Sprawozdanie – programowanie równoległe i rozproszone

Ćwiczenie 2 – wątki w Java oraz Python

Adrian Nowosielski, Cezary Skorupski

1. Zarys problemu poruszonego w ćwiczeniu

Ćwiczenie 2 polegało na implementacji mnożenia dwóch macierzy wczytanych z pliku równolegle przy pomocy wielowątkowości w języku c oraz java oraz policzenie przy okazji normy Frobeniusa macierzy wyjściowej. Zasadniczym problemem mnożenia obu macierzy równolegle jest równe podzielenie pracy między wątkami oraz synchronizacja dostępu do zmiennej, w której przechowujemy aktualny wynik sumy normy Frobeniusa.

2. Sposób implementacji problemu w języku C

Do implementacji zadania w języku C użyliśmy biblioteki pthread, która pozwoliła nam tworzyć nowe wątki oraz używać mutexów(locków) do dodawnia poszczególnych wyników macierzy wyjściowej do ogólnego wyniku normy Frobeniusa. Na początku programu wczytujemy dane oraz sprawdzamy poprawność danych wejściowych (liczba danych, liczba elementów macierzy). Następnie przechodzimy do implementacji mnożenia równoległego:

- sprawdzamy, czy w ogóle możemy pomnożyć macierz w celu uzyskania macierzy wynikowej z dwóch wczytanych macierzy

- następnie alokujemy pamięć na nową macierz oraz przydzielamy zadania dla każdego wątka – zadania są przydzielanie na zasadzie przydzielenia komórek macierzy wyjściowej, które ma policzyć każdy z wątków

- następnie każdy wątek liczy swoją część macierzy wynikowej i na końcu zawsze dodaje do ogólnego wyniku normy Frobeniusa – sumę elementów, którą policzył

- na koniec oczywiście czekamy, aż wszystkie wątki skończą pracę i zwalniamy pamięć przydzieloną na poszczególne macierze, tablice itp.

Kod, który przedstawia podzielenie pracy pomiędzy wątkami oraz funkcję, którą wykonuje każdy wątek można przedstawić następująco:

```
int cells_per_thread = (cells_of_output_matrix) / number_of_threads;
thread_work_t **thread_works =
    calloc(number_of_threads, sizeof(thread_work_t *));

int starting_cell_work = 0;
int ending_cell_work;

for (int i = 0; i < number_of_threads; ++i) {
  thread_works[i] = malloc(sizeof(thread_work_t));
  thread_works[i]->starting_cell = i * cells_per_thread;
  thread_works[i]->ending_cell = i == number_of_threads - 1
                                 ? cells_of_output_matrix
                                 : (i + 1) * cells_per_thread;

  pthread_create(&thread_pool[i], NULL, single_multiply_task,
                 (void *)thread_works[i]);
}
```

```
void *single_multiply_task(void *args) {
  thread_work_t *thread_work = (thread_work_t *)args;
  double frobenius_thread_result = 0;
  int output_matrix_columns = multiply_result_matrix->number_of_columns;

  for (int cell = thread_work->starting_cell; cell < thread_work->ending_ce
       ++cell) {
    int row = cell / output_matrix_columns;
    int column = cell % output_matrix_columns;

    for (int k = 0; k < second_matrix->number_of_rows; ++k) {
      multiply_result_matrix->matrix_data[row][column] +=
          first_matrix->matrix_data[row][k] *
          second_matrix->matrix_data[k][column];
    }

    frobenius_thread_result +=
        pow(multiply_result_matrix->matrix_data[row][column], 2);
  }

  pthread_mutex_lock(&mutex);
  multiply_matrix_squared_sum += frobenius_thread_result;
  pthread_mutex_unlock(&mutex);

  pthread_exit(NULL);
}
```

3. Sposób implementacji problemu w języku Java

W zasadzie sposób implementacji nie różnił się bardzo od przytoczonego wcześniej języka C. Użyliśmy jedynie innego podejścia do tworzenia oprogramowania (bardziej obiektowo). Do utworzenia wątków użyliśmy klasy Thread oraz interfejsu Runnable, który pozwolił nam na utworzenie zbioru prac dla wątków i policzenia macierzy wyjściowej. Przykład podziału pracy oraz kodu, który wykonywał poszczególny wątek prezentuje się następująco. Do synchronizacji użyliśmy słowa kluczowego w Javie synchronized:

```
for (int i = 0; i < numberOfThreads; ++i) {
    final int currentIndex = i;

    final Runnable singleThreadMultiplyTask = () -> {
        double frobeniusThreadTaskSum = 0;   Nowosielski, 20 minutes ago • Added new
        final int startingThreadWorkCell = currentIndex * cellsPerThread;
        final int endingThreadWorkCell =
                currentIndex == numberOfThreads - 1 ? cellsOfOutputMatrix : (currentI

        for (int cell = startingThreadWorkCell; cell < endingThreadWorkCell; ++cell)
            int row = cell / outputMatrix.getNumberOfColumns();
            int column = cell % outputMatrix.getNumberOfColumns();
            double elementValue = 0;

            for (int k = 0; k < firstMatrix.getNumberOfColumns(); ++k) {
                elementValue += firstMatrix.getMatrixElement(row, k) * secondMatrix.g
            }

            outputMatrix.setMatrixElement(row, column, elementValue);
            frobeniusThreadTaskSum += Math.pow(elementValue, 2);
        }

        outputMatrix.addCalculatedFrobeniusTaskSumToSum(frobeniusThreadTaskSum);
    };

    threadPool[i] = new Thread(singleThreadMultiplyTask);
    threadPool[i].start();
```

4. Porównanie wątków w języku Java oraz C

Bez wątpienia można napisać, że pisanie programu w Javie było przyjemniejsze, ponieważ Java ma lepiej przystosowane interfejsy do pisania wielowątkowo (klasa Thread oraz ExecutorsService) – w c raczej trzeba to robić wszystko niskopoziomowo przez co może to się wydawać uciążliwe, ale jest wydajniejsze. Jeśli chodzi o synchronizację, to w Javie mamy dostępne słowo kluczowe synchronized lub klasy AtomicReference<T>, natomiast w C musimy się bawić z lockami oraz blokowaniem dostępu do wątków w bardziej niskopoziomowy sposób.

5. Porównanie szybkości działania

Rozmiar macierzy	Liczba wątków	**Czas Java [ms]**	**Czas C [ms]**
5x3 X 3x2	1	5	1
5x3 X 3x2	4	3	1
5x3 X 3x2	8	2	4
5x3 X 3x2	16	3	6
5x3 X 3x2	32	4	9
40x15 X 15x60	1	6	1
40x15 X 15x60	4	4.5	1.5
40x15 X 15x60	8	5	5
40x15 X 15x60	16	5	10
40x15 X 15x60	32	6	12

Wnioski:

Skuteczność języka C przy małej liczbie wątków - w przypadku małych macierzy i niewielkiej liczby wątków, program napisany w języku C wydaje się być bardziej wydajny. Wynika to z mniejszego nakładu czasu potrzebnego na obsługę wielowątkowości przez program w języku C w porównaniu z programem w Javie. C jest językiem bliższym sprzętowi, co pozwala na bardziej efektywne zarządzanie zasobami, szczególnie w przypadku małych obliczeń.

Wyższa wydajność Javy przy większej liczbie wątków - program napisany w Javie wydaje się być bardziej wydajny przy większej liczbie wątków. Jest to spowodowane tym, że wraz ze wzrostem liczby wątków, program w Javie nadal utrzymuje wysoką wydajność, prawdopodobnie dzięki bardziej zaawansowanej i zoptymalizowanej obsłudze wielowątkowości przez maszynę wirtualną Javy.

Wnioski przy większych macierzach: Można przypuszczać, że przy znacznie większych macierzach oba programy będą działały najlepiej przy większej liczbie wątków. Wraz z zwiększeniem rozmiaru danych, większa liczba wątków będzie miała więcej zadań do wykonania i zazwyczaj prowadzi do lepszej wydajności, pod warunkiem, że zarządzanie wątkami jest skuteczne.