Aleksandra Jarzyńska **136722** L8
Grzegorz Bryk **136686** L8
Termin zajęć: środa 9:45
Data: 27.04.2020 [*wymagany termin: 27.04.2020*]
WERSJA ~~PIERWSZA~~ DRUGA
mail: aleksandra.w.jarzynska@student.put.poznan.pl
grzegorz.bryk@student.put.poznan.pl

Przetwarzanie równoległe – laboratorium

Projekt nr 1: Analiza efektywności przetwarzanie równoległego realizowanego w komputerze równoległym z procesem wielordzeniowym z pamięcią współdzieloną na podstawie problemu znajdowania liczb pierwszych w podanym jako parametr przedziale.

1. Procesor: *i7-8750H*

Liczba procesorów fizycznych	Liczba procesorów logicznych	Liczba uruchamianych wątków w systemie	Typ procesora	Wielkość i organizacja pamięci podręcznych	Wielkość i organizacja bufora translacji adresów
6	12	2 * 6	CPU / Microprocessor	9 MB Intel® Smart Cache L1 384KB L2 1.5MB L3 9MB	64-byte Prefetching

System operacyjny	IDE	Wersja VTune
Windows 10	Visual Studio 2019	2020 Update 1

UPDATE WERSJA DRUGA: Teraz testy przeprowadzaliśmy na komputerze drugiej osoby z grupy, stąd nowa tabela:

Procesor: i5-7200U

Liczba procesorów fizycznych	Liczba procesorów logicznych	Liczba uruchamianych wątków w systemie	Typ procesora	Wielkość i organizacja pamięci podręcznych	Wielkość i organizacja bufora translacji adresów
2	4	2 * 2	CPU / Microprocessor	3 MB Intel® Smart Cache L1 128KB L2 512KB L3 3MB	64-byte Prefetching

System operacyjny	IDE	Wersja VTune
Windows 10	Visual Studio 2019	2020 Update 1

Potencjalne problemy efektywnościowe:

- **false sharing** – zjawisko, które jest odpowiedzialne za niezamierzone/fałszywe współdzielenie danych. Polega na tym, że wiele procesów modyfikuje zmienne, które znajdują się na tej samej linii danych pamięci podręcznej procesora. False sharing najłatwiej zaobserwować, kiedy użyjemy zmiennych typu **volatile**. Typ ten przestrzega przed tym, że zmienna może być modyfikowana i ~~w pewien sposób zapewnia użytkownikowi synchronizację~~ powstrzymuje kompilator optymalizujący przed pomijaniem zapisów do pamięci – broni przed odczytaniem nieaktualnej wartości takiej zmiennej, co jest wymuszone poprzez ~~aktualizację~~ unieważnienie kopii całej linii danych pamięci podręcznej w każdym procesie, który z niej korzysta – zaraz po zapisie do tej linii danych przez jakikolwiek z nich. Łatwo zauważyć, że każde żądanie dostępu będzie zajmowało znacznie więcej (o około rząd jednostki) czasu, niż w przypadku, kiedy nie używamy typu volatile, należy jednak pamiętać, że narażamy wtedy pozostałe procesy na odczytanie nieaktualnej wartości (w przypadku jeśli rzeczywiście korzystają z tej samej zmiennej). **False sharing** występuje, kiedy procesy korzystają z **tej samej linii danych** pamięci podręcznej, ale **modyfikują zmienne, które są niezależne** (zapis do zmiennej jest traktowany jako zapis do konkretnej linii danych, co w każdym procesie, który korzysta z tej linii wymusza ~~odświeżenie~~ wstrzymanie dostępu do danych do momentu sprowadzenia do pamięci podręcznej linii aktualnej).
- **Synchronizacja** – podobnie jak w opisanym wcześniej przypadku false sharingu (gdzie również mamy do czynienia z synchronizacją – dopiero po zapisie i uaktualnieniu wartości może nastąpić jej odczyt), synchronizacja jest realnym problemem efektywnościowym.

Dostęp synchroniczny do niektórych części pamięci jest wielokrotnie **niezbędny** do poprawnego funkcjonowania programu, jednocześnie jej zapewnienie (poprzez np. umieszczanie pewnych operacji w sekcji krytycznej) jest bardzo **kosztowne czasowo**.

Potencjalne problemy poprawnościowe:

- **wyścig** – warunkiem wystąpienia wyścigu jest dostęp dwóch wątków do tej samej (współdzielonej – shared) zmiennej w **tym samym czasie**. Pierwszy wątek odczytuje wartość zmiennej – jednocześnie drugi wątek odczytuje taką samą wartość. Oba wątki wykonują działania na tej wartości zmiennej, więc ten, który **skończy pracę później** (więc później nadpisze jej wartość) **wygrywa wyścig**. Mamy tutaj do czynienia z działaniem wątku, które nie przynosi żadnego wymiernego efektu, a zużywa ograniczone zasoby komputera.

Jak działa Intel Parallel Studio XE?

- Sposób zbierania informacji o efektywności przetwarzania: Parallel Studio składa się z kilku części komponentów, z których każdy wspomaga dewelopera na innej płaszczyźnie. Composer zawiera wysokowydajnościowy kompilator, biblioteki wydajnościowe oraz rozszerzenie do debuggera Visual Studio, które jest specjalnie przeznaczone dla równoległych programów. Parallel Inspector dostarcza informacji, które mogą pomóc programiście zlokalizować nieprawidłowości, przede wszystkim ułatwia znalezienie błędów, które polegają na błędnym użyciu synchronizacji, które mogą powodować wyścig lub deadlock, czy błędy związane z dostępem do pamięci.
- Znaczenie zebranych i prezentowanych w sprawozdaniu informacji:
  - Liczba cykli procesora w czasie wykonywania badanego kodu, liczba instrukcji kodu asemblera – pozwalają na obliczenie CPI (Clockticks per Instructions Retired) poprzez podzielenie liczby cykli przez liczbę instrukcji kodu. Wartość CPI jest wskaźnikiem tego, jak duże opóźnienia miały miejsce w trakcie przetwarzania kodu. Wyższe wartości oznaczają, że mieliśmy do czynienia z większymi opóźnieniami w systemie – więcej cykli procesora było potrzebnych do przetworzenia instrukcji.
  - udział procentowy wykorzystanych zasobów procesora do przetwarzania kodu – jak nazwa wskazuje. Często celem jest osiągnięcie tej wartości równej 100% i często (ale nie zawsze) idzie to w parze z maksymalnie zoptymalizowanym i zrównoleglonym kodem.
  - front-end bounds, back-end bound – te dwie miary odnotowują kolejno pierwszą i drugą część pracy, za którą odpowiedzialny jest rdzeń procesora. Front-end pobiera operacje, które są później wykonywane przez część back-end.
  - memory bound – informuje o tym, jak kwestie związane z podsystemem pamięciowym wpływają na wydajność. Mierzy, jaki ułamek gniazd procesora jest blokowany poprzez żądania ładowania/przechowywania instrukcji.
  - core bound – mierzy jak dużo kwestii niezwiązanych z pamięcią powodowało wąskie gardła.
  - effective physical core utilization – pokazuje procent średniej wydajności wszystkich rdzeni fizycznych, które są w systemie.
- Wykorzystywany tryb Intel Parallel Studio: Vtune Microarchitecture Exploration z metrykami: Front-End Bound, Bad Speculation, Memory Bound, Core Bound, Retiring oraz CPU sampling intervals równy 1 ms.

2. Warianty kodu.

- Wariant sekwencyjny tradycyjny.

```
int* traditional_sequential(int min = MIN, int max = MAX) {
  int* numbers = (int*)malloc((max-min+1)*sizeof(int));

  for (int i = 0; i <= max-min; i++)
    numbers[i] = min + i;

  for (int i = min; i <= max; i++) {
    for (int j = 2; j <= ceil(sqrt(i)); j++) {
      if (i % j == 0) {
        numbers[i - min] = 0;
        break;
      }
    }
  }

  if (min < 3) {
    numbers[0] = 2;
  }

  return numbers;
}
```

1. Listing kodu w wariancie sekwencyjnym – podejście tradycyjne – dzielenie przez liczby z przedziału <2;sqrt(sprawdzana_liczba)>

Jest to implementacja opisanej przez Prowadzącego pierwszej wersji podejścia koncepcyjnego – każda liczba z podanego na wejściu przedziału jest testowana pod względem podzielności przez liczby mniejsze. Zgodnie z sugestią ograniczamy od góry liczbę sprawdzanych dzielników przez pierwiastek kwadratowy z aktualnie badanej liczby. W przypadku, kiedy podane na wejście minimum jest równe 2 – liczba 2 może być uznana za liczbę złożoną – sprawdzamy ten warunek w ostatnim if'ie i w razie potrzeby ustawiamy jej wartość w tablicy numbers.

- Wariant sekwencyjny sita Erastotenesa.

```
int* sieve_sequential(int min = MIN, int max=MAX) {

  int* numbers = (int*)malloc((max-min + 1) * sizeof(int));

  for (int i = 0; i <= max-min; i++)
    numbers[i] = min + i;

  int* primaries = traditional_sequential(2, ceil(sqrt(max)));

  for (int i = 2; i <= ceil(sqrt(max)); i++) {
```

```
    if (primaries[i - 2]) {
      for (int j = primaries[i - 2] * primaries[i - 2];
           j <= max;
           j += (i == 2 ? i : 2 * i))
        {
        if (j >= min) {
          numbers[j - min] = 0;
        }
      }
    }
  }
  return numbers;
}
```

2. Listing kodu w wariancie sekwencyjnym – metoda sita Erastotenesa.

W tym przypadku mamy do czynienia z nieco zmodyfikowaną wersją sita Erastotenesa – zamiast zaczynać od podwojonej wielokrotności liczby pierwszej – algorytm zaczyna od jej kwadratu, a zamiast wykreślać wszystkie wielokrotności liczb pierwszych, zauważamy, że wystarczy tym sposobem pozbyć się jedynie liczb parzystych. Kiedy iterujemy po wielokrotnościach liczby pierwszej różnej od dwójki – co każde wykonanie pętli zwiększamy zmienną o dwukrotność liczby pierwszej.

- Wariant zrównoleglony podejścia tradycyjnego.

```
int* traditional_parallel() {
  int max = MAX, min = MIN;
  int i, j;
  int max_sqrt = ceil(sqrt(max));

  int* numbers = (int*)malloc((MAX - MIN + 1) * sizeof(int));

  #pragma omp parallel for private(i)
  for (i = 0; i <= MAX - MIN; i++)
    numbers[i] = MIN + i;

  #pragma omp parallel for collapse(2) shared(numbers) private(i,j)
  for (i = min; i <= max; i++)
  {
    for (j = 2; j <= max_sqrt; j++)
    {
      if (i % j == 0) {
        numbers[i - min] = 0;
      }
    }
  }

  if (min < 3) {
```

```
    numbers[0] = 2;
  }

  return numbers;
}
```

3. Listing kodu zrównoleglonego w podejściu koncepcyjnym tradycyjnym.

W powyższym kodzie widoczne są trzy modyfikacje względem wersji przedstawionej wyżej w sprawozdaniu. Pierwszą z nich jest zrównoleglenie wypełniania tablicy numbers poprzez dyrektywę `#pragma omp parallel for private(i)` – spowoduje to rozdzielenie pracy przez wątki w taki sposób, że liczba iteracji wykonana przez każdy z nich będzie równa w przybliżeniu 1/n wszystkich iteracji. Drugą zmianą jest pozbycie się instrukcji `break`; z zagnieżdżonej pętli (niemożliwe jest poprawne użycie jej w przypadku rozdzielania pracy wielowątkowo). Trzecią zmianą jest dodanie `#pragma omp parallel for collapse(2) shared(numbers) private(i,j)` (collapse(2) w połączeniu z zadeklarowanymi jako prywatne iteratorami obu pętli, spowodują, że wykonywanie poszczególnych iteracji będzie rozdzielone na wiele wątków nie tylko w ramach wiodącej pętli, ale również tej niższej, tablicę liczb zadeklarowano jako współdzieloną, aby każdy wątek mógł zmodyfikować dowolną komórkę – w rzeczywistości każdy z nich zmodyfikuje tylko część, ale z góry nie wiemy dokładnie do którego fragmentu tablicy będzie potrzebował dostępu).

- [1] Sito Erastotenesa.

```
    int* sieve_parallel(int min = MIN, int max = MAX) {

    int i, j;

    int max_sqrt = ceil(sqrt(max));

    int* numbers = (int*)malloc((max - min + 1) * sizeof(int));

#pragma omp parallel for private(i)
    for (i = 0; i <= max - min; i++)
      numbers[i] = min + i;

    int* primaries = traditional_sequential(2, ceil(sqrt(max)));

    for (i = 2; i <= max_sqrt; i++) {
      if (primaries[i - 2]) {
#pragma omp parallel for schedule(static) private(j) shared(numbers)
        for (j = i * i; j <= max; j += (i == 2 ? i : 2 * i)) {
          if (j >= min) {
            numbers[j - min] = 0;
          }
        }
      }
    }
    free(primaries);
    return numbers;
```

```
}
```

4. Listing kodu – zrównoleglone sito Erastotenesa – wersja A.

Główną zmianą, którą można zauważyć jest zrównoleglenie jedynie drugiej pętli (zagnieżdżonej). Wątki dzielą się między sobą pracą w ramach każdej liczby pierwszej, a liczba iteracji, które wykona każdy wątek jest w przybliżeniu równa liczbie wszystkich iteracji podzielonych przez liczbę wątków (to za sprawą schedule(static)). Podejście takie jest według nas ciekawe z tego względu, że przejście przez niektóre iteracje (np. dla dwójki) głównej pętli wymaga dużo więcej operacji, niż przez inne (dla każdej liczby innej niż 2 liczba iteracji gwałtownie spada).

Nazwa	NUM_THREADS	Przedział	Czas przetwarzania
sieve_parallel_A	4	2...MAX	0.601 s
sieve_parallel_A	4	MAX/2...MAX	0.357 s
sieve_parallel_A	2	2…MAX	0.625 s

Przyspieszenie przetwarzania	Prędkość przetwarzania	Efektywność przetwarzania
82	83 000 000 [liczb/s]	41
84	70 000 000 [liczb/s]	42
78	80 000 000 [liczb/s]	39

Wyniki zbiorcze dla niektórych parametrów wersji A sita Erastotenesa.

- [2] Sito Erastotenesa podejście równoległe .

```
  int* sieve_parallel(int min = MIN, int max = MAX) {

    int* numbers = (int*)malloc((max - min + 1) * sizeof(int));

#pragma omp parallel for private(i)
    for (int i = 0; i <= max - min; i++)
      numbers[i] = min + i;

    int* primaries = traditional_sequential(2, ceil(sqrt(max)));

    int i, j;
    int max_sqrt = ceil(sqrt(max));

#pragma omp parallel for schedule(static) default(none) private(i) shared(j,max,
primaries,numbers,max_sqrt,min)
        for (i = 2; i <= max_sqrt; i++) {
          if (primaries[i - 2]) {
            for (j = primaries[i - 2] * primaries[i - 2]; j <= max;
                 j += (primaries[i-2]==2 ? 2 : 2 * primaries[i - 2])) {
              if (j >= min) {
                numbers[j - min] = 0;
```

```
            }
          }
        }
      }
    free(primaries);
    return numbers;
  }
```

5. Listing kodu – zrównoleglenie – sito Erastotenesa – wersja B.

W tym przypadku procesy będą dzieliły się pracą w ramach pierwszej pętli – każdy z nich wykona całość pracy dla konkretnej liczby pierwszej. Możliwe jest tutaj zaobserwowanie zróżnicowanie między ilością pracy, które muszą wykonać poszczególne wątki.

Nazwa	NUM_THREADS	Przedział	Czas przetwarzania
sieve_parallel_B	4	2...MAX	0.620 s
sieve_parallel_B	4	MAX/2...MAX	0.377 s
sieve_parallel_B	2	2…MAX	0.625 s

Przyspieszenie przetwarzania	Prędkość przetwarzania	Efektywność przetwarzania
79	81 000 000 [liczb/s]	39,5
80	66 000 000 [liczb/s]	40
78	80 000 000 [liczb/s]	39

Wyniki zbiorcze dla niektórych parametrów wersji B sita Erastotenesa.

- Sito Erastotenesa podejście domenowe.

```
    int** sieve_init(int min = MIN, int max = MAX) {

    omp_set_num_threads(NUM_THREADS);
    int in_one_row = ceil((double)(max - min + 1) / (double)NUM_THREADS);

    int i,j;

    int max_sqrt = ceil(sqrt(max));

    int** numbers = (int**)malloc(NUM_THREADS * sizeof(int*));
    int* primaries = traditional_sequential(2, ceil(sqrt(max)));

#pragma omp parallel for
    for (int i = 0; i < NUM_THREADS; i++) {
      numbers[i] = (int*)malloc(in_one_row * sizeof(int));
    }

#pragma omp parallel for
    for (int i = 0; i <= max - min; i++) {
```

```
      int row = floor(i / in_one_row);
      int j = i % in_one_row;
      numbers[row][j] = min + i;
      if (i == max - min && j < in_one_row-1) {
        for (j; j < in_one_row; j++) {
          numbers[row][j] = 0;
        }
      }
    }

#pragma omp parallel sections private(i,j)
    {

#pragma omp section
      {
        int th_nb = omp_get_thread_num();
        int maks = numbers[th_nb][in_one_row - 1] == 0 ? max : numbers[th_nb]
[in_one_row - 1];
        int minm = numbers[th_nb][0];

        for (i = 2; i <= max_sqrt; i++) {
          if (primaries[i - 2]) {
            for (j = primaries[i - 2] * primaries[i - 2]; j <= maks; j += (primar
ies[i - 2] == 2 ? 2 : 2 * primaries[i - 2])) {
              if (j >= minm) {
                numbers[th_nb][j - minm] = 0;
              }
            }
          }
        }
      }
#pragma omp section
      {
        int th_nb = omp_get_thread_num();
        int maks = numbers[th_nb][in_one_row - 1] == 0 ? max : numbers[th_nb]
[in_one_row - 1];
        int minm = numbers[th_nb][0];

        for (i = 2; i <= max_sqrt; i++) {
          if (primaries[i - 2]) {
            for (j = primaries[i - 2] * primaries[i - 2]; j <= maks; j += (primar
ies[i - 2] == 2 ? 2 : 2 * primaries[i - 2])) {
              if (j >= minm) {
                numbers[th_nb][j - minm] = 0;
              }
            }
          }
```

```
        }
      }
#pragma omp section
      {
        int th_nb = omp_get_thread_num();
        int maks = numbers[th_nb][in_one_row - 1] == 0 ? max : numbers[th_nb]
[in_one_row - 1];
        int minm = numbers[th_nb][0];

        for (i = 2; i <= max_sqrt; i++) {
          if (primaries[i - 2]) {
            for (j = primaries[i - 2] * primaries[i - 2]; j <= maks; j += (primar
ies[i - 2] == 2 ? 2 : 2 * primaries[i - 2])) {
              if (j >= minm) {
                numbers[th_nb][j - minm] = 0;
              }
            }
          }
        }
      }
#pragma omp section
      {
        int th_nb = omp_get_thread_num();
        int maks = numbers[th_nb][in_one_row - 1] == 0 ? max : numbers[th_nb]
[in_one_row - 1];
        int minm = numbers[th_nb][0];

        for (i = 2; i <= max_sqrt; i++) {
          if (primaries[i - 2]) {
            for (j = primaries[i - 2] * primaries[i - 2]; j <= maks; j += (primar
ies[i - 2] == 2 ? 2 : 2 * primaries[i - 2])) {
              if (j >= minm) {
                numbers[th_nb][j - minm] = 0;
              }
            }
          }
        }

      }
    }

    return numbers;
  }

        int th_nb = omp_get_thread_num();
        int maks = numbers[th_nb][in_one_row - 1] == 0 ? max : numbers[th_nb]
[in_one_row - 1];
```

```
        int minm = numbers[th_nb][0];

        for (i = 2; i <= max_sqrt; i++) {
          if (primaries[i - 2]) {
            for (j = primaries[i - 2] * primaries[i - 2]; j <= maks; j += (primar
ies[i - 2] == 2 ? 2 : 2 * primaries[i - 2])) {
              if (j >= minm) {
                numbers[th_nb][j - minm] = 0;
              }
            }
          }
        }
      }
 ......
```

6. Listing kodu – sito Erastotenesa – podejście domenowe – wersja do badania czterowątkowo.

Podejście domenowe zdecydowaliśmy się zrealizować za pomocą sekcji OpenMP. Tablica numbers została zainicjowana jako tablica dwuwymiarowa, a wierszy (i sekcji section) zostało utworzonych tyle, ile jest obecnie badanych wątków. Dzięki temu każdy proces dostaje całą tablicę liczb pierwszych i wykonuje dla wszystkich liczb pierwszych wykreślenia na fragmencie tablicy.

	NUM_THREADS=4	NUM_THREADS=2	NUM_THREADS=1
2...MAX	0.708957 [s]	0.880985 [s]	1.14899 [s]
MAX/2...MAX	0.434958 [s]	0.640611 [s]	0.732272 [s]
2...MAX/2	0.34832 [s]	0.430850 [s]	0.657851 [s]

Wyniki otrzymane za pomocą omp_get_wtime();
**NUM_THREADS=4 – maksymalna liczba dostępnych w systemie procesorów logicznych*
***NUM_THREADS=2 – maksymalna dostępna w systemie liczba procesorów fizycznych*
****NUM_THREADS=1 – liczba procesorów równa ½ liczby procesorów fizycznych*

- Sito Erastotenesa podejście funkcyjne.

```
  int* sieve_init(int min = MIN, int max = MAX) {

    omp_set_num_threads(NUM_THREADS);

    int i, j;

    int max_sqrt = ceil(sqrt(max));

    int* numbers = (int*)malloc((max-min+1) * sizeof(int*));
    int* primaries = traditional_sequential(2, ceil(sqrt(max)));

#pragma omp parallel for
    for (int i = 0; i < max-min+1; i++) {
      numbers[i] = min+i;
    }
```

```
#pragma omp parallel sections private(i,j)
    {

#pragma omp section
      {
        int th_nb = omp_get_thread_num();

        for (i = 2; i <= 2; i++) {
          if (primaries[i - 2]) {
            for (j = primaries[i - 2] * primaries[i - 2]; j <= max; j += (primari
es[i - 2] == 2 ? 2 : 2 * primaries[i - 2])) {
              if (j >= min) {
                numbers[j - min] = 0;
              }
            }
          }
        }
      }
#pragma omp section
      {

        for (i = 3; i <= 5; i++) {
          if (primaries[i - 2]) {
            for (j = primaries[i - 2] * primaries[i - 2]; j <= max; j += (primari
es[i - 2] == 2 ? 2 : 2 * primaries[i - 2])) {
              if (j >= min) {
                numbers[j - min] = 0;
              }
            }
          }
        }
      }
#pragma omp section
      {
        for (i = 7; i <= 13; i++) {
          if (primaries[i - 2]) {
            for (j = primaries[i - 2] * primaries[i - 2]; j <= max; j += (primari
es[i - 2] == 2 ? 2 : 2 * primaries[i - 2])) {
              if (j >= min) {
                numbers[j - min] = 0;
              }
            }
          }
        }
      }
```

```
#pragma omp section
      {
        for (i = 17; i <= max_sqrt; i++) {
          if (primaries[i - 2]) {
            for (j = primaries[i - 2] * primaries[i - 2]; j <= max; j += (primari
es[i - 2] == 2 ? 2 : 2 * primaries[i - 2])) {
              if (j >= min) {
                numbers[j - min] = 0;
              }
            }
          }
        }

      }
    }

    return numbers;
  }
```

6. Listing kodu – wersja funkcyjna sita Erastotenesa – wersja do badań czterowątkowych.

W podejściu funkcyjnym również wykorzystaliśmy pomysł z podziałem na sekcje. Świadomi tego, że skoro jeden wątek ma obsłużyć wszystkie wielokrotności konkretnego fragmentu liczb pierwszych – jednemu wątkowi przydzielamy jedynie dwójkę, jednemu – tylko 3 i 5, kolejnemu 7,11,13, a następny otrzymuje wszystkie pozostałe wielokrotności. W przypadku załączonym powyżej aktywne są 4 wątki z 4 dostępnych. W momencie kiedy uruchamiamy dwa wątki – jeden z nich dostaje tylko dwójkę, drugi – wszystko pozostałe. Tutaj spodziewamy się i tak tego, że praca wątku, który szuka wielokrotności dwójek będzie najdłuższa.

Spośród wszystkich wersji równoległych wyszukiwania liczb pierwszych, najszybszą okazała się być modyfikacja sita Erastotenesa w podejściu funkcyjnym. Jest ona 6 razy szybsza od algorytmu traditional_parallel, 1,27 razy szybsza od podejścia domenowego, 1,08 od wersji B zrównoleglonego sita (która może nieco przypominać podejście funkcyjne – każdy wątek odpowiada za wykreślenie wszystkich wielokrotności liczby/liczb, która przypadła mu w udziale), oraz 1,04 razy szybsza od wersji A, która wykorzystuje zrównoleglenie jedynie drugiej iteracji zagnieżdżonych pętli. Największym atutem okazało się być zastosowanie podziału kodu na sekcje, oraz fakt, że wątki nie powtarzają więcej niż raz obliczeń na liczbach mniejszych od dolnej granicy (w przypadku sieve_functional mamy też najwyższe (jedno 70%, reszta > 92%) wykorzystanie efektywne rdzeni fizycznych procesora oraz najwyższe przyspieszenie). Taka sytuacja ma miejsce w wersji domenowej kodu, której główna myśl mogłaby być porównywany do wersji A – przez to, że wiele wątków wykreśla różne liczby pierwsze z tablicy wyjściowej. Wersja A okazała się mieć jednak przewagę (około 1,22 razy szybsza) właśnie przez to, że udało uniknąć się powtórzeń tych samych operacji (mimo, że mamy do czynienia z false sharingiem wyjściowej tablicy liczb, a podejście domenowe jest jedynym, gdzie tego zjawiska nie doświadczymy).

Zauważyć można również, że najlepsze osiągi mają algorytmy wykorzystujące przewidziany wcześniej rozmiar zadań dla każdego wątku – jak wspomniane zostało wyżej – taka

sytuacja ma miejsce w wersji funkcyjnej (dzięki zastosowaniu sekcji) i w wersji A kodu (dzięki zastosowaniu schedule(static)).

Przyglądając się wynikom zauważamy, że znaczącym obciążeniem dla systemu jest utrzymanie wielu wątków. Przy takich samych wielkościach instancji mamy czasami do czynienia nawet ze wzrostem (!) czasu przetwarzania wraz ze zwiększoną liczbą wątków (widzimy to m.in. na przykładzie sieve_domain, gdzie dla instancji 2...MAX/2 dla liczby wątków równych kolejno 1,2,4 osiągamy odpowiednio czasy: 0.491s, 0.403s, **0.441s**). Warto o tym pamiętać – zwłaszcza, kiedy chcemy używać wielu wątków, że nie zawsze więcej, znaczy lepiej.

Nieoczywistym faktem, który można zauważyć podczas analizy wyników jest również to, że sekwencyjny odpowiednik podejścia tradycyjnego dla instancji o rozmiarze MAX=500 000, sprawdził się lepiej niż którykolwiek (mam na myśli 1,2,4-wątkowy) odpowiednik wersji zrównoleglonej (mamy tutaj wzrost czasu od 20-krotnych do nawet 60-krotnych zależnie od wielkości i NUM_THREADS w wersji równoległej). Głównymi winowajcami są tutaj: brak instrukcji break, która nie została użyta w przetwarzaniu wielowątkowym oraz ograniczenie odgórne pętli – ustalone jako stała równa ceil(sqrt(max)), a nie – jak w wydajniejszym odpowiedniku ceil(sqrt(i)), gdzie „i” to liczba, której sprawdzamy dzielniki. Użycie OpenMP wymaga niekiedy perfekcyjnie zagnieżdżonych pętli, czy prostych warunków postępu – i nie zawsze dany pomysł uda się zrealizować dokładnie odwzorowując podejście sekwencyjne.

Nazwa	Przedział	NUM_THREADS	Elapsed time	Instructions retired	Clockticks	Retiring
sieve_domain	2….MAX	1	0.941 s	4,039,392,048	3,894,353,664	38.3%
	MAX/2….MAX	1	0.617 s	2,821,723,536	4,811,639,712	35.3%
	2...MAX/2	1	0.491 s	3,637,794,408	1,516,837,824	40.5%
	2….MAX	2	0.737 s	2,364,085,248	4,841,730,216	33.3%
	MAX/2….MAX	2	0.470 s	2,099,866,800	2,262,196,920	60.0%
	2….MAX/2	2	0.403 s	701,061,216	3,050,144,928	47.1%
	2...MAX	4	0.729 s	3,298,245,288	6,783,184,056	37.7%
	MAX/2...MAX	4	0.465 s	6,068,620,632	3,563,102,328	43.7%
	2...MAX/2	4	0.441 s	1,211,413,632	4,309,957,296	40.2%

Front-end bound	Back-end bound	Memory bound	Core bound	Effective physical core utilization	Przyspieszenie przetwarzania	Prędkość przetwarzania [liczb/s]	Efektywność przetwarzania
16.9%	40.9%	26.0%	14.9%	60.0%	52	1/0,0000000188	52
39.3%	16.3%	12.4%	3.9%	108.8%	49	1/0,0000000247	49
45.6%	8.0%	3.2%	4.7%	89.8%	37	1/0,0000000196	37
14.7%	49.6%	35.9%	13.7%	77.7%	65	1/0,0000000147	32,5
31.9%	4.0%	2.2%	1.8%	78.7%	64	1/0,0000000295	32
22.9%	24.7%	13.8%	10.8%	86.8%	44	1/0,0000000403	22
19.6%	38.7%	27.3%	11.4%	78.6%	67	1/0,0000000146	33,5
29.8%	22.9%	11.9%	11.0%	81.9%	65	1/0,0000000186	32,5
26.7%	29.9%	18.1%	11.7%	94.8%	49	1/0,0000000176	24,5

MAX=50 000 000

Nazwa	Przedział	NUM_THREADS	Elapsed time	Instructions retired	Clockticks	Retiring
traditional_parallel	2...MAX/100	1	8.111 s	40,984,582,704	42,018,493,944	48.8%
	MAX/200...MAX/100	1	3.510 s	17,765,247,288	18,419,988,336	52.2%
	2...MAX/200	1	2.391 s	11,949,251,712	10,207,501,704	54.3%
	2...MAX/100	2	4.204 s	30,969,069,840	32,941,702,176	56.5%
	MAX/200...MAX/100	2	2.097 s	16,044,407,376	15,045,246,360	57.0%
	2...MAX/200	2	1.499 s	10,422,112,824	10,333,813,464	67.3%
	2...MAX/100	4	3.313 s	29,579,372,928	39,164,711,208	65.1%
	MAX/200…MAX/100	4	1.612 s	15,973,442,136	20,137,410,720	57.7%
	2...MAX/200	4	1.164 s	10,690,153,920	13,688,974,656	68.6%

Front-end bound	Back-end bound	Memory bound	Core bound	Effective physical core utilization	Przyspieszenie przetwarzania	Prędkość przetwarzania [liczb/s]	Efektywność przetwarzania
29.7%	17.2%	8.5%	8.7%	67.6%	0,017	61 644	0,017
25.9%	17.9%	9.7%	8.2%	70.0%	0,022	71 225	0,022
22.5%	20.2%	9.0%	11.3%	62.0%	0,024	105 000	0,024
24.4%	16.5%	6.4%	10.1%	95.0%	0,031	119 000	0,0155
24.1%	16.7%	5.6%	11.1%	86.0%	0,037	120 000	0,0185
27.4%	2.4%	0.9%	1.5%	86.1%	0,038	167 000	0,019
27.8%	5.2%	1.7%	3.5%	97.5%	0,039	151 000	0,0195
30.5%	9.2%	3.1%	6.1%	109.3%	0,049	155 000	0,0245
27.1%	2.2%	0.8%	1.4%	94.5%	0,049	215 000	0,0245

MAX/100 = 500 000

Nazwa	Przedział	NUM_THREADS	Elapsed time	Instructions retired	Clockticks	Retiring
sieve_functional	2...MAX	1	0.602 s	2,313,528,984	2,347,216,176	34.7%
	MAX/2….MAX	1	0.398 s	2,157,247,704	4,544,671,080	27.0%
	2….MAX/2	1	0.339 s	567,506,016	2,244,526,848	42.7%
	2….MAX	2	0.582 s	936,144,000	6,225,051,264	25.3%
	MAX/2….MAX	2	0.336 s	1,572,700,368	2,966,497,776	35.2%
	2...MAX/2	2	0.308 s	551,408,880	3,456,863,568	26.5%
	2...MAX	4	0.574 s	900,913,584	6,809,609,856	16.7%
	MAX/2...MAX	4	0.374 s	1,101,239,424	3,945,766,320	31.8%
	2….MAX/2	4	0.309 s	1,462,105,992	3,401,115,552	25.5%

Front-end bound	Back-end bound	Memory bound	Core bound	Effective physical core utilization	Przyspieszenie przetwarzania	Prędkość przetwarzania [liczb/s]	Efektywność przetwarzania
51.3%	3.7%	3.0%	0.7%	70.6%	82	83 000 000	82
24.7%	43.4%	35.3%	8.1%	110.1%	75	63 000 000	75
43.9%	3.5%	2.4%	1.1%	109.2%	53	74 000 000	53
11.6%	58.8%	49.6%	9.2%	97.7%	52	86 000 000	26
46.0%	10.3%	8.6%	1.7%	92.1%	89	75 000 000	44,5
21.5%	45.6%	36.2%	9.4%	102.4%	58	81 000 000	29
16.1%	64.4%	56.3%	8.1%	104.9%	32	87 000 000	16
33.8%	25.1%	17.9%	7.2%	99.5%	80	67 000 000	40
34.9%	33.6%	27.5%	6.2%	99.9%	58	81 000 000	29

MAX=50 000 000

Ze starego sprawozdania zostawiliśmy tylko tabelkę, żeby niepotrzebnie nie zaśmiecać.

Wariant	**Czas dla liczby wątków = *1***	**Czas dla liczby wątków = *3***	**Czas dla liczby wątków = *6***	**Czas dla liczby wątków = *12***
sequential	522.890290	---	---	---
parallel_1	45.245497	59.373896	64.112328	59.715072
parallel_2	64.308417	59.520607	64.881943	60.102123
parallel_3	64.400735	60.419365	64.950640	60.309975
parallel_4	64.719053	60.326552	64.666392	60.497431
parallel_5	64.960069	60.449499	64.785784	60.148380
parallel_6	64.116249	60.063787	64.331333	60.212495
parallel_7	64.360763	60.276481	63.696052	59.909399
parallel_8	64.558589	59.738300	64.438075	59.946487
parallel_9	64.123436	59.987361	64.521003	59.747855
parallel_10	63.489519	59.936017	64.423157	59.929436
parallel_11	64.457955	60.404314	64.479184	60.402309
parallel_12	64.487153	59.936047	64.895313	60.126156
parallel_13	64.345228	60.413464	64.542227	60.279202
parallel_14	64.161730	60.077114	64.453470	60.391956
parallel_15	63.936027	60.099122	63.898915	59.917218
parallel_16	64.305679	60.081819	64.915416	60.410302
parallel_17	64.711414	60.053957	65.039139	60.130069