Завдання:

1.1 Визначити, чи підтримує компілятор обробку директив OpenMP
1.2 За допомогою функцій OpenMP визначити час, потрібний системі для роботи функції вимірювання часу. Визначити точність системного таймера.
2. Реалізувати множення квадратних матриць з використанням OpenMP.

Теоретичні відомості:

Множення матриць ($A \times B$) – є операція обчислення матриці C, кожен елемент якої дорівнює сумі добутків елементів у відповідному рядку першого множника і стовпці другого.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Кількість стовпців в матриці A має збігатися з кількістю рядків у матриці B, іншими словами, матриця A обов'язково має бути узгодженою з матрицею B. Якщо матриця A має розмірність $n \times m$, B – $m \times k$, то розмірність їхнього добутку C є $n \times k$.

Реалізація завдань 1.1, 1.2, 1.3 (код, демонстрація)

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

int main()
{
	#ifdef _OPENMP
	puts("OpenMP is supported.");
	#endif

	long double start_time, end_time, tick;
	start_time = omp_get_wtime();
	end_time = omp_get_wtime();
	tick = omp_get_wtick();

	printf("end_time - start_time = %.20lf; tick = %.7lf\n", end_time - start_time, tick);

	#pragma omp parallel
	{
		int thread_num = omp_get_thread_num();
		printf("Hello world from thread with thread_num = %d\n", thread_num);
	}

	return 0;
}
```

```
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\DemoOpenMP.exe
OpenMP is supported.
end_time - start_time = 0.00000010000076144934; tick = 0.0000001
Hello world from thread with thread_num = 0
Hello world from thread with thread_num = 7
Hello world from thread with thread_num = 5
Hello world from thread with thread_num = 4
Hello world from thread with thread_num = 2
Hello world from thread with thread_num = 6
Hello world from thread with thread_num = 3
Hello world from thread with thread_num = 1
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release>
```

Реалізація завдання 2 (перевірка коректності)

Для тестування коректності рішення було використано 2 приклади:

Приклад 1:

A:

0	1	2	3	4
1	2	3	4	5
2	3	4	5	6
3	4	5	6	7

B:

0	1	2	3	4	5
1	3	5	7	9	11
2	5	8	11	14	17
3	7	11	15	19	23
4	9	14	19	24	29

Приклад 2:

A:

1	1	1	-1
-5	-3	-4	4
5	1	4	-3
-16	-11	-15	14

B:

7	-2	3	4
11	0	3	4
5	4	3	0
22	2	9	8

Запустити виконання прикладів можна за допомогою аргументу командного рядка v:

```
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe v
=== Single thread example 1 ===
Time: 0.000000
| 30.000000 | 70.000000 | 110.000000 | 150.000000 | 190.000000 | 230.000000 |

| 40.000000 | 95.000000 | 150.000000 | 205.000000 | 260.000000 | 315.000000 |

| 50.000000 | 120.000000 | 190.000000 | 260.000000 | 330.000000 | 400.000000 |

| 60.000000 | 145.000000 | 230.000000 | 315.000000 | 400.000000 | 485.000000 |


=== Single thread example 2 ===
Time: 0.000000
| 1.000000 | 0.000000 | 0.000000 | 0.000000 |

| 0.000000 | 2.000000 | 0.000000 | 0.000000 |

| 0.000000 | 0.000000 | 3.000000 | 0.000000 |

| 0.000000 | 0.000000 | 0.000000 | 4.000000 |


=== Multi thread example 1 ===
Time: 0.000610
| 30.000000 | 70.000000 | 110.000000 | 150.000000 | 190.000000 | 230.000000 |

| 40.000000 | 95.000000 | 150.000000 | 205.000000 | 260.000000 | 315.000000 |

| 50.000000 | 120.000000 | 190.000000 | 260.000000 | 330.000000 | 400.000000 |

| 60.000000 | 145.000000 | 230.000000 | 315.000000 | 400.000000 | 485.000000 |


=== Multi thread example 2 ===
Time: 0.0000007
| 1.000000 | 0.000000 | 0.000000 | 0.000000 |

| 0.000000 | 2.000000 | 0.000000 | 0.000000 |

| 0.000000 | 0.000000 | 3.000000 | 0.000000 |

| 0.000000 | 0.000000 | 0.000000 | 4.000000 |

PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release>
```

Реалізація завдання 2 (дослідження швидкодії)

Для дослідження швидкодії було використано згенерований приклад.
Код, що генерує приклад:

```
// Initialize a
double ** a = matrix_alloc(N_P, M_P);
for (int i = 0; i < N_P; i++)
{
	for (int j = 0; j < M_P; j++)
	{
		a[i][j] = i + j;
	}
}

// Initialize b
double ** b = matrix_alloc(M_P, K_P);
for (int i = 0; i < M_P; i++)
{
	for (int j = 0; j < K_P; j++)
	{
		b[i][j] = i * j + i + j;
	}
}
```

Розмірності матриць:

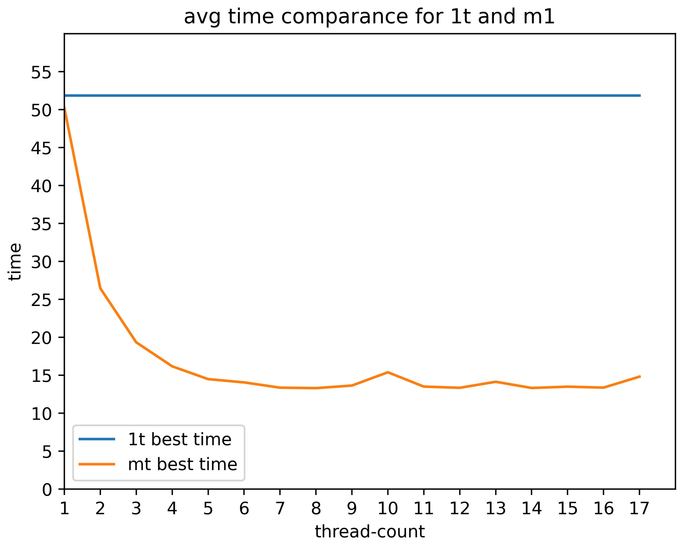
```
#define N_P 1024
#define M_P 2048
#define K_P 3072
```

Single thread performance run:

```
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe s
Single thread run 0 | Time: 53.029524
Single thread run 1 | Time: 52.121171
Single thread run 2 | Time: 52.089143
Single thread run 3 | Time: 51.848553
Single thread run 4 | Time: 52.744421
Best time: 51.848553
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release>
```

Multi thread performance run (n in [1, 17]):

```
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe m 1
Multi thread run 0; n = 1 | Time: 51.767033
Multi thread run 1; n = 1 | Time: 50.221874
Multi thread run 2; n = 1 | Time: 51.283795
Multi thread run 3; n = 1 | Time: 53.225370
Multi thread run 4; n = 1 | Time: 52.290860
Best time: 50.221874
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe m 2
Multi thread run 0; n = 2 | Time: 26.460316
Multi thread run 1; n = 2 | Time: 26.918980
Multi thread run 2; n = 2 | Time: 26.696870
Multi thread run 3; n = 2 | Time: 26.530649
Multi thread run 4; n = 2 | Time: 26.595843
Best time: 26.460316
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe m 3
Multi thread run 0; n = 3 | Time: 19.336353
Multi thread run 1; n = 3 | Time: 20.785103
Multi thread run 2; n = 3 | Time: 21.053827
Multi thread run 3; n = 3 | Time: 21.354076
Multi thread run 4; n = 3 | Time: 21.818698
Best time: 19.336353
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe m 4
Multi thread run 0; n = 4 | Time: 16.175317
Multi thread run 1; n = 4 | Time: 18.211670
Multi thread run 2; n = 4 | Time: 19.016858
Multi thread run 3; n = 4 | Time: 18.939337
Multi thread run 4; n = 4 | Time: 19.277555
Best time: 16.175317
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe m 5
Multi thread run 0; n = 5 | Time: 14.486773
Multi thread run 1; n = 5 | Time: 16.988550
Multi thread run 2; n = 5 | Time: 17.876853
Multi thread run 3; n = 5 | Time: 17.817613
Multi thread run 4; n = 5 | Time: 18.142916
Best time: 14.486773
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe m 6
Multi thread run 0; n = 6 | Time: 14.056830
Multi thread run 1; n = 6 | Time: 16.499281
Multi thread run 2; n = 6 | Time: 17.283317
Multi thread run 3; n = 6 | Time: 17.424334
Multi thread run 4; n = 6 | Time: 17.578709
Best time: 14.056830
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe m 7
Multi thread run 0; n = 7 | Time: 13.361369
Multi thread run 1; n = 7 | Time: 16.827416
Multi thread run 2; n = 7 | Time: 17.036933
Multi thread run 3; n = 7 | Time: 17.230839
Multi thread run 4; n = 7 | Time: 17.107746
Best time: 13.361369
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe m 8
Multi thread run 0; n = 8 | Time: 13.302053
Multi thread run 1; n = 8 | Time: 16.991370
Multi thread run 2; n = 8 | Time: 17.202358
Multi thread run 3; n = 8 | Time: 17.261102
Multi thread run 4; n = 8 | Time: 17.211156
Best time: 13.302053
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe m 9
Multi thread run 0; n = 9 | Time: 13.639270
Multi thread run 1; n = 9 | Time: 16.892380
Multi thread run 2; n = 9 | Time: 17.397582
Multi thread run 3; n = 9 | Time: 17.330922
Multi thread run 4; n = 9 | Time: 17.312324
Best time: 13.639270
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe m 10
Multi thread run 0; n = 10 | Time: 15.389566
Multi thread run 1; n = 10 | Time: 17.275815
Multi thread run 2; n = 10 | Time: 17.181204
Multi thread run 3; n = 10 | Time: 17.360946
Multi thread run 4; n = 10 | Time: 17.363062
Best time: 15.389566
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe m 11
Multi thread run 0; n = 11 | Time: 13.502090
Multi thread run 1; n = 11 | Time: 17.166458
Multi thread run 2; n = 11 | Time: 17.118975
Multi thread run 3; n = 11 | Time: 17.139387
Multi thread run 4; n = 11 | Time: 17.283431
Best time: 13.502090
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe m 12
Multi thread run 0; n = 12 | Time: 13.343987
Multi thread run 1; n = 12 | Time: 16.359024
Multi thread run 2; n = 12 | Time: 17.080481
Multi thread run 3; n = 12 | Time: 17.349253
Multi thread run 4; n = 12 | Time: 17.254864
Best time: 13.343987
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe m 13
Multi thread run 0; n = 13 | Time: 14.138751
Multi thread run 1; n = 13 | Time: 17.234241
Multi thread run 2; n = 13 | Time: 17.615324
Multi thread run 3; n = 13 | Time: 17.443238
Multi thread run 4; n = 13 | Time: 17.326847
Best time: 14.138751
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe m 14
Multi thread run 0; n = 14 | Time: 13.321032
Multi thread run 1; n = 14 | Time: 17.192534
Multi thread run 2; n = 14 | Time: 17.361008
Multi thread run 3; n = 14 | Time: 17.259120
Multi thread run 4; n = 14 | Time: 17.252583
Best time: 13.321032
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe m 15
Multi thread run 0; n = 15 | Time: 13.492839
Multi thread run 1; n = 15 | Time: 16.769180
Multi thread run 2; n = 15 | Time: 17.165785
Multi thread run 3; n = 15 | Time: 17.316608
Multi thread run 4; n = 15 | Time: 17.530316
Best time: 13.492839
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe m 16
Multi thread run 0; n = 16 | Time: 13.368078
Multi thread run 1; n = 16 | Time: 16.759268
Multi thread run 2; n = 16 | Time: 17.346903
Multi thread run 3; n = 16 | Time: 17.352960
Multi thread run 4; n = 16 | Time: 17.268071
Best time: 13.368078
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release> .\MatrixMultiplicationDemo.exe m 17
Multi thread run 0; n = 17 | Time: 14.811779
Multi thread run 1; n = 17 | Time: 17.238525
Multi thread run 2; n = 17 | Time: 17.213776
Multi thread run 3; n = 17 | Time: 17.484404
Multi thread run 4; n = 17 | Time: 17.444418
Best time: 14.811779
PS J:\repos\Parallel computing\MatrixMultiplication\x64\Release>
```



Порівняння часу виконання для однопоточної та багатопоточної реалізацій

Висновки

Спостерігаємо різке зменшення часу при n in [1; 4]
На цьому відрізку час роботи багатопоточної реалізації приблизно рівний часу однопоточної реалізації, поділеної на кількість потоків.

На відрізку [4; 8] час все ще зменшується, але дуже повільно. Загалом графік схожий на такий, що асимптотично прямує до деякої межі.
Починаючи з n = 8, (можливо), спостерігаємо дію Amdahl's law.

Різке збільшення часу при n = 10 в порівнянні з n = 9, n = 11 пов’язане із особливостями машини, на якій виконувались дослідження (тротлінг).
Також дія тротлінгу помітна при дослідженні не тільки кращого часу, але часу інших спроб: майже в усіх випадках кращою спробою є перша спроба.

Найкращий час виконання було отримано при n = 8.
Таким чином, встановлення кількості потоків значенню, що перевищує кількість фізичних потоків CPU (у випадку Intel Core i5-8250U—8 потоків) не має сенсу.

Реалізація завдання 2 (код)

Посилання на github репозиторій: https://github.com/Bohdan628318ylypchenko/parallel-programming-lab2.git

mmatrix.h

```
#pragma once

/// <summary>
/// Single thread matrix multiplication.
/// </summary>
/// <param name="n"> row count of a </param>
/// <param name="m"> column count of a = row count of b </param>
/// <param name="k"> column count of b </param>
/// <param name="a"> matrix a as 2d pointer </param>
/// <param name="b"> matrix b as 2d pointer </param>
/// <param name="c"> matrix c as 2d pointer to write result in </param>
void mmatrix_1t(int n, int m, int k,
                const double * const restrict * const restrict a, const double * const restrict * const restrict b,
                double * restrict * restrict * restrict c);


/// <summary>
/// Multi thread matrix multiplication.
/// </summary>
/// <param name="n"> row count of a </param>
/// <param name="m"> column count of a = row count of b </param>
/// <param name="k"> column count of b </param>
/// <param name="a"> matrix a as 2d pointer </param>
/// <param name="b"> matrix b as 2d pointer </param>
/// <param name="c"> matrix c as 2d pointer to write result in </param>
void mmatrix_mt(int n, int m, int k,
                const double * const restrict * const restrict a, const double * const restrict * const restrict b,
                double * restrict * restrict * restrict c);
```

main.c

```
#include "mmatrix.h"

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <float.h>

#define N_S 4
#define M_S 5
#define K_S 6

#define N_P 1024
#define M_P 2048
#define K_P 3072

#define USAGE "Usage: [v]alidation | [s]ingle-threaded | [m]ulti-threaded thread_count: int"

#define RUN_COUNT 5
#define OUT_SINGLE_THREAD "out-s.txt"
#define OUT_MULTI_THREAD "out-m.txt"

static void mmatrix_validation1(void(*mmatrix)(int, int, int, double **, double **, double ***));
static void mmatrix_validation2(void(*mmatrix)(int, int, int, double **, double **, double ***));
static void mmatrix_validation(void(*mmatrix)(int, int, int, double **, double **, double ***),
                               int n, int m, int k,
                               double ** a, double ** b, double ** c,
                               double ** expected);

static void mmatrix_demo_performance(void(*mmatrix)(int, int, int, double **, double **, double ***), char * outname);

static double ** matrix_alloc(int n, int m);
static void matrix_free(double ** matrix, int n);

static double min_time = DBL_MAX;

int main(int argc, char ** argv)
{
	if (argc < 2)
	{
		puts(USAGE);
		return EXIT_SUCCESS;
	}

	switch (argv[1][0])
	{
		case 'v':

			puts("=== Single thread example 1 ===");
			mmatrix_validation1(mmatrix_1t);
			putchar('\n');
			puts("=== Single thread example 2 ===");
			mmatrix_validation2(mmatrix_1t);
			putchar('\n');
			puts("=== Multi thread example 1 ===");
			mmatrix_validation1(mmatrix_mt);
			putchar('\n');
			puts("=== Multi thread example 2 ===");
			mmatrix_validation2(mmatrix_mt);

			break;
		case 's':

			for (int i = 0; i < RUN_COUNT; i++)
			{
				printf("Single thread run %d | ", i);
				mmatrix_demo_performance(mmatrix_1t, OUT_SINGLE_THREAD);
			}
			printf("Best time: %lf\n", min_time);

			break;

		case 'm':

			if (argc != 3)
			{
				puts(USAGE);
				return EXIT_SUCCESS;
			}

			int n = atoi(argv[2]);
			if (n <= 0)
			{
				printf("Invalid thread count: %s", argv[2]);
				return EXIT_SUCCESS;
			}

			omp_set_num_threads(n);
			for (int i = 0; i < RUN_COUNT; i++)
			{
				printf("Multi thread run %d; n = %d | ", i, n);
				mmatrix_demo_performance(mmatrix_mt, OUT_MULTI_THREAD);
			}
			printf("Best time: %lf\n", min_time);

			break;

		default:
			puts(USAGE);
			break;
	}

	return EXIT_SUCCESS;
}

static void mmatrix_validation1(void(*mmatrix)(int, int, int, double **, double **, double ***))
{
	// Initialize a
	double ** a = matrix_alloc(N_S, M_S);
	for (int i = 0; i < N_S; i++)
	{
		for (int j = 0; j < M_S; j++)
		{
			a[i][j] = i + j;
		}
	}

	// Initialize b
	double ** b = matrix_alloc(M_S, K_S);
	for (int i = 0; i < M_S; i++)
	{
		for (int j = 0; j < K_S; j++)
		{
			b[i][j] = i * j + i + j;
		}
	}

	// Initialize c
	double ** c = matrix_alloc(N_S, K_S);

	// Initialize expected
	double ** expected = matrix_alloc(N_S, K_S);
	expected[0][0] = 30; expected[0][1] = 70; expected[0][2] = 110; expected[0][3] = 150; expected[0][4] = 190; expected[0][5] = 230;
	expected[1][0] = 40; expected[1][1] = 95; expected[1][2] = 150; expected[1][3] = 205; expected[1][4] = 260; expected[1][5] = 315;
	expected[2][0] = 50; expected[2][1] = 120; expected[2][2] = 190; expected[2][3] = 260; expected[2][4] = 330; expected[2][5] = 400;
	expected[3][0] = 60; expected[3][1] = 145; expected[3][2] = 230; expected[3][3] = 315; expected[3][4] = 400; expected[3][5] = 485;

	// Validate
	mmatrix_validation(mmatrix, N_S, M_S, K_S, a, b, c, expected);

	// Free resources
	matrix_free(a, N_S);
	matrix_free(b, M_S);
	matrix_free(c, N_S);
	matrix_free(expected, N_S);
}

static void mmatrix_validation2(void(*mmatrix)(int, int, int, double **, double **, double ***))
{
	// Initialize a
	double ** a = matrix_alloc(N_S, N_S);
	a[0][0] = 1; a[0][1] = 1; a[0][2] = 1; a[0][3] = -1;
	a[1][0] = -5; a[1][1] = -3; a[1][2] = -4; a[1][3] = 4;
	a[2][0] = 5; a[2][1] = 1; a[2][2] = 4; a[2][3] = -3;
	a[3][0] = -16; a[3][1] = -11; a[3][2] = -15; a[3][3] = 14;

	// Initialize b
	double ** b = matrix_alloc(N_S, N_S);
	b[0][0] = 7; b[0][1] = -2; b[0][2] = 3; b[0][3] = 4;
	b[1][0] = 11; b[1][1] = 0; b[1][2] = 3; b[1][3] = 4;
	b[2][0] = 5; b[2][1] = 4; b[2][2] = 3; b[2][3] = 0;
	b[3][0] = 22; b[3][1] = 2; b[3][2] = 9; b[3][3] = 8;

	// Initialize c
	double ** c = matrix_alloc(N_S, N_S);

	// Initialize expected
	double ** expected = matrix_alloc(N_S, N_S);
	expected[0][0] = 1; expected[0][1] = 0; expected[0][2] = 0; expected[0][3] = 0;
	expected[1][0] = 0; expected[1][1] = 2; expected[1][2] = 0; expected[1][3] = 0;
	expected[2][0] = 0; expected[2][1] = 0; expected[2][2] = 3; expected[2][3] = 0;
	expected[3][0] = 0; expected[3][1] = 0; expected[3][2] = 0; expected[3][3] = 4;

	// Validate
	mmatrix_validation(mmatrix, N_S, N_S, N_S, a, b, c, expected);

	// Free resources
	matrix_free(a, N_S);
	matrix_free(b, N_S);
	matrix_free(c, N_S);
	matrix_free(expected, N_S);
}

static void mmatrix_validation(void(*mmatrix)(int, int, int, double **, double **, double ***),
                               int n, int m, int k,
                               double ** a, double ** b, double ** c,
                               double ** expected)
{
	// Multiply
	double s_time = omp_get_wtime();
	mmatrix(n, m, k, a, b, &c);
	double e_time = omp_get_wtime();

	// Assertion
	printf("Time: %lf\n", e_time - s_time);
	for (int i = 0; i < n; i++)
	{
		for (int j = 0; j < k; j++)
		{
			printf("| %lf ", c[i][j]);
			if (expected[i][j] != c[i][j])
			{
				printf("\nAssert failed for [%d][%d]\n", i, j);
				break;
			}
		}
		puts("|\n");
	}
}

static void mmatrix_demo_performance(void(*mmatrix)(int, int, int, double **, double **, double ***), char * outname)
{
	// Initialize a
	double ** a = matrix_alloc(N_P, M_P);
	for (int i = 0; i < N_P; i++)
	{
		for (int j = 0; j < M_P; j++)
		{
			a[i][j] = i + j;
		}
	}

	// Initialize b
	double ** b = matrix_alloc(M_P, K_P);
	for (int i = 0; i < M_P; i++)
	{
		for (int j = 0; j < K_P; j++)
		{
			b[i][j] = i + j;
		}
	}

	// Initialize c
	double ** c = matrix_alloc(N_P, K_P);

	// Multiply
	double s_time = omp_get_wtime();
	mmatrix(N_P, M_P, K_P, a, b, &c);
	double e_time = omp_get_wtime();
	double time = e_time - s_time;

	// Assertion
	printf("Time: %lf\n", time);

	// Save time
	if (min_time > time)
		min_time = time;

	// Save result
	FILE * f;
	fopen_s(&f, outname, "w");
	for (int i = 0; i < N_P; i++)
	{
		for (int j = 0; j < K_P; j++)
		{
			fprintf(f, "| %lf ", c[i][j]);
		}
		fputs("|\n", f);
	}
	fputc('\n', f);
	fclose(f);

	// Free resources
	matrix_free(a, N_P);
	matrix_free(b, M_P);
	matrix_free(c, N_P);
}

static double ** matrix_alloc(int n, int m)
{
	double ** matrix = (double **)malloc(n * sizeof(double *));
	for (int i = 0; i < n; i++)
	{
		matrix[i] = (double *)malloc(m * sizeof(double));
	}
	return matrix;
}

static void matrix_free(double ** matrix, int row_count)
{
	for (int i = 0; i < row_count; i++)
	{
		free(matrix[i]);
	}
	free(matrix);
}
```

mmatrix.c

```
#include "pch.h"

#include "mmatrix.h"

#include <omp.h>

/// <summary>
/// Single thread matrix multiplication.
/// </summary>
/// <param name="n"> row count of a </param>
/// <param name="m"> column count of a = row count of b </param>
/// <param name="k"> column count of b </param>
/// <param name="a"> matrix a as 2d pointer </param>
/// <param name="b"> matrix b as 2d pointer </param>
/// <param name="c"> matrix c as 2d pointer to write result in </param>
void mmatrix_1t(int n, int m, int k,
                const double * const restrict * const restrict a, const double * const restrict * const restrict b,
                double * restrict * restrict * restrict c)
{
	for (int p1 = 0; p1 < n; p1++)
	{
		for (int p2 = 0; p2 < k; p2++)
		{
			(*c)[p1][p2] = 0;
			for (int i = 0; i < m; i++)
			{
				(*c)[p1][p2] += a[p1][i] * b[i][p2];
			}
		}
	}
}

/// <summary>
/// Multi thread matrix multiplication.
/// </summary>
/// <param name="n"> row count of a </param>
/// <param name="m"> column count of a = row count of b </param>
/// <param name="k"> column count of b </param>
/// <param name="a"> matrix a as 2d pointer </param>
/// <param name="b"> matrix b as 2d pointer </param>
/// <param name="c"> matrix c as 2d pointer to write result in </param>
void mmatrix_mt(int n, int m, int k,
                const double * const restrict * const restrict a, const double * const restrict * const restrict b,
                double * restrict * restrict * restrict c)
{
	/*
	 * p1              - private as 1st cycle variable in "for" construct
	 * p2, k           - private as vars declared inside parallel construct
	 * n, m, k, a, b, c - shared as vars declared outside of parallel construct
	*/
	int p1;
	#pragma omp parallel for schedule(static)
	for (p1 = 0; p1 < n; p1++)
	{
		for (int p2 = 0; p2 < k; p2++)
		{
			(*c)[p1][p2] = 0;
			for (int i = 0; i < m; i++)
			{
				(*c)[p1][p2] += a[p1][i] * b[i][p2];
			}
		}
	}
}
```