Jakub Kuczkowiak

Algorytmy wielowątkowe

25 stycznia 2018

Streszczenie

Z nadzieją na lepsze jutro...

Spis treści

1. Wprowadzenie

W poniższej pracy przyjrzymy się bliżej algorytmom wielowątkowym. Na początku przedstawimy je bardzo teoretycznie. W dalszej części pracy porównamy je z algorytmami sekwencyjnymi, opiszemy podstawowe sposoby optymalizacji oraz zastanowimy się, jak dużą przewagę związaną z wykorzystaniem zasobów jesteśmy w stanie dzięki nim uzyskać. Praca w dużej części bazuje na teorii zawartej w [THC13] i jest jej swobodną interpretacją.

2. Opis

Aby móc mówić o algorytmach wielowątkowych, nazywanych też inaczej algorytmami równoległymi (ang. parallel algorithms) musimy najpierw zdefiniować nasz model obliczeń. Nasz komputer będzie teraz potrafił wykonywać wiele niezależnych od siebie rozkazów naraz. Algorytmy te mają ogromne znaczenie w całym przemyśle komputerowym. Chodzi tu zarówno o komputery klasy osobistej, o których możemy myśleć jak o komputerach z pojedynczym procesorem wielordzeniowym, w którym każdy rdzeń ma dostęp do wspólnej pamięci, jak i o ogromne serwery, w skład których wchodzą setki takich procesorów. W takich układach często mamy do czynienia z wątkami, które współdzielą pamięć. Każdy z nich posiada licznik rozkazów i wątek do procesora, aby można go było wykonać i wywłaszczyć z procesora, jeśli inny wątek powinien zostać uruchomiony.

2.1. Realizacja programistyczna

Do tej pory powstało mnóstwo bibliotek ułatwiających programowanie wielowątkowe. Każdy z systemów operacyjnych realizuje je na swój własny sposób. W niektórych językach programowania, takich jak na przykład C# mamy nawet specjalne pętle asynchroniczne for, które automatycznie rozdzielają różne iteracje między odpowiednie wątki.

3. Model obliczeń

3.1. Podstawy

Nasz model obliczeń równoległych będzie opierał się na tradycyjnym modelu obliczeń sekwencyjnych rozszerzonym o dwa udogodnienia: zagnieżdżona równoległość oraz pętle równoległe. Pozwoli to na dobre zrozumienie algorytmów, a ponadto te dwa sposoby są zaimplementowane prawie we wszystkich systemach operacyjnych i językach, które rozważamy. W ten sposób mamy trzy nowe słowa kluczowe w naszym pseudokodzie: *parallel* (równolegle), *spawn* (rozmnóż) oraz *sync* (synchronizuj). Słowo *parallel* będziemy umieszczać w pętli for, *spawn* będzie służył do uruchomienia danej instrukcji w osobnym wątku, a *sync* będzie oczekiwał na zakończenie wcześniej uruchomionych wątków.

3.2. Interpretacja grafowa

Każdy algorytm ma określoną interpretację grafową. O algorytmie myślimy zwykle, jak o liście kroków, które musimy wykonać. Pewne kroki występują przed innymi i tworzą uporządkowany ciąg instrukcji do wykonania. Oczywiście niektóre kroki możemy wykonywać niezależnie od siebie, podczas gdy w innych potrzebujemy wyników z wcześniejszych. W ten sposób o każdym problemie możemy myśleć jak o bijekcji pomiędzy listą kroków, a acyklicznym grafem skierowanym. Niech zbiór wierzchołków V będzie zawierał instrukcje do wykonania. W naszym grafie rysujemy krawędź z wierzchołka v_i do wierzchołka v_j wtedy i tylko wtedy gdy instrukcja o indeksie i musi być wykonana przed instrukcją o indeksie j. Przedstawienie naszego problemu w ten sposób pozwala nam na dokładne zrozumienie, które czynności możemy wykonywać równolegle. Prowadzi ona również do bardzo ważnego wniosku:

Twierdzenie 1 (Granica czasu wykonywania algorytmu równoległego)
Niech dany będzie sekwencyjny algorytm A oraz acykliczny graf skierowany G będący jego interpretacją grafową. Wtedy nie istnieje algorytm równoległy powstały z algorytmu A poprzez dodanie do niego słów kluczowych parallel, spawn oraz sync, który wykona się szybciej niż najdłuższa ścieżka w grafie G.

Dowód powyższego twierdzenia jest bardzo prosty. Wsród wszystkich dróg, istnieje najdłuższa droga w tym grafie oznaczająca ciąg instrukcji, którą pewien wątek musi wykonać i nie da się jej już podzielić na mniejsze, niezależne fragmenty.

3.3. Race problem

Rozszerzając pewien model obliczeniowy, chcielibyśmy oczywiście mieć z niego same korzyści. Tak jednak nie jest i pomimo, że potrafi on znacznie przyspieszać obliczenia, programista jest narażony na szereg problemów, które mogą wystąpić. Aby wyraźnie zaznaczyć, jak poważne mogą to być problemy, warto uświadomić sobie, że z powodu błędu z wyścigiem, maszyna do radioterapii nowotworów Therac-25 spowodowała śmierć trzech osób, obrażenia kilku innych, a w 2003 roku 50 milionów ludzi zostało bez prądu w Ameryce Północnej. Problem ten polega na jednoczesnym dostępie do danego adresu w pamięci przez dwa niezależne od siebie wątki. Wyobraźmy sobie następującą pętlę równoległą, której wszystkie iteracje uruchamiane są na kolejnych wątkach celem przyspieszenia działania takiego programu:

```
x = 0
parallel for i = 1 to 2
        x = x + 1
print x
```

Zadajmy proste pytanie: co wypisze ten program? Na pierwszy rzut oka chciałoby się powiedzieć, że będzie to oczywiście wartość 2. Z pewnością, wśród wielu uruchomień programu na pewno tak się zdarzy, ale nie jest to pewne! W wielu przypadkach zdarzy się, że x będzie miał wartość 1. Dlaczego tak się dzieje? Musimy sobie uświadomić, jak dokładnie działa operacja zwiększania o jeden w pamięci komputera. Najpierw do rejestru wczytywana jest wartość zmiennej x. Następnie, ta wartość w rejestrze jest zwiększana o jeden i na końcu ponownie zapisywana pod adresem x. Teraz jest już jasne, co może pójść nie tak. Załóżmy bowiem, że w tym samym momencie odczytana zostaje zmienna x przez oba wątki (równoległe iteracje pętli) i zapisywana do rejestrów r_1 i r_2. Wtedy oba rejestry mają wartość zero i każdy z nich jest niezależnie zwiększony o jeden, co powoduje, że na końcu przypisywana wartość do zmiennej x z dowolnego z tych rejestrów wynosi właśnie jeden.
Płynie stąd bardzo istotny wniosek. Musimy zawsze zapewnić, że metody wykonywane wewnątrz pętli równoległej są zawsze od siebie niezależne! Inaczej może wystąpić problem wyścigów.

3.4. Równoległość

Aby móc precyzyjniej opisywać korzyści wynikające z naszego algorytmu, musimy wprowadzić pojęcie równoległości. Definiujemy T_1 jako czas wykonywania naszego algorytmu sekwencyjnie, z użyciem pojedynczego procesora. Podobnie definiujemy T_{inf} jako czas wykonywania naszego algorytmu na nieskończonej ilości procesorów. Wtedy *równoległość* będzie oznaczać przyspieszenie jakie jesteśmy w stanie uzyskać porównując algorytm sekwencyjny i równoległy, i będziemy ją definiować jako stosunek $\frac{T_1}{T_{\text{inf}}}$.

4. Przykłady

4.1. Mergesort

4.1.1. Algorytm sekwencyjny

W poniższym dziale zastanowimy się jak usprawnić Mergesort i dokładnie przeanalizujemy jego nową złożoność obliczeniową. Na początek przypomnijmy tradycyjną implementację Mergesort'a:

```
1 Merge-Sort(T, l, r)
2     if (l < r)
3         q = (l + r) / 2 // podłoga
4         Merge-Sort(T, l, q)
5         Merge-Sort(T, q + 1, r)
6         merge(T, l, q, r)
```

Poniższa implementacja wykorzystuje funkcję *merge*, która przyjmuje tablicę oraz jej punkt podziału q.

Twierdzenie 2 (Poprawność i czas działania sekwencyjnego Merge-Sort)
Niech n = r - l. Merge-Sort poprawnie sortuje tablicę T w przedziale $[l, r]$ *w czasie* $\Theta(n \log n)$.

Dowód:
Najpierw udowodnimy poprawność powyższego algorytmu przez indukcję. Przypadek bazowy sprawdzany jest w drugim wierszu. Jeśli bowiem $l \geq r$, to tablica do posortowania jest pusta, więc nie trzeba nic robić. Dla kroku indukcyjnego załóżmy, że procedura $Merge - Sort$ poprawnie sortuje tablicę T dla wszystkich przedziałów długości mniejszej niż n. Zmienna q wyznacza wówczas środek tablicy. Oczywiście $q = (l + r)/2 < r$, więc wywołania w wierszu czwartym i piątym poprawnie sortują tablicę T w przedziałach [l, q] oraz [q + 1, r]. Teraz wywoływana jest procedura *merge*, która poprawnie scala te dwie posortowane już tablice w jedną, co kończy dowód poprawności.

Aby udowodnić czas działania, zauważmy, że otrzymujemy następujące równanie rekurencyjne opisujące Merge-Sort:
$T(n) = 2T(n/2) + \Theta(n)$,
gdyż wewnątrz procedury mamy dwa wywołania rekurencyjne dla lewej i prawej części tablicy, a na końcu mamy funkcję *merge*, który scala dwie tablice w czasie liniowym. Indukcyjnie łatwo sprawdzić, że równanie to rozwiązuje się do Θ(n log n).

4.1.2. Prosty algorytm równoległy

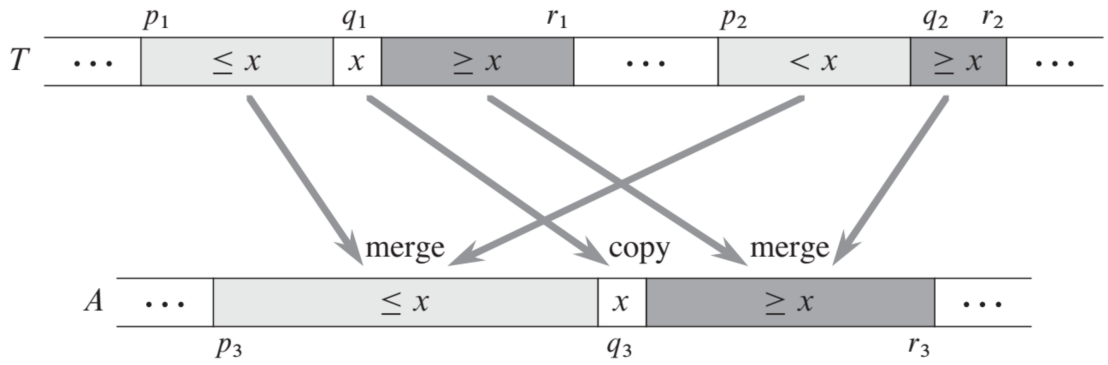
Pokażemy teraz, jak w niektórych przypadkach można łatwo zmodyfikować metodę sekwencyjną na metodę równoległą. Wiemy, że nasze wywołania dla lewej i prawej podtablicy są niezależne - wynik jednego z nich nie ma wpływu na drugi, więc bez problemu możemy wykonywać je w osobnych wątkach.

```
1 Merge-Sort-Async(T, l, r)
2     if (l < r)
3         q = (l + r) / 2 // podłoga
4         spawn Merge-Sort-Async(T, l, q)
5         Merge-Sort-Async(T, q + 1, r)
6         sync
7         merge(T, l, q, r)
```

Powyższa implementacja różni się jedynie dodaniem dwóch słów kluczowych: *spawn* oraz *sync* w wierszu czwartym i szóstym. Najpierw każemy programowi uruchomić pierwsze wywołanie w osobnym wątku, a na końcu przed wykonaniem *merge*, czekamy na wyniki z obu wątków. Dopiero wtedy możemy rozpocząć scalanie. Oczywiście dowód poprawności przebiega identycznie, co w poprzednim przypadku. Co jednak dzieje się ze złożonością czasową nowego algorytmu? Równanie rekurencyjne opisujące nowy czas działania to:
$T_{\text{inf}}(n) = T_{\text{inf}}(n/2) + \Theta(n)$,
gdyż teraz oba wywołania rekurencyjne mogą być wykonywane jednocześnie. Powyższa zależność rekurencyjna rozwiązuje się do $T_{\text{inf}}(n) = n + n/2 + n/4 + ... = n * (1 + 1/2 + 1/4 + ...) = 2n$, skąd $T_{\text{inf}}(n) = \Theta(n)$. Zatem równoległość powyższej metody to $\frac{T(n)}{T_{\text{inf}}(n)} = \frac{\Theta(nlogn)}{\Theta(n)} = \Theta(log\ n)$. Istotny jest wniosek z tej obserwacji. Kilka procesorów więcej będzie miało prawie liniowy wzrost wydajności, ale już 1000 razy procesorów więcej przekłada się jedynie na 8-krotny wzrost wydajności.

4.1.3. Rozwiązanie wzorcowe

Jak zoptymalizować nasz program? Czy da się coś jeszcze poprawić? Okazuje się, że tak. Naszym największym problemem jest procedura *merge*, która nadal jest sekwencyjna. Musimy zrobić coś, aby ta procedura była również oparta na technice $Divide-and-Conquer$. Stworzymy więc *merge*, który będzie można rozdzielić na dwa wątki.



Powyższy obrazek pochodzi z [THC13]. Wykonanie *merge* na dwóch posortowanych ciągach $T[p_1..r_1]$ oraz $T[p_2..r_2]$ do nowego ciągu $A[p_3..r_3]$. Niech $x = T[q_1]$ będzie medianą $T[p_1..r_1]$, a q_2 miejscem w $T[p_2..r_2]$, takim że x wpada pomiędzy $T[q_2 - 1]$ i $T[q_2]$. Podciągi $T[p_1..q_1 - 1]$ oraz $T[p_2..q_2 - 1]$ (lekko zacieniowane) są mniejsze lub równe x, a każdy element w podciągach $T[q_1 + 1..r_1]$ oraz $T[q_2 + 1..r_2]$ (mocno zacieniowane) wynosi co najmniej x. By dokonać *merge* obliczamy index q_3 gdzie x należy w ciągu $A[p_3..r_3]$, kopiujemy x do $A[q_3]$ i rekurencyjnie wykonujemy *merge* dla $T[p_1..q_1 - 1]$ z $T[p_2..q_2 - 1]$ do $A[p_3..q_3 - 1]$ i $T[q_1 + 1..r_1]$ z $T[q_2..r_2]$ do $A[q_3 + 1..r_3]$.
Pseudokod równoległego *merge*:

```
P-MERGE(T, p1, r1, p2, r2, A, p3)
1  n1 = r1 - p1 + 1
2  n2 = r2 - p2 + 1
3  if n1 < n2 // zapewniamy, że n1 >= n2
4          exchange p1 with p2
5          exchange r1 with r2
6          exchange n1 with n2
7  if n1 == 0 // oba puste?
8          return
9  else
10         q1 = (p1 + r1) / 2 // podłoga
11         q2 = BINARY-SEARCH(T[q1], T, p2, r2)
12         q3 = p3 + (q1 - p1) + (q2 - p2)
13         A[q3] = T[q1]
14         spawn P-MERGE(T, p1, q1 - 1, p2, q2 - 1, A, p3)
15         P-MERGE(T, q1 + 1, r1, q2, r2, A, q3 + 1)
16         sync
```

Przeanalizujemy teraz czas działania powyższej procedury. Mamy dwie części tablicy T o łącznej ilości elementów $n = n_1 + n_2$, gdzie n_1 oraz n_2 oznaczają odpowiednio ilości elementów w lewej i prawej części tablicy. Musimy teraz zastanowić się, jak bardzo źle dzieli się nasza tablica. W najgorszym przypadku mamy bowiem środek lewej części, czyli połowę jej elementów i żadnego z elementów z prawej części tablicy (lub symetrycznie wszystkie elementy prawej części), co oznacza, że możemy tę liczbę ograniczyć z góry przez $3/4n$. Daje nam to następującą zależność rekurencyjną opisującą najgorszy przypadek czasowy:
$T_{\text{inf}}(n) = T_{\text{inf}}(3/4n) + \Theta(log\ n)$, co rozwiązuje się do $T_{\text{inf}} = \Theta(log^2\ n)$.

4.1.4. Analiza złożoności czasowej wzorcowego Merge-Sort'a

Rekurencja opisująca czas działania Merge-Sort z ulepszoną procedurą *merge* wynosi:
$PMS_{\text{inf}}(n) = PMS_{\text{inf}}(n/2) + PM_{\text{inf}}(n) = PMS_{\text{inf}}(n/2) + \Theta(log^2 n)$,
co rozwiązuje się do: $PMS_{\text{inf}}(n) = \Theta(log^3 n)$.
Stąd nowa równoległość uzyskana w wyniku naszej optymalizacji wynosi:
$PMS_1(n)/PMS_{\text{inf}}(n) = \Theta(n\ log\ n)\ /\ \Theta(log^3\ n) = \Theta(n\ /\ log^2\ n)$.
Wniosek jest prosty. Odpowiednie przekształcenie metody na bardziej równoległą pozwoliło na znaczny zysk czasowy. Wcześniejsza, naiwna implementacja polegająca na dodaniu tylko dwóch słów, miała równoległość na poziomie $\Theta(log\ n)$. Przy dużej ilości procesorów zysk będzie bliższy liniowemu.

4.1.5. Problem z implementacją P-MERGE w wyniku kopiowania tablic

Chociaż pseudokod funkcji P-MERGE wydaje się prosty, to jednak mamy tutaj do czynienia z pewnymi problemami implementacyjnymi. W procedurze tej tworzymy bowiem nową tablicą. Nie możemy bazować na tablicy T i modyfikować jej w obu wątkach, gdyż procedury te nie muszą być niezależne. Musimy być zatem bardzo uważni przy kopiowaniu naszych tablic. Z [fas15] wynika, że zdecydowanie w tym aspekcie góruje funkcja *memcpy* i że wszelkie porównania należy dokonywać w wersji *release* naszego oprogramowania. Jednak jakiekolwiek kopiowanie pamięci sekwencyjne byłoby liniowe i wówczas nie osiągnęlibyśmy czasu podanego w rekurencji, gdyż zostałby on zdominowany przez kopiowanie. W tym celu musimy zrobić podobny trick, co w naszym asynchronicznym $Merge - Sort - Async$. Będziemy kopiować tablicę metodą dziel i zwyciężaj do nowej pamięci używając dostępnych wątków w systemie, co przy liczbie procesorów dążącej do nieskończoności prowadzi do pożądanej przez nas złożoności: $\Theta(log\ n)$.

5. Implementacja

Poniżej przedstawimy przykładową implementację w języku C++ przy użyciu wątków działającą pod system operacyjnym Linux. Program składa się z trzech plików i ich odpowiedników nagłówkowych.

5.1. main.cpp

```
void generateRandomArray(int* T, int n, int m)
{
    for (int i = 0; i < n; i++)
        T[i] = rand() % m;
}

void printArray(int* T, int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("%d", T[i]);
        if (i != n - 1)
            printf(", ");
    }

    printf("\n");
}

int main()
{
    for (int i = 1; i < 16; i++)
        copytest(i); // i = 0 copies with default threads number for CPU

    printf("\n---------------------------\n");

    for (int i = 1; i < 16; i++)
        mergetest(i); // i = 0 sorts with default threads number for CPU

    return 0;
}
```

Funkcja *generateRandomArray* służy do generowania losowej tablicy typu *int* o wartościach z przedziału od 0 do $m - 1$. Procedura *printArray* wypisuje zawartość tablicy na wyjście. Główna część programu uruchamiana na początku, czyli *main* składa się z dwóch testów: kopiowania oraz sortowania. Każdy z nich wywołujemy z parametrem i oznaczającym maksymalną ilość wątków na jakie będziemy się rozdzielać wewnątrz tych procedur.

5.2. copy.cpp

```
// copies dest into src from l to r
void* copy_async(void* args)
{
    copyparams_t _args = *(copyparams_t*)args;
    int* src = _args.src;
    int* dest = _args.dest;
    int l = _args.l;
    int r = _args.r;
    int threadsAvailable = _args.threadsAvailable;

    if (l == r)
    {
        dest[l] = src[l];
    }
    else if (l < r)
    {
        int q = l + (r - l) / 2;

        if (threadsAvailable < 1)
        {
            memcpy(&dest[l], &src[l], sizeof(int) * (r - l + 1));
            return nullptr;
        }

        // Spawn in thread
        copyparams_t args;
        args.src = src;
        args.dest = dest;
        args.l = l;
        args.r = q;
        args.threadsAvailable = threadsAvailable - 1;

        copyparams_t args2;
        args2.src = src;
        args2.dest = dest;
        args2.l = q + 1;
        args2.r = r;
        args2.threadsAvailable = threadsAvailable - 1;

        pthread_t thread;
        pthread_create(&thread, NULL, copy_async, &args);
        copy_async(&args2);

        // wait for the thread to finish before merging
        pthread_join(thread, NULL);
    }
}
```

Głównym sercem modułu *copy.cpp* jest funkcja *copy_ async*. W kroku bazowym, gdy zostaje nam jeden element dokonujemy przypisania wprost. Podobnie, gdy wyczerpiemy dostępne dla nas wątki, to wówczas wykonujemy proste kopiowanie za pomocą funkcji *memcpy*. W kroku indukcyjnym rozdzielamy tablicę na dwie połówki i kopiujemy równolegle lewą i prawą część tablicy. Oczywiście zadania te są od siebie niezależne. Funkcja ta bardzo przypomina implementację procedury *mergesort_ async* z tą różnicą, że tutaj kopiujemy, zamiast wykonywać *merge*. W tym pliku znajduje się również procedura *copytest*, która odpowiada za mierzenie czasu pomiędzy zwykłym kopiowaniem *memcpy*, a rozdzielaniem kopiowania tablicy między wątki naszej funkcji *copy_ async*. Ideą, która przyświecała stworzeniu tej funkcji, jest opisany wcześniej problem kopiowania tablicy.

5.3. mergesort.cpp

```
void* mergesort_async(void* args)//,int* T, unsigned l, unsigned r)
{
    params_t _args = *(params_t *)args;
    int* T = _args.T;
    int l = _args.l;
    int r = _args.r;
    int threadsAvailable = _args.threadsAvailable;

    if (l < r)
    {
        // This is the same as (l+r)/2 but avoids overflow for big data
        int q = l + (r - l) / 2;

        if (threadsAvailable < 1)
        {
            mergesort(T, l, q);
            mergesort(T, q + 1, r);
            return nullptr;
        }

        // Spawn in threads
        params_t args;
        args.T = T;
        args.l = l;
        args.r = q;
        args.threadsAvailable = threadsAvailable - 1;

        params_t args2;
        args2.T = T;
        args2.l = q + 1;
        args2.r = r;
        args2.threadsAvailable = threadsAvailable - 1;

        pthread_t thread;

        pthread_create(&thread, NULL, mergesort_async, &args);
        mergesort_async(&args2);

        // wait for the thread to finish before merging
        pthread_join(thread, NULL);

        merge(T, l, q, r);
    }
}
```

W tym pliku znajdują się również funkcje *merge* oraz *mergesort*, których implementacje można znaleźć w wielu artykułach. Skorzystaliśmy z gotowej implementacji *merge* pochodzącej z [mer]. Zajmiemy się omówieniem głównej funkcji *mergesort_ async*. Podobnie, jak w przypadku kopiowania, rozpatrujemy przypadek bazowy, gdy mamy do czynienia z tablicą pustą. Wtedy nie trzeba już nic sortować. Drugi przypadek dotyczy wyczerpania dostępnych nam wątków przydzielonych na początku. Wówczas wywołujemy już zwykły *mergesort* oraz kończymy działanie. W kroku indukcyjnym rozdzielamy tablicę na dwie połówki i przydzielamy je do dwóch wątków. Na koniec czekamy, aż oba zakończą działanie i dopiero wtedy wykonujemy *merge*. Funkcja *mergetest* porównuje działanie zwykłego *mergesort* z naszym asynchronicznym *mergesort_ async*.

6. Testowanie programu

6.1. Kompilacja programu i uruchamianie

Poniższy program należy kompilować pod kontrolą systemu Linux za pomocą pliku Makefile:

```
$ make
```

1: Kompilacja programu

Uruchamiamy go w następujący sposób:

```
$ ./main
```

2: Uruchamianie programu

6.2. Testowanie

Testowanie programu odbywa się na komputerze Lenovo YOGA 900 z procesorem Intel Core I7-6500U. Przykładowe uruchomienie programu dla tablicy o 100000000 elementach dało następujące wyniki:

```
test:~/Documents/projects/merge-sort$ ./main
Copy test using 1 threads
Time needed to copy using memcpy: 217428.000000
Time neeeded to copy using copy_async: 43687.000000

Copy test using 2 threads
Time needed to copy using memcpy: 82681.000000
Time neeeded to copy using copy_async: 89306.000000

Copy test using 3 threads
Time needed to copy using memcpy: 81435.000000
Time neeeded to copy using copy_async: 122158.000000

Copy test using 4 threads
Time needed to copy using memcpy: 196458.000000
Time neeeded to copy using copy_async: 144110.000000

Copy test using 5 threads
Time needed to copy using memcpy: 83142.000000
Time neeeded to copy using copy_async: 128212.000000

Copy test using 6 threads
Time needed to copy using memcpy: 278848.000000
Time neeeded to copy using copy_async: 180741.000000

Copy test using 7 threads
Time needed to copy using memcpy: 87669.000000
Time neeeded to copy using copy_async: 218758.000000

Copy test using 8 threads
Time needed to copy using memcpy: 152285.000000
Time neeeded to copy using copy_async: 239483.000000

Copy test using 9 threads
Time needed to copy using memcpy: 161784.000000
Time neeeded to copy using copy_async: 276739.000000

Copy test using 10 threads
Time needed to copy using memcpy: 172474.000000
Time neeeded to copy using copy_async: 483813.000000

Copy test using 11 threads
Time needed to copy using memcpy: 110845.000000
Time neeeded to copy using copy_async: 531619.000000

Copy test using 12 threads
Time needed to copy using memcpy: 95751.000000
Time neeeded to copy using copy_async: 546661.000000

Copy test using 13 threads
```

```
Time needed to copy using memcpy: 107334.000000
Time neeeded to copy using copy_async: 922793.000000

Copy test using 14 threads
Time needed to copy using memcpy: 309049.000000
Time neeeded to copy using copy_async: 979710.000000

Copy test using 15 threads
Time needed to copy using memcpy: 113264.000000
Time neeeded to copy using copy_async: 1443969.000000


_____________________________
Starting mergesort test using 1 threads
Time neeeded to sort using mergesort: 45099722.000000
Time needed to sort using mergsort_async: 44848191.000000

Starting mergesort test using 2 threads
Time neeeded to sort using mergesort: 36928764.000000
Time needed to sort using mergsort_async: 38637296.000000

Starting mergesort test using 3 threads
Time neeeded to sort using mergesort: 34747439.000000
Time needed to sort using mergsort_async: 49715933.000000

Starting mergesort test using 4 threads
Time neeeded to sort using mergesort: 34734030.000000
Time needed to sort using mergsort_async: 49381276.000000

Starting mergesort test using 5 threads
Time neeeded to sort using mergesort: 34719190.000000
Time needed to sort using mergsort_async: 49483907.000000

Starting mergesort test using 6 threads
Time neeeded to sort using mergesort: 34830090.000000
Time needed to sort using mergsort_async: 49629231.000000

Starting mergesort test using 7 threads
Time neeeded to sort using mergesort: 34987838.000000
Time needed to sort using mergsort_async: 49645821.000000

Starting mergesort test using 8 threads
Time neeeded to sort using mergesort: 34920566.000000
Time needed to sort using mergsort_async: 49520635.000000

Starting mergesort test using 9 threads
Time neeeded to sort using mergesort: 34857624.000000
Time needed to sort using mergsort_async: 49749060.000000

Starting mergesort test using 10 threads
Time neeeded to sort using mergesort: 35055392.000000
Time needed to sort using mergsort_async: 49781287.000000

Starting mergesort test using 11 threads
Time neeeded to sort using mergesort: 34968986.000000
Time needed to sort using mergsort_async: 51587691.000000

Starting mergesort test using 12 threads
Time neeeded to sort using mergesort: 34996784.000000
Time needed to sort using mergsort_async: 53030602.000000

Starting mergesort test using 13 threads
Time neeeded to sort using mergesort: 34920110.000000
Time needed to sort using mergsort_async: 47753078.000000

Starting mergesort test using 14 threads
Time neeeded to sort using mergesort: 34913834.000000
Time needed to sort using mergsort_async: 46959019.000000

Starting mergesort test using 15 threads
Time neeeded to sort using mergesort: 34986660.000000
Time needed to sort using mergsort_async: 45296899.000000
```

3: Przykładowe uruchomienie programu

Wyniki te są dosyć niepokojące. zdecydowanie zbyt duża ilość wątków wpływa negatywnie na szybkość procesu. Przy małej ilości (do czterech) wątków, program ten działa porównywalnie.

7. Podsumowanie

Programowanie wielowątkowe (równoległe) potrafi przysparzać wielu kłopotów, zarówno od strony teoretycznej, jak i od strony implementacyjnej. Powoduje to często wiele utrudnień dla pracy programisty. Musi on bowiem zadbać o poprawność takiego programu oraz przetestować go od strony wydajnościowej.

Literatura

[fas15] C/C++ tip: How to copy memory quickly. May 2015.

[mer] Merge Sort.

[THC13] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. *Wprowadzenie do algorytmów*, pages 791–832. Wydawnictwo Naukowe PWN, 2013.