Akademia Nauk Stosowanych w Nowym Sączu Programowanie Współbieżne i Rozproszone		
Temat:		**spr nr 9**
Nazwisko i imię: Ciapała Tadeusz	**Ocena sprawozdania**	**Zaliczenie:**
Data wykonania ćwiczenia:	**Grupa: P3**	

1) KODY:

```
#include <cstdio>
#include <cstdint>
#include <cstdlib>
#include <chrono>
#include <assert.h>
#include <time.h>

//matrix * vector

#define MATRIX_H 30000
#define MATRIX_W 30000
#define VECTOR_S 30000
#define MUL_TIME 25

uint16_t** matrix;
uint16_t* vector;
uint16_t* result;

int32_t i;
int32_t k;

int main() {
    srand(time(NULL));

    //check if vector size == matrix width
    assert(MATRIX_W == VECTOR_S);

    //alloc matrix
    matrix = (uint16_t**)new uint16_t * [MATRIX_H];
    for (i = 0; i < MATRIX_H; i++)
        matrix[i] = new uint16_t[MATRIX_W];

    //alloc vectors
    vector = (uint16_t*)new uint16_t[VECTOR_S];
    result = (uint16_t*)new uint16_t[VECTOR_S];

    //fill matrix random data normal way
    auto start = std::chrono::high_resolution_clock::now();
    for (i = 0; i < MATRIX_H; i++) {
        for (k = 0; k < MATRIX_W; k++) {
            matrix[i][k] = (uint16_t)(rand() % 100);
        }
    }

    //fill vector random data
    for (i = 0; i < VECTOR_S; i++) {
        vector[i] = (uint16_t)(rand() % 100);
    }

    auto end = std::chrono::high_resolution_clock::now();

    printf("Fill in %llu miliseconds\n",
        std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count());
    //normal execution
    start = std::chrono::high_resolution_clock::now();
    for (uint32_t p = 0; p < MUL_TIME; p++) {
        for (i = 0; i < MATRIX_H; i++) {
            for (k = 0; k < MATRIX_W; k++) {
                result[i] += matrix[i][k] * vector[k];
            }
        }
    }
    end = std::chrono::high_resolution_clock::now();

    printf("Calculated normal way in %llu miliseconds\n",
        std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count());

    start = std::chrono::high_resolution_clock::now();
#pragma omp parallel for default(shared) private(i, k)
    for (i = 0; i < MATRIX_H; i++) {
        for (k = 0; k < MATRIX_W; k++) {
            matrix[i][k] = (uint16_t)(rand() % 100);
        }
    }

#pragma omp parallel for default(shared) private(i)
    for (i = 0; i < VECTOR_S; i++) {
        vector[i] = (uint16_t)(rand() % 100);
    }
    end = std::chrono::high_resolution_clock::now();

    printf("Fill parallel way in %llu miliseconds\n",
        std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count());

    start = std::chrono::high_resolution_clock::now();
    for (uint32_t p = 0; p < MUL_TIME; p++) {
#pragma omp parallel for shared(matrix, vector, result) private(i, k)
        for (i = 0; i < MATRIX_H; i++) {
            for (k = 0; k < MATRIX_W; k++) {
                result[i] += matrix[i][k] * vector[k];
            }
        }
    }
    end = std::chrono::high_resolution_clock::now();

    printf("Calculated parallel way in %llu miliseconds\n",
        std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count());

    //free memory
    delete[] vector;
    delete[] result;

    for (i = 0; i < MATRIX_H; i++)
        delete[] matrix[i];

    delete[] matrix;

    return 0;
}
```

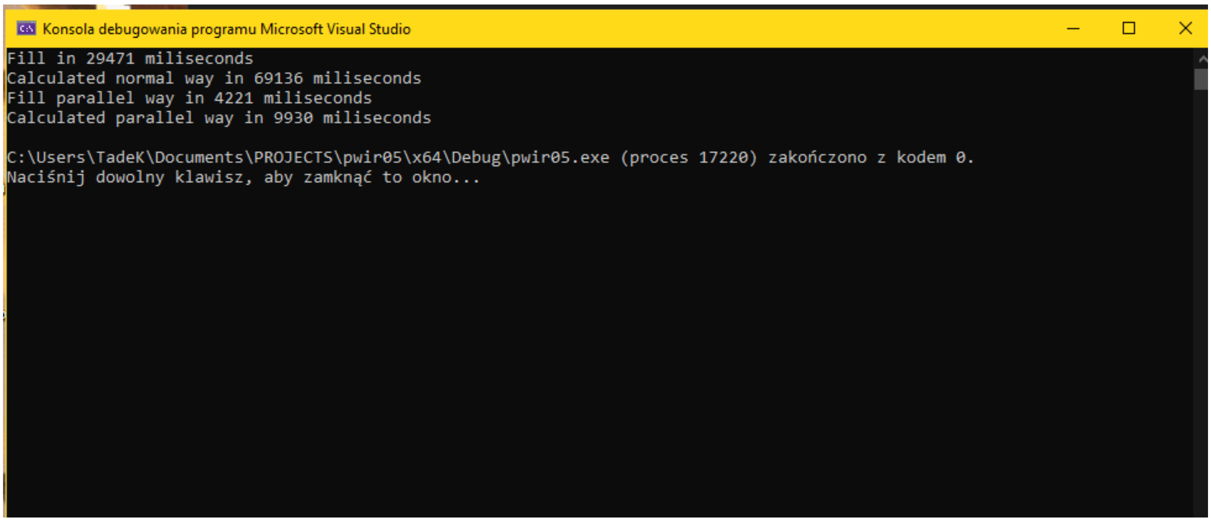
```
#include <cstdio>
#include <cstdint>
#include <cstdlib>
#include <chrono>
#include <assert.h>
#include <time.h>

//matrix * vector

#define MATRIX_H 30000
#define MATRIX_W 30000
#define VECTOR_S 30000
#define MUL_TIME 25

uint16_t** matrix;
uint16_t* vector;
uint16_t* result;
```

```
int32_t i;
int32_t k;

int main() {
    srand(time(NULL));
    //check if vector size == matrix width
    assert(MATRIX_W == VECTOR_S);

    //alloc matrix
    matrix = (uint16_t**)new uint16_t * [MATRIX_H];
    for (i = 0; i < MATRIX_H; i++)
        matrix[i] = new uint16_t[MATRIX_W];

    //alloc vectors
    vector = (uint16_t*)new uint16_t[VECTOR_S];
    result = (uint16_t*)new uint16_t[VECTOR_S];

    //fill matrix random data normal way
    for (i = 0; i < MATRIX_H; i++) {
        for (k = 0; k < MATRIX_W; k++) {
            matrix[i][k] = (uint16_t)(rand() % 100);
        }
    }

    //fill vector random data
    for (i = 0; i < VECTOR_S; i++) {
        vector[i] = (uint16_t)(rand() % 100);
        result[i] = 0;
    }

    //schedule(how to split iteration, [chunk size])
    //let user to controll iteration split between threads
    //chunk - how many iteration counts every chunk
    //first arg:
    //static:
    //  -iterations split on equal chunk of specified size or iterations/threads
    //  -chunk is mapped statically to thread
    //dynamic:
    //  -like static
    //  -but chunks are mapped dynamically
    //guided:
    //  -like dynamic
    //  -chunk count decrease during processing
    //runtime:
    //  -both argument are specified at runtime base on OMP_SCHEDULE var

#pragma omp parallel for default(shared) private(i, k)
    for (i = 0; i < MATRIX_H; i++) {
        for (k = 0; k < MATRIX_W; k++) {
            matrix[i][k] = (uint16_t)(rand() % 100);
        }
    }

#pragma omp parallel for default(shared) private(i)
    for (i = 0; i < VECTOR_S; i++) {
        vector[i] = (uint16_t)(rand() % 100);
    }

    auto start = std::chrono::high_resolution_clock::now();
#pragma omp parallel for schedule(static) shared(matrix, vector, result) private(i, k)
    for (uint32_t p = 0; p < MUL_TIME; p++) {
#pragma omp parallel for schedule(static) shared(matrix, vector, result) private(i, k)
        for (i = 0; i < MATRIX_H; i++) {
            for (k = 0; k < MATRIX_W; k++) {
                result[i] += matrix[i][k] * vector[k];
            }
        }
    }
    auto end = std::chrono::high_resolution_clock::now();

    printf("Calculated parallel static way in %llu miliseconds\n",
        std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count());

    start = std::chrono::high_resolution_clock::now();
    for (uint32_t p = 0; p < MUL_TIME; p++) {
```

```
#pragma omp parallel for schedule(static, MATRIX_H/10) shared(matrix, vector, result)
private(i, k)
        for (i = 0; i < MATRIX_H; i++) {
            for (k = 0; k < MATRIX_W; k++) {
                result[i] += matrix[i][k] * vector[k];
            }
        }
    }
    end = std::chrono::high_resolution_clock::now();

    printf("Calculated parallel static N(MATRIX_H/10) way in %llu miliseconds\n",
        std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count());

    start = std::chrono::high_resolution_clock::now();
    for (uint32_t p = 0; p < MUL_TIME; p++) {
#pragma omp parallel for schedule(dynamic, MATRIX_H/10) shared(matrix, vector, result)
private(i, k)
        for (i = 0; i < MATRIX_H; i++) {
            for (k = 0; k < MATRIX_W; k++) {
                result[i] += matrix[i][k] * vector[k];
            }
        }
    }
    end = std::chrono::high_resolution_clock::now();

    printf("Calculated parallel dynamic N(MATRIX_H/10) way in %llu miliseconds\n",
        std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count());

    start = std::chrono::high_resolution_clock::now();
    for (uint32_t p = 0; p < MUL_TIME; p++) {
#pragma omp parallel for schedule(guided, MATRIX_H/10) shared(matrix, vector, result)
private(i, k)
        for (i = 0; i < MATRIX_H; i++) {
            for (k = 0; k < MATRIX_W; k++) {
                result[i] += matrix[i][k] * vector[k];
            }
        }
    }
    end = std::chrono::high_resolution_clock::now();

    printf("Calculated parallel guided N(MATRIX_H/10) way in %llu miliseconds\n",
        std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count());

    start = std::chrono::high_resolution_clock::now();
    for (uint32_t p = 0; p < MUL_TIME; p++) {
#pragma omp parallel for schedule(runtime) shared(matrix, vector, result) private(i, k)
        for (i = 0; i < MATRIX_H; i++) {
            for (k = 0; k < MATRIX_W; k++) {
                result[i] += matrix[i][k] * vector[k];
            }
        }
    }
    end = std::chrono::high_resolution_clock::now();

    printf("Calculated parallel runtime way in %llu miliseconds\n",
        std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count());

    //free memory
    delete[] vector;
    delete[] result;

    for (i = 0; i < MATRIX_H; i++)
        delete[] matrix[i];

    delete[] matrix;

    return 0;
}
```

```
#include <cstdio>
#include <cstdint>
#include <cstdlib>
#include <chrono>
#include <assert.h>
#include <time.h>
```

```
#define MATRIX_H 30000
#define MATRIX_W 30000

//operators
//+
//-
//*
//&
//|
//^
//&&
//||

uint8_t** matrix;

uint32_t sumMatrix() {
    uint32_t sum = 0;

    for (uint32_t i = 0; i < MATRIX_H; i++) {
        for (uint32_t k = 0; k < MATRIX_W; k++) {
            sum += matrix[i][k];
        }
    }

    return sum;
}

uint32_t sumMatrixParallel() {
    uint32_t sum = 0;
    int32_t i;
    int32_t k;

#pragma omp parallel for shared(matrix) private(i, k) reduction(+ : sum)
    for (i = 0; i < MATRIX_H; i++) {
        for (k = 0; k < MATRIX_W; k++) {
            sum = sum + matrix[i][k];
        }
    }

    return sum;
}

int main() {
    srand(time(NULL));

    //alloc matrix
    matrix = (uint8_t**)new uint8_t * [MATRIX_H];
    for (uint32_t i = 0; i < MATRIX_H; i++)
        matrix[i] = new uint8_t[MATRIX_W];

    //fill matrix random data normal way
    for (uint32_t i = 0; i < MATRIX_H; i++) {
        for (uint32_t k = 0; k < MATRIX_W; k++) {
            matrix[i][k] = (uint16_t)(rand() % 10);
        }
    }

    auto start = std::chrono::high_resolution_clock::now();
    uint32_t sum = sumMatrix();
    auto end = std::chrono::high_resolution_clock::now();

    printf("Sum calculated normal way: %u in time: %llu ms\r\n", sum,
        std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count());

    start = std::chrono::high_resolution_clock::now();
    sum = sumMatrixParallel();
    end = std::chrono::high_resolution_clock::now();

    printf("Sum calculated parralel way: %u in time: %llu ms\r\n", sum,
        std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count());

    for (uint32_t i = 0; i < MATRIX_H; i++) delete[] matrix[i];
    delete[] matrix;
    return 0;
}
```

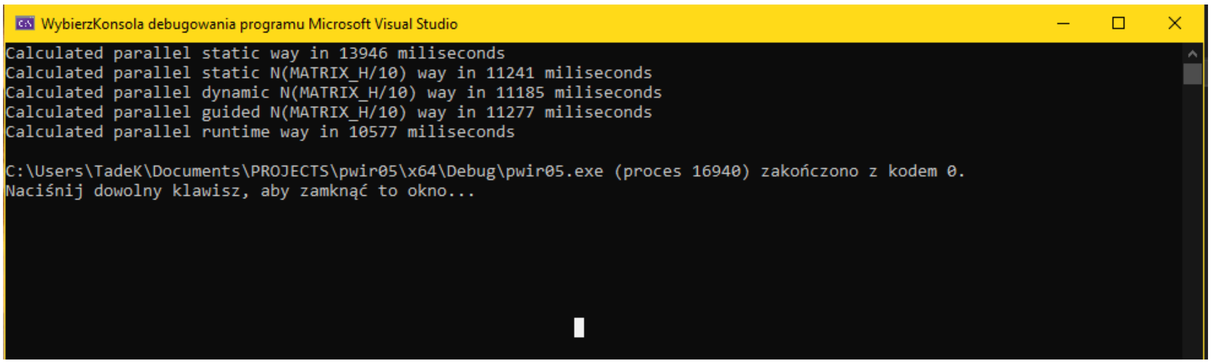


```
Fill in 29471 miliseconds
Calculated normal way in 69136 miliseconds
Fill parallel way in 4221 miliseconds
Calculated parallel way in 9930 miliseconds

C:\Users\TadeK\Documents\PROJECTS\pwir05\x64\Debug\pwir05.exe (proces 17220) zakończono z kodem 0.
Naciśnij dowolny klawisz, aby zamknąć to okno...
```

Kod PWIR_03_00.cpp

2) Zadania i rozwiązania:

Zadanie pierwsze polegało na przetestowaniu każdej z opcji schedule i wykonaniu pomiarów dla każdej dostępnej opcji. Następnie należało wyciągnąć z nich średnią. Wyniki zapisać i udokumentować.



```
Calculated parallel static way in 13946 miliseconds
Calculated parallel static N(MATRIX_H/10) way in 11241 miliseconds
Calculated parallel dynamic N(MATRIX_H/10) way in 11185 miliseconds
Calculated parallel guided N(MATRIX_H/10) way in 11277 miliseconds
Calculated parallel runtime way in 10577 miliseconds

C:\Users\TadeK\Documents\PROJECTS\pwir05\x64\Debug\pwir05.exe (proces 16940) zakończono z kodem 0.
Naciśnij dowolny klawisz, aby zamknąć to okno...
```

Kod PWIR_03_01.cpp

static	**static N(MATRIX_H/10)**	**dynamic N(MATRIX_H/10)**	**guided N(MATRIX_H/10)**	**runtime**
13946	11241	11185	11277	10577
14825	11223	11256	11308	10740
14607	11177	11173	11264	10319
14058	11213	11291	11277	10196
14125	11280	11207	11199	10546

Tabela pomiarów: wyniki są w milisekundach

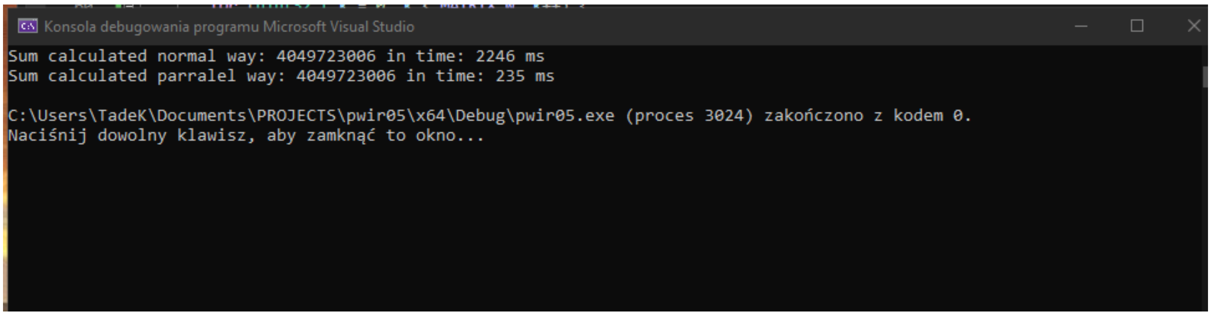
static	static N(MATRIX_H/10)	dynamic N(MATRIX_H/10)	guided N(MATRIX_H/10)	runtime
14312,2	11226,8	11222,4	11265	10475,6

Tabela średnich wartości pomiarów.

Różne metody podziału iteracji (static, dynamic, guided i runtime) dostępne w OpenMP mają zróżnicowany wpływ na czas wykonania programu. Metoda static, która dzieli iteracje na równe podzbiory, skutecznie skraca czas wykonania. Metody dynamic, guided i runtime charakteryzują się nieco krótszym czasem wykonania, przy czym dla mniejszych wartości parametru "chunk" czas jest dłuższy, a dla większych - krótszy. Wykorzystanie OpenMP do równoległego przetwarzania znacząco przyspiesza wykonanie programu w porównaniu do sekwencyjnej wersji.

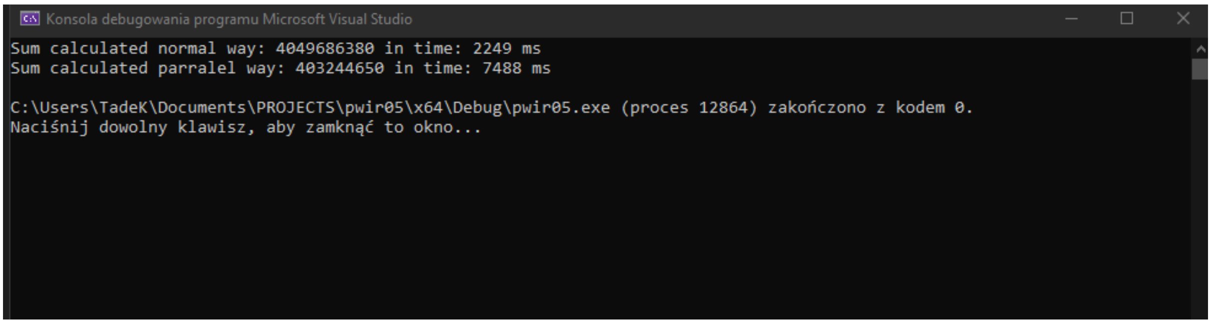
Zadanie drugie polegało na pracy z trzecim kodem. Trzeba było usunąć dyrektywę reduction i porównać wyniki.

PRZED:



```
Sum calculated normal way: 4049723006 in time: 2246 ms
Sum calculated parralel way: 4049723006 in time: 235 ms

C:\Users\TadeK\Documents\PROJECTS\pwir05\x64\Debug\pwir05.exe (proces 3024) zakończono z kodem 0.
Naciśnij dowolny klawisz, aby zamknąć to okno...
```

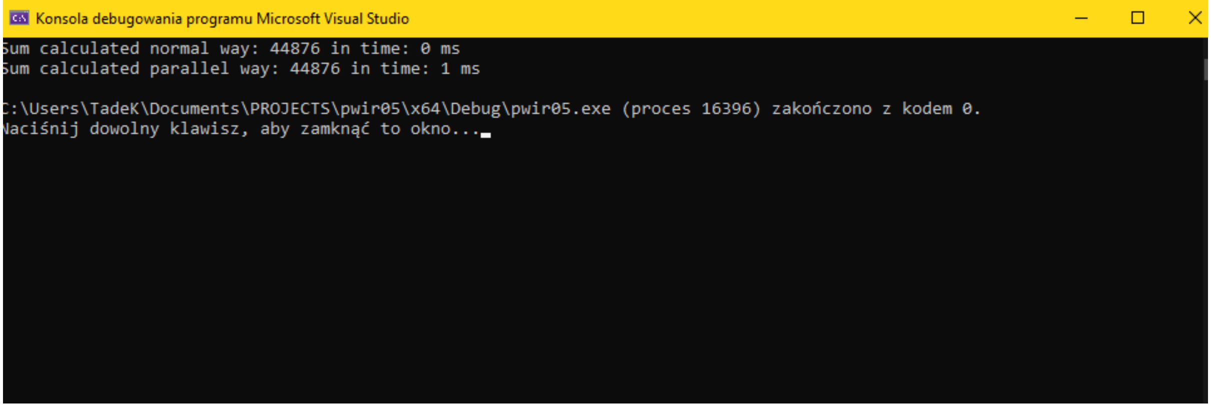
PO USUNIĘCIU:



```
Sum calculated normal way: 4049686380 in time: 2249 ms
Sum calculated parralel way: 403244650 in time: 7488 ms

C:\Users\TadeK\Documents\PROJECTS\pwir05\x64\Debug\pwir05.exe (proces 12864) zakończono z kodem 0.
Naciśnij dowolny klawisz, aby zamknąć to okno...
```

Bez uwzględnienia dyrektywy "reduction", każdy wątek tworzyłby własną instancję zmiennej "sum", co naraziłoby proces sumowania na potencjalne błędy. Łatwo zauważyć, że przedstawione wyniki wskazują na nieprawidłowe działanie kodu, prawdopodobnie spowodowane usterkami w środowisku wykonawczym. Aby zapewnić poprawne sumowanie w równoległym środowisku, nieodzowne jest skorzystanie z dyrektywy "reduction", która umożliwia właściwe grupowanie wyników.

Zadanie trzecie polegało napisaniu funkcji tworzącej wektor jednowymiarowy o wielkości 10 000 elementów. Należało go uzupełniać losowymi liczbami z zakresu od 0 do 10, a następnie obliczyć jego długość. Trzeba było porównać wyniki z wykorzystaniem zrównoleglenia lub bez.



```
Sum calculated normal way: 44876 in time: 0 ms
Sum calculated parallel way: 44876 in time: 1 ms

C:\Users\TadeK\Documents\PROJECTS\pwir05\x64\Debug\pwir05.exe (proces 16396) zakończono z kodem 0.
Naciśnij dowolny klawisz, aby zamknąć to okno...
```

Rezultaty są identyczne zarówno dla jednej, jak i drugiej metody przetwarzania, ponieważ sumowanie jest działaniem łącznym, a dyrektywa "reduction" zapewnia poprawne sumowanie w przypadku równoległej obróbki.