Maticové operace

## Násobení dvou matic

$$\begin{array}{|c|c|c|} \hline a_{11} & a_{12} & a_{13} \\ \hline a_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array} \times \begin{array}{|c|c|c|} \hline b_{11} & b_{12} & b_{13} \\ \hline b_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline c_{11} & c_{12} & c_{13} \\ \hline c_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array}$$

- A co dál? Lze využít vektorizaci?
- Můžeme najednou spočítat vektor částečných hodnot?

# Maticové operace

## Násobení dvou matic

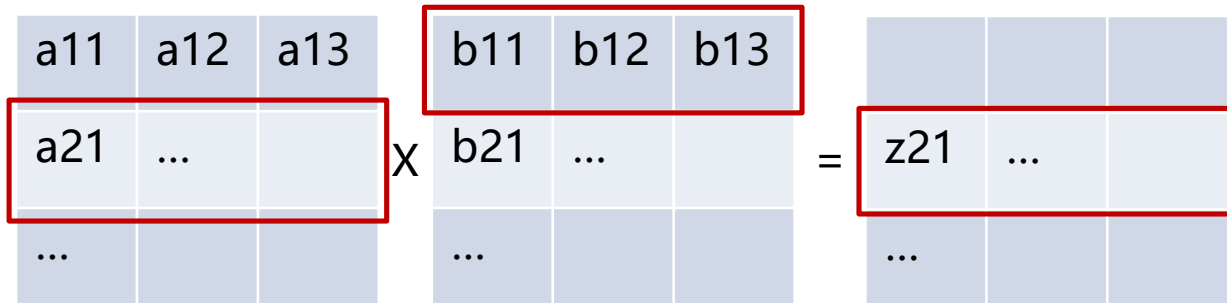
a11	a12	a13		b11	b12	b13		c11	c12	c13
a21	...		$\times$	b21	...		=	c21	...	
...				...				...		

- A co dál? Lze využít vektorizaci?
- Můžeme najednou spočítat vektor částečných hodnot?
- Co když budeme násobit řádek  $i$  matice A a řádek  $j$  matice B?

a11	a12	a13		b11	b12	b13				
a21	...		$\times$	b21	...		=	z21	...	
...				...				...		

# Maticové operace

## Násobení dvou matic



- Opět máme vektor dílčích výsledků z
- Násobení je vektorové, součet různých vektorů z lze také vektorizovat


```
void multiply(std::vector<int>& A, std::vector<int>& B, std::vector<int>& C) {  
    #pragma omp declare reduction(vec_int_plus : std::vector<int> : \  
        std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>())) \  
        initializer(omp_priv = omp_orig)  
  
    #pragma omp parallel for collapse(2) num_threads(thread_count) reduction(vec_int_plus : C)  
    for (int r1=0; r1<ROWS; r1++) {  
        for (int r2=0; r2<ROWS; r2++) {  
            for (int k=0; k<ROWS; k++) {  
                C[r1 * COLS + k] += A[r1 * COLS + r2] * B[r2 * COLS + k];  
            }  
        }  
    }  
}
```

# Maticové operace

## Gaussova eliminace

- Dalším typickým úkolem je řešení soustavy lineárních rovnic
- Lze využít Gaussovu eliminaci

a11	a12	a13	b1
a21	...		b2
...			b3



a11	a12	a13	b1
0	a'22	a'23	b'2
0	0	a'33	b'3

- Jak můžeme paralelizovat?

# Maticové operace

## Gaussova eliminace

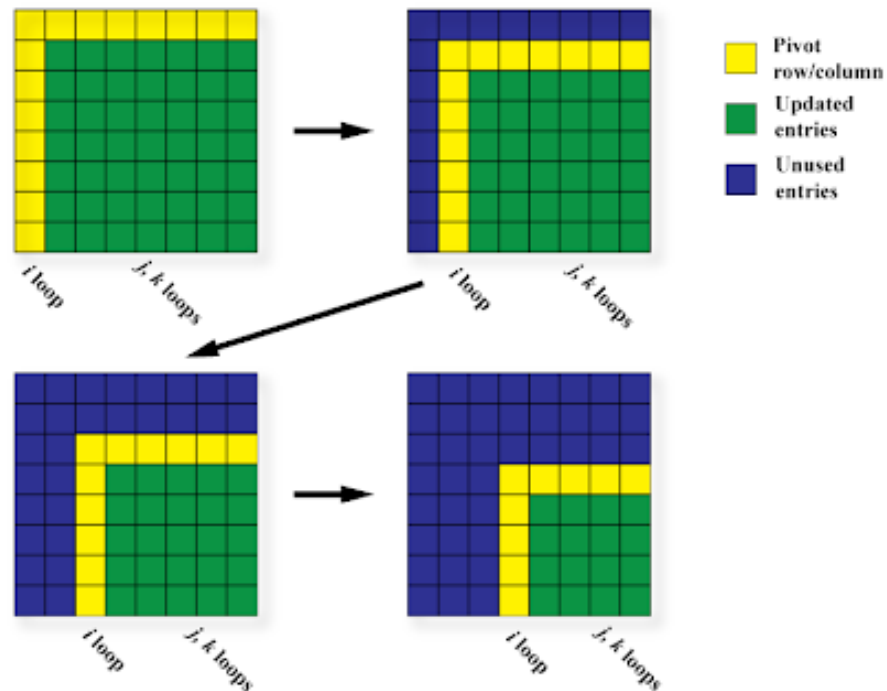
- Jaké jsou závislosti mezi hodnotami?
  - Po výběru pivotu se změny všechny hodnoty pro řádky a sloupce větší než pozice pivotu

# Maticové operace

## Gaussova eliminace

- Jaké jsou závislosti mezi hodnotami?
  - Po výběru pivotu se změny všech hodnot pro řádky a sloupce větší než pozice pivotu
- Kterou část můžeme paralelizovat?

```
void gauss(std::vector<double>& A) {  
    for (int i=0; i<ROWS; i++) {  
        // Make all rows below this one 0 in current column  
        for (int k=i+1; k<ROWS; k++) {  
            double c = -A[k * COLS + i]/A[i*COLS + i];  
            for (int j=i; j<ROWS; j++) {  
                if (i==j) {  
                    A[k * COLS + j] = 0;  
                } else {  
                    A[k * COLS + j] += c * A[i * COLS + j];  
                }  
            }  
        }  
    }  
}
```



# Maticové operace

## Gaussova eliminace

- Jaké jsou závislosti mezi hodnotami?
  - Po výběru pivotu se změny všechny hodnoty pro řádky a sloupce větší než pozice pivotu

```
void gauss_par(std::vector<double>& A) {
#pragma omp declare reduction(vec_int_plus : std::vector<double> : \
    std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<double>())) \
    initializer(omp_priv = omp_orig)

    for (int i=0; i<ROWS; i++) {
        // Make all rows below this one 0 in current column
#pragma omp parallel for num_threads(thread_count)
        for (int k=i+1; k<ROWS; k++) {
            double c = -A[k * COLS + i]/A[i*COLS + i];
            for (int j=i; j<ROWS; j++) {
                if (i==j) {
                    A[k * COLS + j] = 0;
                } else {
                    A[k * COLS + j] += c * A[i * COLS + j];
                }
            }
        }
    }
}
```



# Maticové operace

## Gaussova eliminace

- Co lze dále zefektivnit?

# Maticové operace

## Gaussova eliminace

- Co lze dále zefektivnit?

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)

(a) Iteration  $k = 0$  starts

1	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	√(1,1)	√(1,2)	√(1,3)	√(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)

(b)

1	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	√(2,1)	√(2,2)	√(2,3)	√(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)

(c)

1	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	√(3,1)	√(3,2)	√(3,3)	√(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)

(d)

1	(0,1)	(0,2)	(0,3)	(0,4)
0	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	√(4,1)	√(4,2)	√(4,3)	√(4,4)

(e) Iteration  $k = 1$  starts

1	(0,1)	(0,2)	(0,3)	(0,4)
0	1	(1,2)	(1,3)	(1,4)
0	(2,1)	√(2,2)	√(2,3)	√(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)

(f)

1	(0,1)	(0,2)	(0,3)	(0,4)
0	(1,1)	(1,2)	(1,3)	(1,4)
0	(2,1)	(2,2)	(2,3)	(2,4)
0	(3,1)	√(3,2)	√(3,3)	√(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)

(g) Iteration  $k = 0$  ends

1	(0,1)	(0,2)	(0,3)	(0,4)
0	1	(1,2)	(1,3)	(1,4)
0	(2,1)	(2,2)	(2,3)	(2,4)
0	(3,1)	(3,2)	(3,3)	(3,4)
0	(4,1)	√(4,2)	√(4,3)	√(4,4)

(h)

# Shrnutí paralelní části

- Základní nástroje pro psaní paralelního programu
  - vlákna a práce s nimi
  - Atomické proměnné
  - OpenMP
  - (Vektorizace, SIMD paralelizace)

# Shrnutí paralelní části

- Základní nástroje pro psaní paralelního programu
  - vlákna a práce s nimi
  - Atomické proměnné
  - OpenMP
  - (Vektorizace, SIMD paralelizace)
- Základní techniky paralelizace
  - Rozděluj a panuj
  - Threadpool
  - Dekompozice, nalezení co možná nejvíc paralelně bezkonfliktně vykonatelných operací

# Shrnutí paralelní části

- Základní nástroje pro psaní paralelního programu
  - vlákna a práce s nimi
  - Atomické proměnné
  - OpenMP
  - (Vektorizace, SIMD paralelizace)
- Základní techniky paralelizace
  - Rozděluj a panuj
  - Threadpool
  - Dekompozice, nalezení co možná nejvíc paralelně bezkonfliktně vykonatelných operací
- Základní algoritmy
  - Řazení
  - Maticové operace

# Shrnutí paralelní části

- Paralelizujete s cílem zefektivnit běh programu/algoritmu
- Musíte (alespoň částečně) rozumět vykonávání programu na HW
  - False sharing
  - Cache optimization

# Shrnutí paralelní části

- Paralelizujete s cílem zefektivnit běh programu/algoritmu
- Musíte (alespoň částečně) rozumět vykonávání programu na HW
  - False sharing
  - Cache optimization
- Vynechali jsme spoustu věcí
  - Úvodní kurz, aby jste získali základní znalosti a zkušenosti
- Pokud Vás paralelní programování zaujalo
  - Paralelní algoritmy (B4M35PAG)
  - Obecné výpočty na grafických procesorech (B4M39GPU)

# Shrnutí paralelní části

- Pro implementační zkoušku – programujte, programujte, programujte!



# Shrnutí paralelní části

- Pro implementační zkoušku – programujte, programujte, programujte!
  - Dostanete problém + sériový algoritmus
  - Cílem bude **zrychlit** algoritmus **díky paralelizaci**
    - **Paralelizace musí být korektní!** (musíte přemýšlet, ne všechny chyby se projeví, paralelní programování je nedeterministické)

# Shrnutí paralelní části

- Pro implementační zkoušku – programujte, programujte, programujte!
  - Dostanete problém + sériový algoritmus
  - Cílem bude **zrychlit** algoritmus **díky paralelizaci**
    - **Paralelizace musí být korektní!** (musíte přemýšlet, ne všechny chyby se projeví, paralelní programování je nedeterministické)
- V dalším studiu/práci – paralelizujte, pokud je to potřeba!
  - Pracujte iterativně – nejdřív je potřeba mít korektní sériovou variantu
  - Pokud je pomalá – zrychlujeme, paralelizujeme, atd.