CS 7172
Parallel and Distributed Computation

OpenMP

Kun Suo

Computer Science, Kennesaw State University

https://kevinsuo.github.io/

Scope of Variables

Scope

- ***In serial programming***, the scope of a variable consists of those parts of a program in which the variable can be used.

- In OpenMP, the scope of a variable refers to the set of threads that can ***access*** the variable ***in a parallel block***.

Scope in OpenMP

- A variable that can be accessed by all the threads in the team has shared scope.
- A variable that can only be accessed by a single thread has private scope.
- The default scope for variables declared before a parallel block is shared.

The default clause

Vairable k and factor are private;
Variable n is shared.

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
            default(none) reduction(+:sum) private(k, factor) \
            shared(n)
        for (k = 0; k < n; k++) {
            if (k % 2 == 0)
               factor = 1.0;
            else
               factor = -1.0;
            sum += factor/(2*k+1);
        }
```

The default clause

- Let the programmer specify the scope of each variable in a block.

```
default(none)
```

- With this clause the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block.

The default clause

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
            default(none) reduction(+:sum) private(k, factor) \
            shared(n)
        for (k = 0; k < n; k++) {
            if (k % 2 == 0)
               factor = 1.0;
            else
               factor = -1.0;
            sum += factor/(2*k+1);
        }
```

The Reduction Clause

Our example

- If we write the code like this…

```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count)
    {
#       pragma omp critical
        global_result += Local_trap(double a, double b, int n);
    }
```

… we force the threads to execute sequentially.

Our example

- We can avoid this problem by declaring a private variable inside the parallel block and moving the critical section after the function call.

```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count)
    {
       double my_result = 0.0;  /* private */

       my_result += Local_trap(double a, double b, int n);
#      pragma omp critical
       global_result += my_result;
    }
```

Parallel computation

Serial computation

The Reduction Clause

- A common loop is to accumulate variables

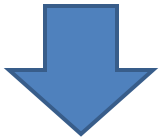
```
int sum = 0;

for (int i = 0; i < 100; i++) {
    sum += array[i];
}
```

sum needs to be ***private*** for parallelization

sum needs to be ***public*** for the correct answer

The Reduction Clause

```
int sum = 0;

for (int i = 0; i < 100; i++) {
    sum += array[i];
}
```



```
int sum = 0;

#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < 100; i++) {
        sum += array[i];
}
```

In the internal implementation, OpenMP provides a ***private*** sum variable for each thread.

When the thread exits, OpenMP adds the sum of each thread's parts to get the final result.

The Reduction Clause

```
int sum = 0;

#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < 100; i++) {
        sum += array[i];
}
```

=

```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count)
    {
       double my_result = 0.0;  /* private */

       my_result += Local_trap(double a, double b, int n);
#      pragma omp critical
       global_result += my_result;
    }
```

Reduction operators

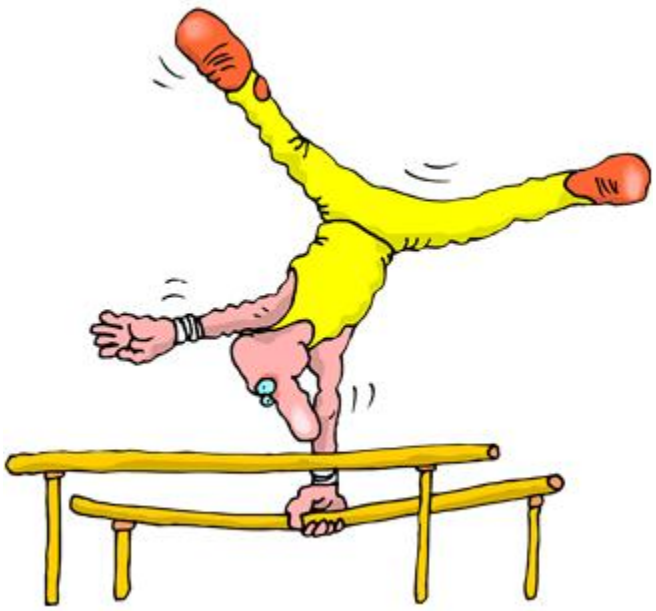
```
reduction(<operator>: <variable list>)
```

+, *, -, &, |, ^, &&, ||

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
   global_result += Local_trap(double a, double b, int n);
```

The global_result is private for each thread and public for the master thread

The “Parallel For” Directive



Parallel for

- Forks a team of threads to execute the following structured block.

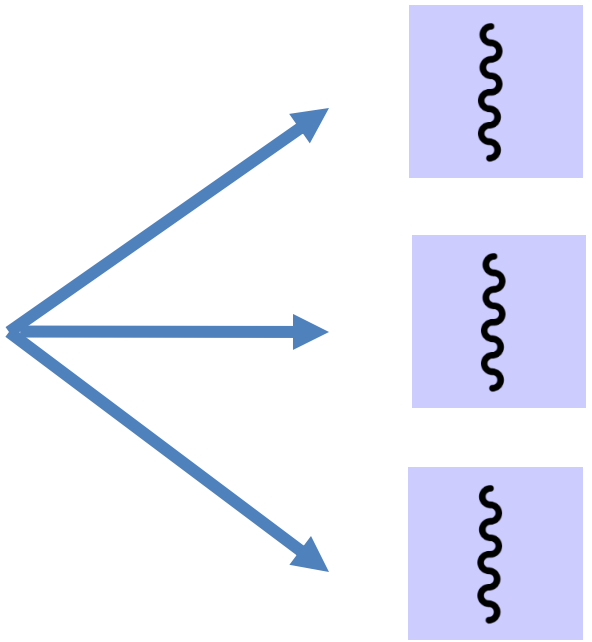
- However, the structured block following the parallel for directive must be ***a for loop***.

```
int sum = 0;

#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < 100; i++) {
        sum += array[i];
}
```

Parallel for

- Furthermore, with the parallel for directive the system parallelizes the for loop by dividing the iterations of the loop among the threads.

```
int sum = 0;

for (int i = 0; i < 100; i++) {
    sum += array[i];
}
```



Parallel for Example

```
int sum = 0;

for (int i = 0; i < 100; i++) {
    sum += array[i];
}
```

```
int sum = 0;

#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < 100; i++) {
        sum += array[i];
}
```

Data dependencies

```
fibo[ 0 ] = fibo[ 1 ] = 1;
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

$$F_n = F_{n\text{-}1} + F_{n\text{-}2}$$

Data dependencies

```
fibo[ 0 ] = fibo[ 1 ] = 1;
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```


note 2 threads

```
fibo[ 0 ] = fibo[ 1 ] = 1;
# pragma omp parallel for num_threads(2)
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

$$F_n = F_{n\text{-}1} + F_{n\text{-}2}$$

Data dependencies

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

- Fibonacci in serial

```c
#include <stdio.h>

int main()
{
        int fibo[10];
        fibo[0] = fibo[1] = 1;
        int i;

        printf("%d\n", fibo[0]);
        printf("%d\n", fibo[1]);

        for (i = 2; i < 10; i++)
        {
                fibo[i] = fibo[i-1] + fibo[i-2];
                printf("%d\n", fibo[i]);
        }

        return 0;
}
```

```
ksuo@ksuo-VirtualBox ~/cs7172> ./fibonacci.o
1
1
2
3
5
8
13
21
34
55
```

Data dependencies

https://github.com/kevinsuo/CS7172/blob/master/fibonacci-omp.c

- Fibonacci in openmp

$ gcc -fopenmp xxx.c -o xxx.o

```c
#include <stdio.h>
#include <omp.h>

int main()
{
        int fibo[10];
        fibo[0] = fibo[1] = 1;
        int i;

        printf("%d\n", fibo[0]);
        printf("%d\n", fibo[1]);

#pragma omp parallel for num_threads(2)
        for (i = 2; i < 10; i++)
        {
                fibo[i] = fibo[i-1] + fibo[i-2];
                printf("%d\n", fibo[i]);
        }

        return 0;
}
```

```
ksuo@ksuo-VirtualBox ~/cs7172> ./fibonacci-omp.o
1
1
2
3
5
8
303804267
303804275
607608542
911412817
```

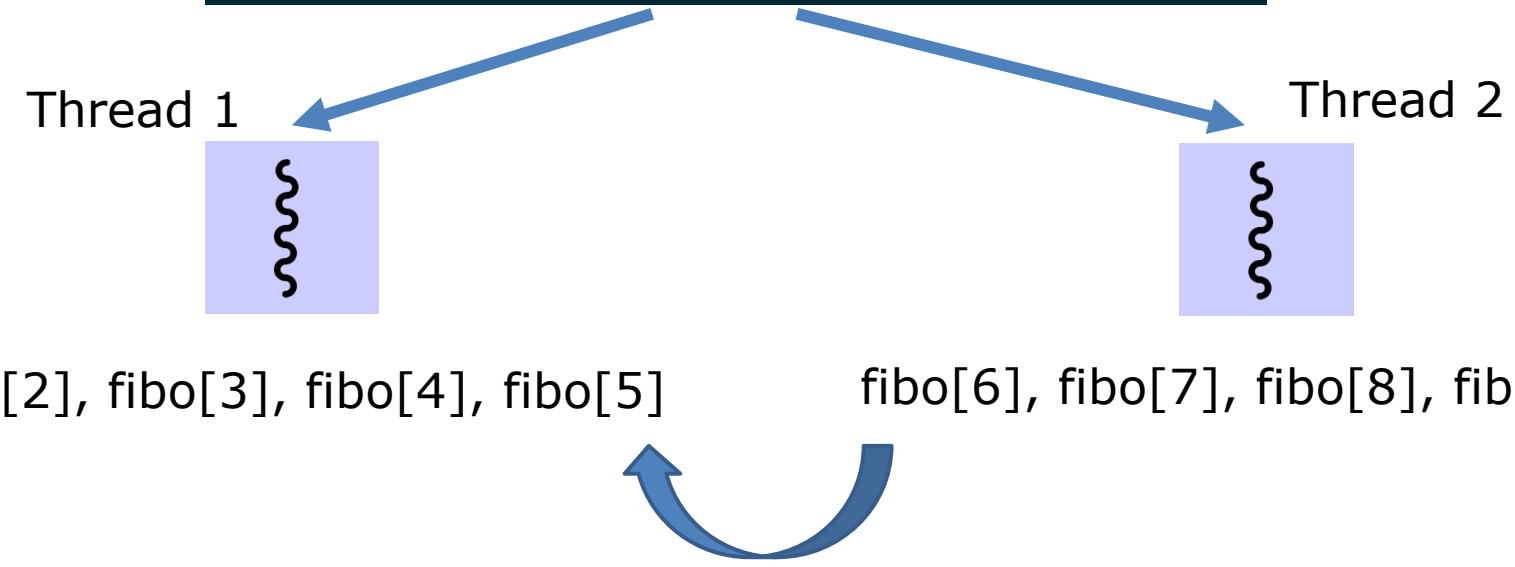
What happened?

1. OpenMP compilers don't check for ***dependences*** among iterations in a loop that's being parallelized with a parallel for directive.

2. A loop in which the results of one or more iterations depend on other iterations ***cannot be correctly parallelized*** by OpenMP.

What happened?

```
#pragma omp parallel for num_threads(2)
        for (i = 2; i < 10; i++)
        {
                fibo[i] = fibo[i-1] + fibo[i-2];
                printf("%d\n", fibo[i]);
        }
```



What will happen if thread 2 runs ahead of thread 1?

Example: Estimating π

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
    sum += factor/(2*k+1);
    factor = -factor;
}
pi_approx = 4.0*sum;
```

OpenMP solution #1

https://github.com/kevinsuo/CS7172/blob/master/pi-cal.c

loop dependency: the factor in k^{th} iteration has dependency of the $k-1^{th}$ iteration

```
        double factor = 1.0;
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
            reduction(+:sum)
        for (k = 0; k < n; k++) {
            sum += factor/(2*k+1);
            factor = -factor;
        }
        pi_approx = 4.0*sum;
```

OpenMP solution #1

https://github.com/kevinsuo/CS7172/blob/master/pi-cal.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define n 100000

int main(int argc, char *argv[]) {
        double factor = 1.0;
        double sum, pi_approx = 0.0;
        int k = 0;


#       pragma omp parallel for reduction(+:sum)
        for (k=0; k<n; k++) {
                sum += factor/(2*k+1);
                factor = -factor;
        }
        pi_approx = 4.0 * sum;

        printf("pi is %f\n", pi_approx);
        return 0;
}
```

gcc pi-cal.c -o pi-cal.o -fopenmp

```
ksuo@ksuo-VirtualBox ~/openmp> ./pi-cal.o
pi is 3.141753
ksuo@ksuo-VirtualBox ~/openmp> ./pi-cal.o
pi is -3.141179
ksuo@ksuo-VirtualBox ~/openmp> ./pi-cal.o
pi is -3.141540
ksuo@ksuo-VirtualBox ~/openmp> ./pi-cal.o
pi is 3.141406
ksuo@ksuo-VirtualBox ~/openmp> ./pi-cal.o
pi is 3.141462
ksuo@ksuo-VirtualBox ~/openmp> ./pi-cal.o
pi is -4.857996
ksuo@ksuo-VirtualBox ~/openmp> ./pi-cal.o
pi is -4.842982
ksuo@ksuo-VirtualBox ~/openmp> ./pi-cal.o
pi is 3.141690
ksuo@ksuo-VirtualBox ~/openmp> ./pi-cal.o
pi is 3.141630
ksuo@ksuo-VirtualBox ~/openmp> ./pi-cal.o
pi is -3.142635
```

OpenMP solution #2

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
            reduction(+:sum) private(factor)
        for (k = 0; k < n; k++) {
            if (k % 2 == 0)
               factor = 1.0;
            else
               factor = -1.0;
            sum += factor/(2*k+1);
        }
```

Insures factor has private scope.

OpenMP solution #2

https://github.com/kevinsuo/CS7172/blob/master/pi-cal-2.c

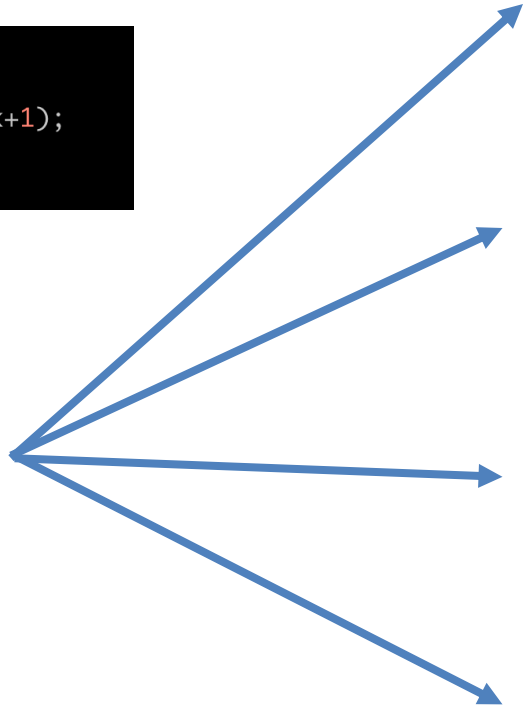
```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define n 100000

int main(int argc, char *argv[]) {
        double factor = 1.0;
        double sum, pi_approx = 0.0;
        int k = 0;


#       pragma omp parallel for reduction(+:sum) private(factor)
        for (k=0; k<n; k++) {
                if (k % 2 == 0)
                        factor = 1.0;
                else
                        factor = -1.0;
                sum += factor/(2*k+1);
        }
        pi_approx = 4.0 * sum;

        printf("pi is %f\n", pi_approx);
        return 0;
}
```

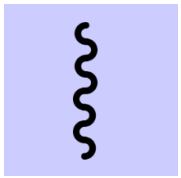
```
ksuo@ksuo-VirtualBox ~/openmp> ./pi-cal.o
pi is 3.141583
ksuo@ksuo-VirtualBox ~/openmp> ./pi-cal.o
pi is 3.141583
ksuo@ksuo-VirtualBox ~/openmp> ./pi-cal.o
pi is 3.141583
ksuo@ksuo-VirtualBox ~/openmp> ./pi-cal.o
pi is 3.141583
ksuo@ksuo-VirtualBox ~/openmp> ./pi-cal.o
pi is 3.141583
ksuo@ksuo-VirtualBox ~/openmp> ./pi-cal.o
pi is 3.141583
```

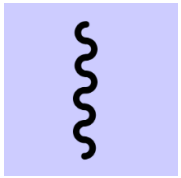
Which thread executes which round of loop?

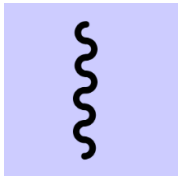
```
#       pragma omp parallel for
        for (k=0; k<n; k++) {
                sum += factor/(2*k+1);
                factor = -factor;
        }
```

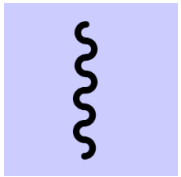
k=0
=1
=2
=3
… …
=10
=11
=12











schedule (type , chunksize)

- Type can be:
  - static: the iterations can be assigned to the threads before the loop is executed.
  - dynamic or guided: the iterations are assigned to the threads while the loop is executing.
  - auto: the compiler and/or the run-time system determine the schedule.
  - runtime: the schedule is determined at run-time.

- The chunksize is a positive integer.

The Schedule Clause

- Default schedule:

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

- Cyclic schedule:

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum) schedule(static,1)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

The Static Schedule Type

- twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,1)
```

$$\begin{aligned} \text{Thread } 0: &\quad 0,3,6,9 \\ \text{Thread } 1: &\quad 1,4,7,10 \\ \text{Thread } 2: &\quad 2,5,8,11 \end{aligned}$$

The Static Schedule Type

- twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,2)
```

$$\begin{aligned} \text{Thread } 0: &\quad 0, 1, 6, 7 \\ \text{Thread } 1: &\quad 2, 3, 8, 9 \\ \text{Thread } 2: &\quad 4, 5, 10, 11 \end{aligned}$$

The Static Schedule Type

- twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,4)
```

$$\begin{aligned} \text{Thread } 0: &\quad 0,1,2,3 \\ \text{Thread } 1: &\quad 4,5,6,7 \\ \text{Thread } 2: &\quad 8,9,10,11 \end{aligned}$$

The Dynamic Schedule Type

- The iterations are also broken up into chunks of chunksize consecutive iterations.
- Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system.
- This continues until all the iterations are completed.
- The allocation results are unpredictable.

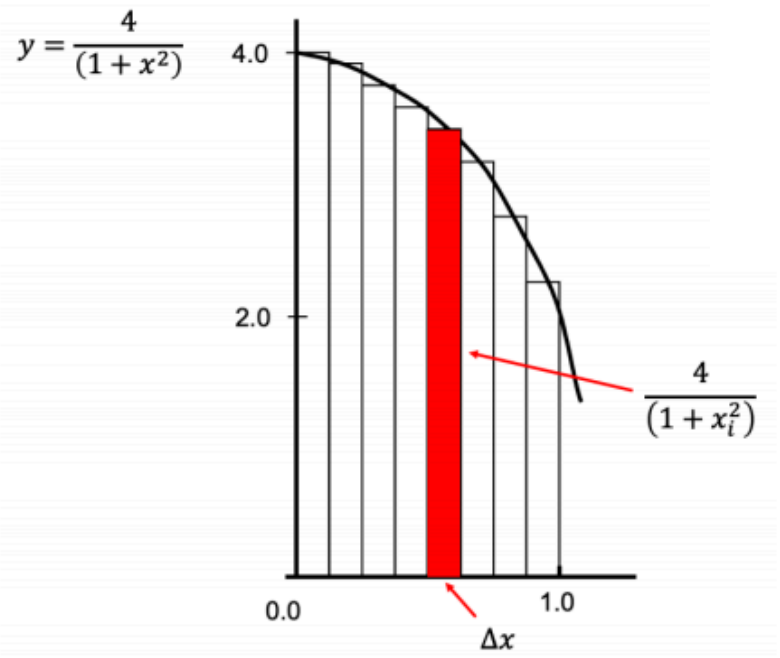
The Dynamic Schedule Type

- twelve iterations, 0, 1, . . . , 11, and three threads

$$\begin{aligned} &\text{Thread } 0: \quad 0, 1, 2, 3, 6, 7 \\ &\text{Thread } 1: \quad 4, 5 \\ &\text{Thread } 2: \quad 8, 9, 10, 11 \end{aligned}$$

Approximate the value of π

$$\int_0^1 \frac{4}{(1+x^2)}dx = \pi$$



$$\int_a^b f(x)dx = y_0\Delta x + y_1\Delta x + \cdots + y_{n-1}\Delta x$$

$$\Delta x = (b-a)/n$$

$$y = f(x)$$

$$y_i = f(a + i * (b-a)/n) \quad i = 0,1,2,\ldots,n$$

Approximate the value of π

```
ksuo@ksuo-VirtualBox ~/Desktop> ./pi.o
elapsed time = 14423708 nanoseconds
3.141593
ksuo@ksuo-VirtualBox ~/Desktop> ./pi.o
elapsed time = 14872618 nanoseconds
3.141593
ksuo@ksuo-VirtualBox ~/Desktop> ./pi.o
elapsed time = 15139145 nanoseconds
3.141593
ksuo@ksuo-VirtualBox ~/Desktop> ./pi.o
elapsed time = 14786384 nanoseconds
3.141593
ksuo@ksuo-VirtualBox ~/Desktop> ./pi.o
elapsed time = 14896299 nanoseconds
3.141593
```

```c
#include <stdio.h>
#include <omp.h>
#include <time.h>

static long num_steps = 1000000;
double step;
void main(){
        struct timespec start, end;
        clock_gettime(CLOCK_MONOTONIC, &start);

        int i;
        double x, pi, sum = 0.0;
        step = 1.0/(double)num_steps;
        for(i=1;i<= num_steps;i++){
                x = (i-0.5)*step;
                sum=sum+4.0/(1.0+x*x);
        }
        pi=step*sum;

        clock_gettime(CLOCK_MONOTONIC, &end);
        double diff = 1000000000L * (end.tv_sec - start.tv_sec) + end.tv_nsec - start.tv_nsec;

        printf("elapsed time = %llu nanoseconds\n", (long long unsigned int) diff);
        printf("%lf\n",pi);
}
```

https://github.com/kevinsuo/CS7172/blob/master/pi-non-openmp.c

Approximate the value of π

```c
#include <stdio.h>
#include <omp.h>
#include <time.h>

static long num_steps = 1000000;
double step;
#define NUM_THREADS 4
void main ()
{
        struct timespec start, end;
        clock_gettime(CLOCK_MONOTONIC, &start);

        int i;
        double x, pi, sum[NUM_THREADS];
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private(i)
        {
                double x;
                int id;
                id = omp_get_thread_num();
                for (i=id, sum[id]=0.0;i< num_steps; i=i+NUM_THREADS){
                        x = (i+0.5)*step;
                        sum[id] += 4.0/(1.0+x*x);
                }
        }
        for(i=0, pi=0.0;i<NUM_THREADS;i++)
                pi += sum[i] * step;

        clock_gettime(CLOCK_MONOTONIC, &end);
        double diff = 1000000000L * (end.tv_sec - start.tv_sec) + end.tv_nsec - start.tv_nsec;

        printf("elapsed time = %llu nanoseconds\n", (long long unsigned int) diff);

        printf("%lf\n",pi);
}
```

https://github.com/kevinsuo/CS7172/blob/master/pi-openmp.c

```
ksuo@ksuo-VirtualBox ~/Desktop> ./pi-openmp.o
elapsed time = 10076458 nanoseconds
3.141593
ksuo@ksuo-VirtualBox ~/Desktop> ./pi-openmp.o
elapsed time = 6482826 nanoseconds
3.141593
ksuo@ksuo-VirtualBox ~/Desktop> ./pi-openmp.o
elapsed time = 16829159 nanoseconds
3.141593
ksuo@ksuo-VirtualBox ~/Desktop> ./pi-openmp.o
elapsed time = 10507221 nanoseconds
3.141593
ksuo@ksuo-VirtualBox ~/Desktop> ./pi-openmp.o
elapsed time = 8458411 nanoseconds
3.141593
```

An example of parallel and distributed computation: 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

```
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

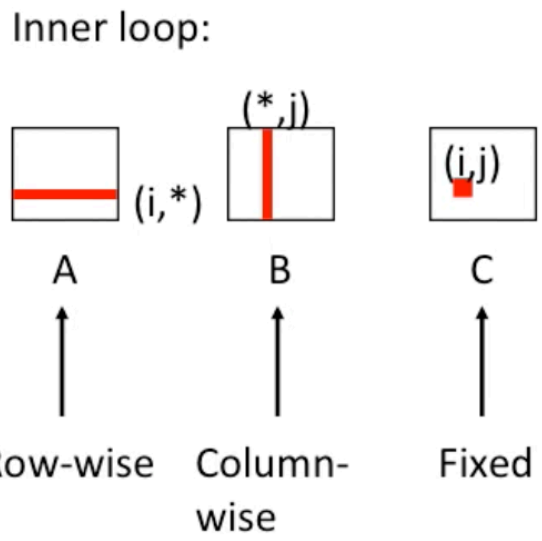
https://github.com/kevinsuo/CS7172/blob/master/matrix.c

```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

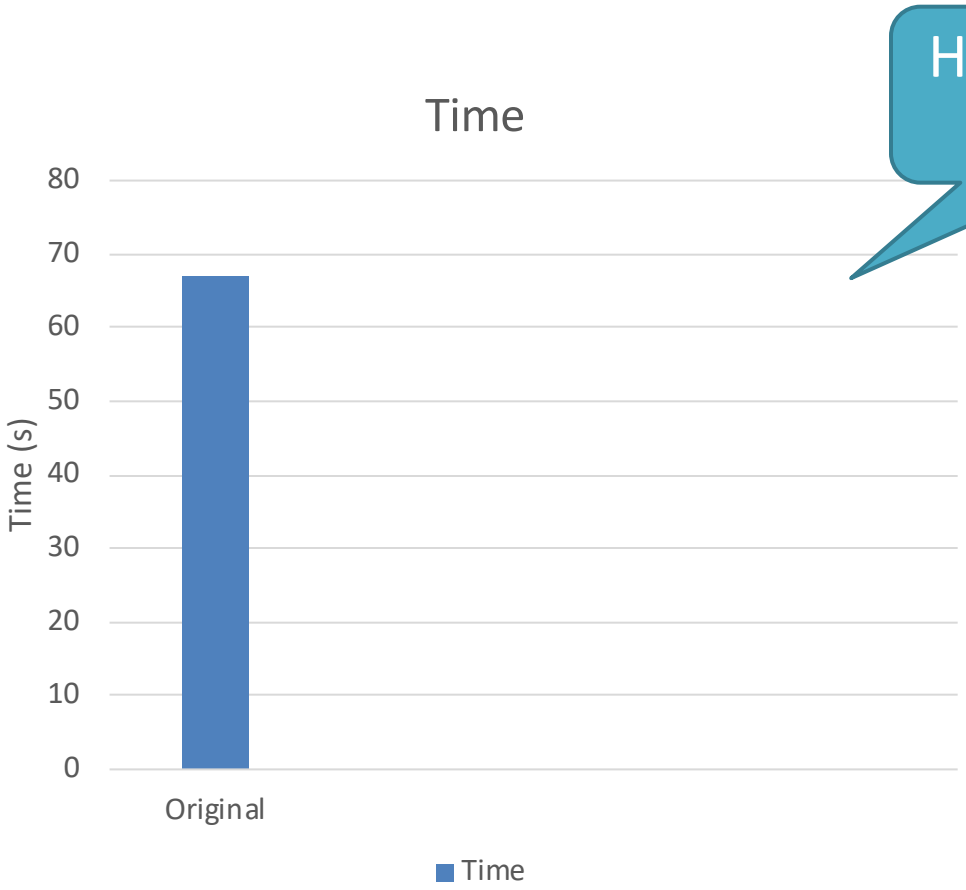
https://github.com/kevinsuo/CS7172/blob/master/matrix.c

```
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];
                        }
                }
        }
}
```



Matrix multiply

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



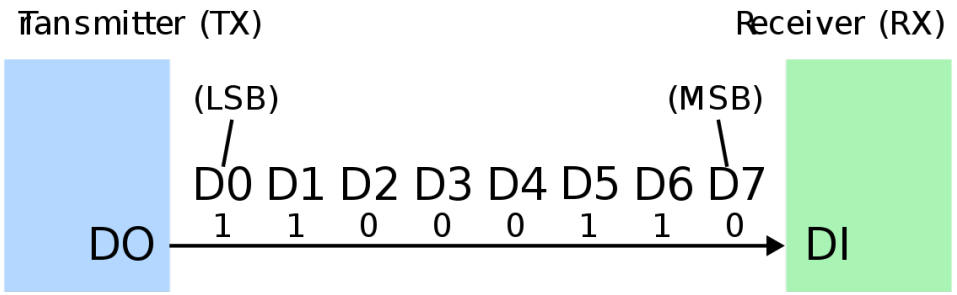
How to run it faster?

1. Accelerate serial execution

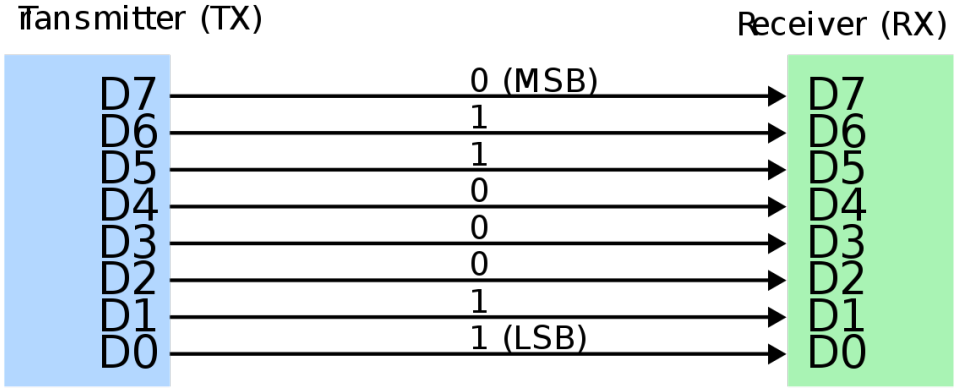
2. Accelerate in parallel

How to run it faster?

1. Accelerate serial execution
Reduce unnecessary steps



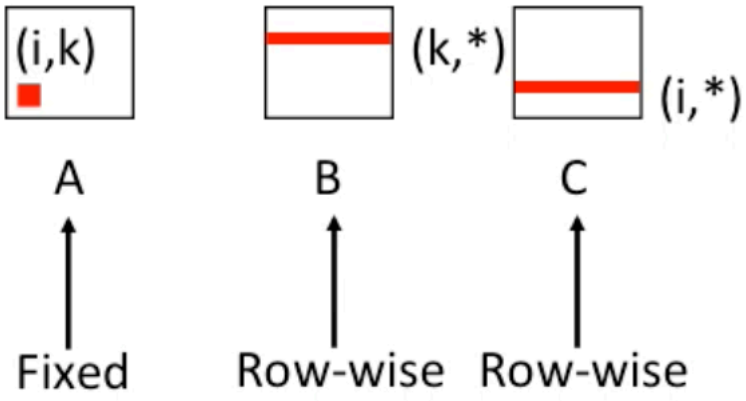
2. Accelerate in parallel



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:



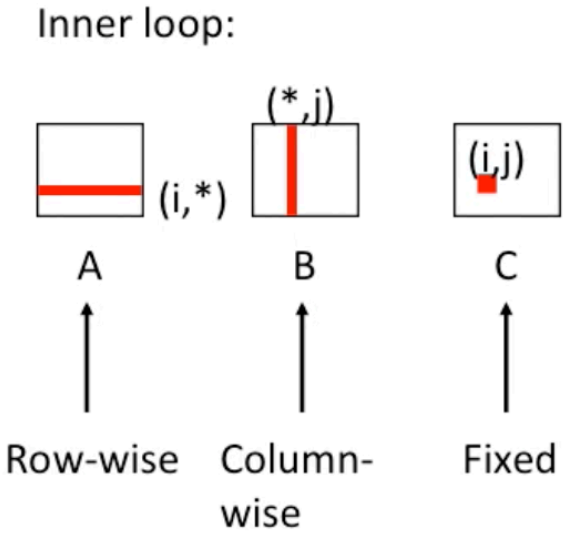
https://github.com/kevinsuo/CS7172/blob/master/matrix-opt.c

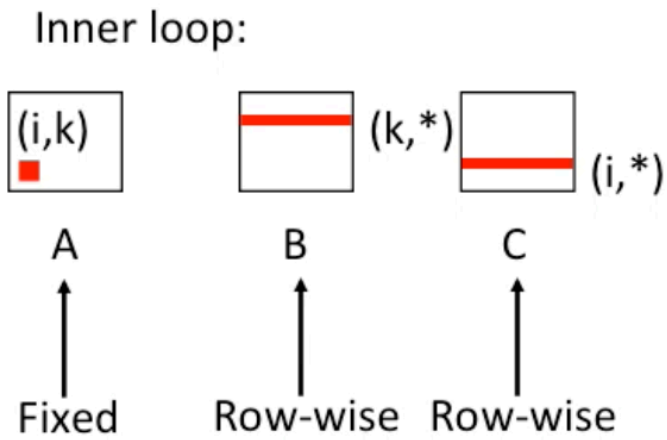
Option 1: Optimization using locality

```
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

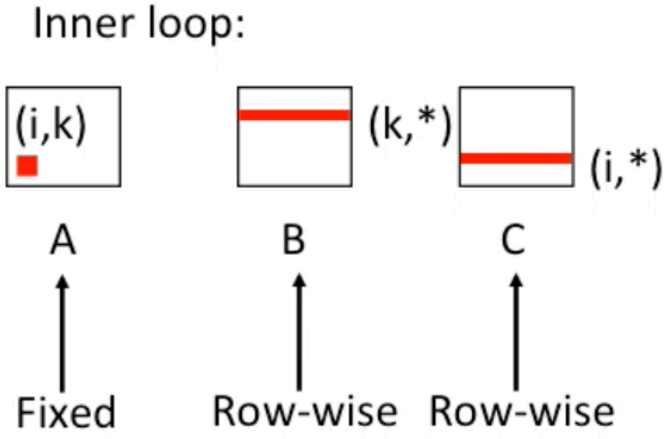
3.7x



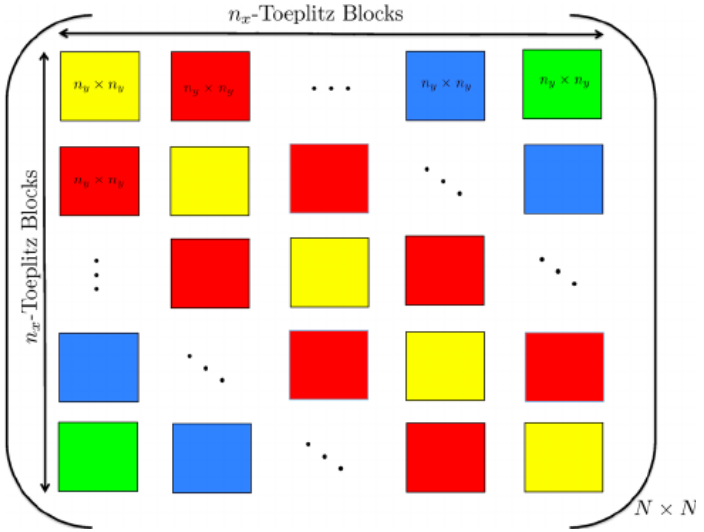


Option 1: Optimization using locality

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



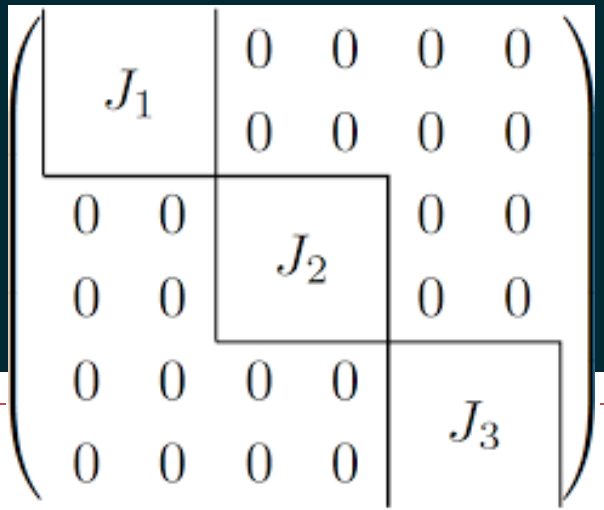
- Spatial locality
  - Divide the large matrix into smaller ones and put it inside the cache during calculation



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



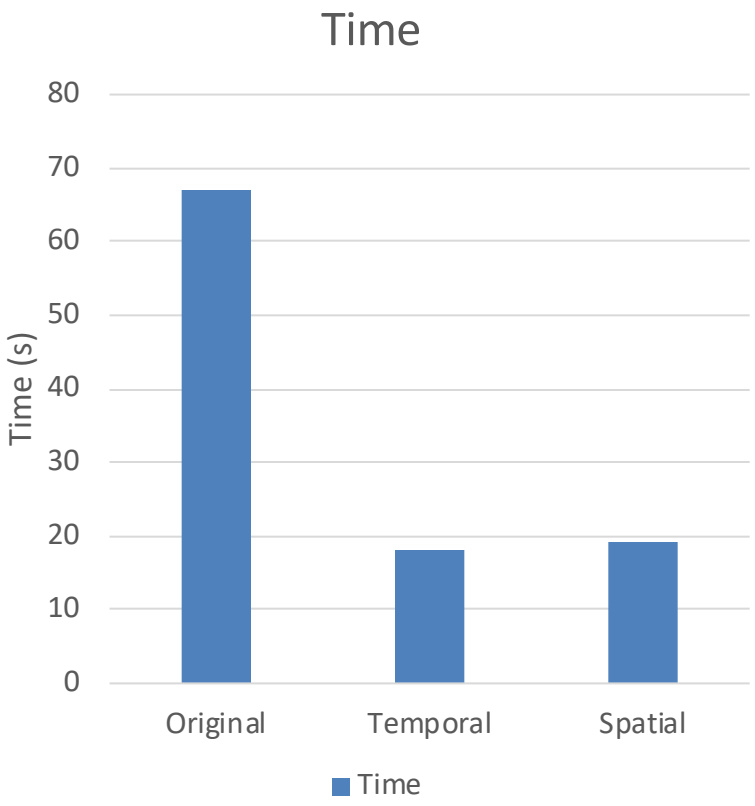
Option 1: Optimization using locality

$$A = \left(\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right) \Longrightarrow \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

$$A_{11} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, A_{12} = \begin{pmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{pmatrix}$$

$$A_{21} = \begin{pmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix}, A_{22} = \begin{pmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{pmatrix}$$

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

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

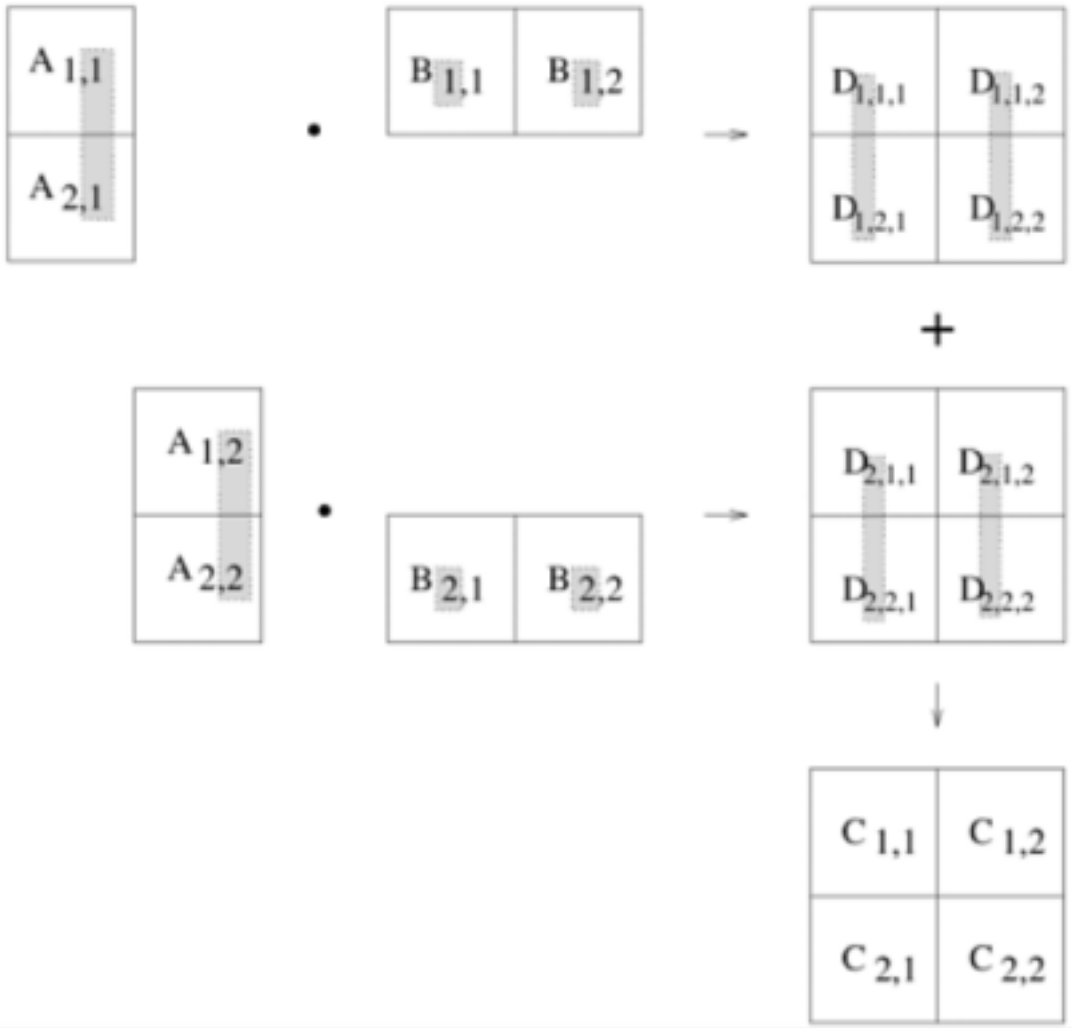
(a)

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

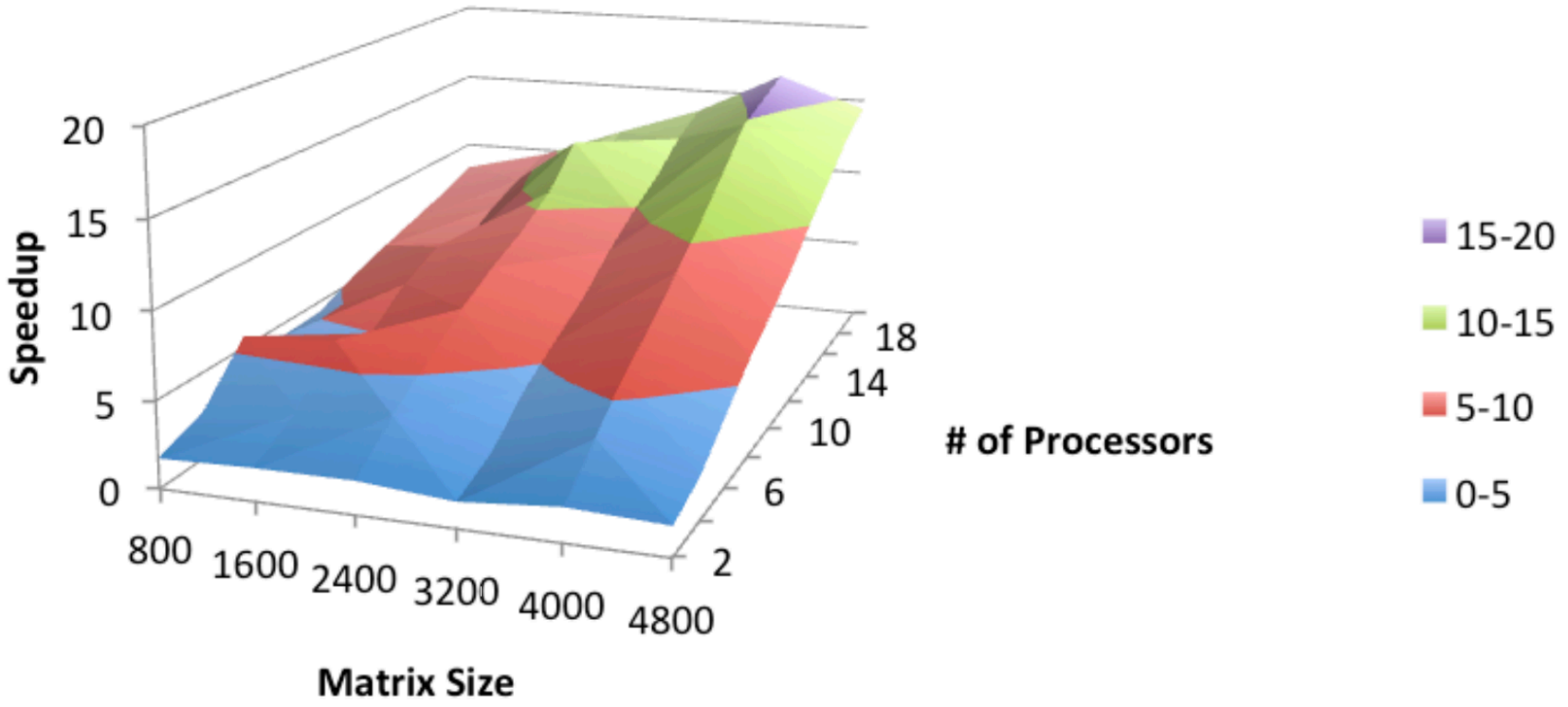
Optimal 2: Optimization using parallel

Thread 1:

Thread 2:



Optimal 2: Optimization using parallel



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

Matrix-vector multiplication

$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1} \end{bmatrix}$$

```
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}
```

Matrix-vector multiplication

```
#  pragma omp parallel for num_threads(thread_count)  \
      default(none) private(i, j) shared(A, x, y, m, n)
   for (i = 0; i < m; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i][j]*x[j];
   }
```

Run-times and efficiencies of matrix-vector multiplication (times are in seconds)

Threads	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Conclusion

- Scope of variable
- Reduce
- Data and loop dependency
- Schedule
- Examples of OpenMP