Programowanie Równoległe - Liczby Pierwsze

Wersja pierwsza

Autorzy

Grupa dziekańska: 4
Grupa labolatoryjna: 7
Termin zajęć: czwartek, 16:50

Tymoteusz Jagła 151811 - tymoteusz.jagla@student.put.poznan.pl
Kaper Magnuszewski 151746 - kacper.magnuszewski@student.put.poznan.pl

Sprawozdanie

Wymagany termin oddania sprawozdania - 10.05.2024
Rzeczywisty termin oddania sprawozdania - 10.05.2024

Opis zadania

Projekt polegał na zbadaniu efektywności przetwarzania równoległego w komputerze z procesorem wielordzeniowym. Badanym zadaniem było znajdowanie liczb pierwszych w określonym zakresie. Podano różne warianty algorytmów - wyznaczanie liczb pierwszych metodą dzielenia oraz przy użyciu sita Erastothenes'a, w tym podejście sekwencyjne oraz równoległe (domenowe i funkcyjne).

Wykorzystany system obliczeniowy

Procesor

- Model: 13th Gen Intel® Core(TM) i5-13600KF
- Liczba procesorów fizycznych: 14
  - 6 Performance-cores
  - 8 Efficient-cores
- Liczba procesorów logicznych: 20
  - 2 wątki na pojedyńczy Performance-core
  - 1 wątek na pojedyńczy Effitient-core
- Oznaczenie typu procesora: KF
- Taktowanie procesora:
  - Minimalne: 800MHz
  - Maksymalne: 51000MHz
- Wielkości pamięci podręcznej procesora:
  - L1d cache: 544 KiB (14 instancji)
  - L1i cache: 704 KiB (14 instancji)
  - L2 cache: 20 MiB (8 instancji)

    - L3 cache: 24 MiB (1 instancja)
- Organizacja pamięci podręcznej: Intel® Smart Cache

System Operacyjny

- Nazwa systemu operacyjnego: Linux Pop!-OS 6.8.0
- Oprogramowanie wykorzystane do przygotowania kodu wynikowego: Visual Studio Code
- Oprogramowanie wykorzystane do przeprowadzenia testów: Intel VTune Profiler

Wersje programów

Zdefiniowane stałych używanych w kodzie

W osobnym pliku nagłówkowym zostały zdefiniowane stałe takie jak:

- Dolna granica poszukiwania liczb pierwszych `M_VAL`
- Górna granica poszukiwania liczb pierwszych `N_VAL`
- Liczba procesorów użyta do wykonania zadania równolegle

```
#define M_VAL 2
#define N_VAL 1000000
#define THREADS_COUNT 8
#define BLOCKSIZE 8
```

Liczby pierwsze wyznaczane sekwencyjnie przez dzielenie w zakresie $< m, n >$ (k1)

Poniższy kod to podejście sekwencyjne. Mierzony jest czas pracy procesora za pomocą zmiennych `spstart` i `spstop` oraz rzeczywisty czas pracy programu za pomocą `sswtime` i `sewtime`. Tablica `primeArray` przechowuje zmienne typu `bool` - pierwiastki liczby `n`, które są liczbami pierwszymi. Program metodą dzielenia wyznacza tablicę `primeArray`, po czym korzystając z wartości do niej wpisanych sprawdza wszystkie liczby z zakresu podanego w pliku nagłówkowym. Jeżeli dana liczba nie jest

podzielna przez żaden z podzielników `n`, oznacza to, że jest to liczba pierwsza. W takim wypadku jest ona zapisywana do tablicy wynikowej `result`.

```
#include <math.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "consts.h"

int main(int argc, char *argv[]) {
  long int m = M_VAL, n = N_VAL;
  clock_t spstart, spstop;
  double sswtime, sewtime;

  bool *result = (bool *)malloc((n - m + 1) * sizeof(bool));
  memset(result, true, (n - m + 1) * sizeof(bool));

  bool *primeArray = (bool *)malloc((sqrt(n) + 1) * sizeof(bool));
  memset(primeArray, true, (sqrt(n) + 1) * sizeof(bool));

  sswtime = omp_get_wtime();
  spstart = clock();

  for (int i = 2; i * i <= n; i++) {
    for (int j = 2; j * j <= i; j++) {
      if (primeArray[j] == true && i % j == 0) {
        primeArray[i] = false;
        break;
      }
    }
  }

  for (int i = m; i <= n; i++) {
    for (int j = 2; j * j <= i; j++) {
      if (primeArray[j] == true && i % j == 0) {
        result[i - m] = false;
        break;
      }
    }
  }

  spstop = clock();
  sewtime = omp_get_wtime();
  printf("Dzielenie sekwencyjne:\n");
  printf("Czas procesorow przetwarzania sekwencyjnego: %f sekund\n",
         ((double)(spstop - spstart) / CLOCKS_PER_SEC));
  printf("Czas trwania obliczen sekwencyjnych - wallclock: %f sekund\n",
         sewtime - sswtime);
}
```

Liczby pierwsze wyznaczane równolegle przez dzielenie w zakresie $< m, n >$ (k2)

Poniższy blok to równoległa implementacja kodu z poprzedniego zadania. W tym celu użyta została biblioteka `OpenMP`. W określonym obszarze równoległym, każdemu z wątków zostaje przydzielona wartość iteratora pętli `i`, który przyjmuje wartości w przedziale $< m, n >$. W tej wersji programu możemy ustawić różne wartości klauzuli `schedule`. Przy wartości `static` wątkom zostaną przydzielone różne liczby z zakresu $< m, n >$, co oznacza, że niektóre wątki zbadają mniej liczb niż inne, jeżeli badane liczby będą wyjątkowo duże. Ten przydział może okazać się niesprawiedliwy i wpłynąć na czas wykonywania programu.
Jeżeli wątkom przydzielone zostałyby kolejne wartości `i` mógłby również wystąpić false-sharing. Oznacza to, że wątki mogłyby nadpisywać tą samą linię pamięci, co znacząco spowolniłoby program. W tym wypadku false-sharing pojawiłby się poprzez nadpisywanie części tablicy `result` znajdujących się w tych samych liniach pamięci.

```
#include <math.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "consts.h"

int main(int argc, char *argv[]) {
  long int m = M_VAL, n = N_VAL;
  clock_t ppstart, ppstop;
  double pswtime, pewtime;

  bool *result = (bool *)malloc((n - m + 1) * sizeof(bool));
  memset(result, true, (n - m + 1) * sizeof(bool));

  bool *primeArray = (bool *)malloc((sqrt(n) + 1) * sizeof(bool));
  memset(primeArray, true, (sqrt(n) + 1) * sizeof(bool));

  pswtime = omp_get_wtime();
  ppstart = clock();

  for (int i = 2; i * i <= n; i++) {
    for (int j = 2; j * j <= i; j++) {
      if (primeArray[j] == true && i % j == 0) {
        primeArray[i] = false;
        break;
      }
    }
  }
```

```
#pragma omp parallel num_threads(THREADS_COUNT)
  {
#pragma omp for
    for (int i = m; i <= n; i++) {
      for (int j = 2; j * j <= i; j++) {
        if (primeArray[j] == true && i % j == 0) {
          result[i - m] = false;
          break;
        }
      }
    }
  }

  ppstop = clock();
  pewtime = omp_get_wtime();

  printf("Dzielenie rownolegle\n");
  printf("Czas procesorow przetwarzania sekwencyjnego: %f sekund\n",
         ((double)(ppstop - ppstart) / CLOCKS_PER_SEC));
  printf("Czas trwania obliczen sekwencyjnych - wallclock: %f sekund\n",
         pewtime - pswtime);
}
```

Sito sekwencyjne bez lokalności dostępu do danych (k3)

Poniższy kod to sekwencyjna implementacja algorytmu sita Erastothenesa. Pierwsza pętla programu ponownie wyznacza liczby pierwsze w zakresie $< 2, n >$. Druga pętla identyfikuje liczby pierwsze w zakresie $< m, n >$ i oznacza jako liczby złożone ich wielokrotności. W ten sposób w tablicy wynikowej otrzymujemy jedynie liczby pierwsze.

```
#include <math.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "consts.h"

int main(int argc, char *argv[]) {
  long int m = M_VAL, n = N_VAL;
  clock_t ppstart, ppstop;
  double pswtime, pewtime;

  bool *result = (bool *)malloc((n - m + 1) * sizeof(bool));
  memset(result, true, (n - m + 1) * sizeof(bool));
```

```
  bool *primeArray = (bool *)malloc((sqrt(n) + 1) * sizeof(bool));
  memset(primeArray, true, (sqrt(n) + 1) * sizeof(bool));

  pswtime = omp_get_wtime();
  ppstart = clock();

  for (int i = 2; i * i * i * i <= n; i++) {
    if (primeArray[i] == true) {
      for (int j = i * i; j * j <= n; j += i) {
        primeArray[j] = false;
      }
    }
  }

  for (int i = 2; i * i <= n; i++) {
    if (primeArray[i]) {
      int firstMultiple = (m / i);

      if (firstMultiple <= 1) {
        firstMultiple = i + i;
      } else if (m % i) {
        firstMultiple = (firstMultiple * i) + i;
      } else {
        firstMultiple = (firstMultiple * i);
      }

      for (int j = firstMultiple; j <= n; j += i) {
        result[j - m] = false;
      }
    }
  }

  ppstop = clock();
  pewtime = omp_get_wtime();

  printf("Sito sekwencyjne bez lokalności dostępu do danych\n");
  printf("Czas procesorow przetwarzania sekwencyjnego: %f sekund\n",
         ((double)(ppstop - ppstart) / CLOCKS_PER_SEC));
  printf("Czas trwania obliczen sekwencyjnych - wallclock: %f sekund\n",
         pewtime - pswtime);
}
```

Sito sekwencyjne z potencjalną lokalnością dostępu do danych (k3a)

Poniższy kod przedstawia sekwencyjna implementację algorytmu sita Erastothenesa. Program został zmodyfikowany względem poprzednika. Została dodana lokalność dostępu do danych w oparciu o koncepcję lokalnego sita domenowego.

```
#include <math.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "consts.h"

int main(int argc, char *argv[]) {
  long int m = M_VAL, n = N_VAL;
  clock_t ppstart, ppstop;
  double pswtime, pewtime;
  int blockSize = BLOCKSIZE;

  bool *result = (bool *)malloc((n - m + 1) * sizeof(bool));
  memset(result, true, (n - m + 1) * sizeof(bool));

  bool *primeArray = (bool *)malloc((sqrt(n) + 1) * sizeof(bool));
  memset(primeArray, true, (sqrt(n) + 1) * sizeof(bool));

  pswtime = omp_get_wtime();
  ppstart = clock();

  for (int i = 2; i * i * i * i <= n; i++) {
    if (primeArray[i] == true) {
      for (int j = i * i; j * j <= n; j += i) {
        primeArray[j] = false;
      }
    }
  }

  int numberOfBlocks = (n - m) / blockSize;
  if ((n - m) % blockSize != 0) {
    numberOfBlocks++;
  }

  int low = m + blockSize;
  int high = m + blockSize + blockSize;
  if (high > n) {
    high = n;
  }

  for (int i = 0; i < numberOfBlocks; i++) {
    for (int j = 2; j * j <= high; j++) {
      if (primeArray[j]) {
        int firstMultiple = (low / j);
        if (firstMultiple <= 1) {
          firstMultiple = j + j;
        } else if (low % j) {
          firstMultiple = (firstMultiple * j) + j;
        } else {
          firstMultiple = (firstMultiple * j);
        }
        for (int k = firstMultiple; k <= high; k += j) {
          result[k - m] = false;
```

```
        }
      }
    }
  }


  ppstop = clock();
  pewtime = omp_get_wtime();

  printf("Sito sekwencyjne bez lokalności dostępu do danych\n");
  printf("Czas procesorow przetwarzania sekwencyjnego: %f sekund\n",
         ((double)(ppstop - ppstart) / CLOCKS_PER_SEC));
  printf("Czas trwania obliczen sekwencyjnych - wallclock: %f sekund\n",
         pewtime - pswtime);
}
```

Sito równoległe funkcyjne bez lokalności dostępu do danych (k4)

Poniższy kod to równoległa implementacja poprzedniego zadania znajdowania liczb pierwszych przy użyciu algorytmu sita Erastothenes'a. W podanym algorytmie występuje brak lokalności dostępu do danych, co spowodowane jest możliwością odczytywania przez wątki komórek tablicy `result`, które dzielą tą samą linię cache (występuje false-sharing). Działanie programu może zostać przez to znacząco spowolnione.

```
#include <math.h>
#include <omp.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "consts.h"

int main() {
  long int m = M_VAL, n = N_VAL;
  clock_t ppstart, ppstop;
  double pswtime, pewtime;

  bool *result = (bool *)malloc((n - m + 1) * sizeof(bool));
  memset(result, true, (n - m + 1) * sizeof(bool));

  bool *primeArray = (bool *)malloc((sqrt(n) + 1) * sizeof(bool));
  memset(primeArray, true, (sqrt(n) + 1) * sizeof(bool));

  pswtime = omp_get_wtime();
  ppstart = clock();
```

```
  for (int i = 2; i * i * i * i <= n; i++) {
    if (primeArray[i] == true) {
      for (int j = i * i; j * j <= n; j += i) {
        primeArray[j] = false;
      }
    }
  }

  int sqrtn = sqrt(n);

#pragma omp parallel for num_threads(THREADS_COUNT)
  for (int i = 2; i <= sqrtn; i++) {
    if (primeArray[i]) {
      int firstMultiple = (m / i);

      if (firstMultiple <= 1) {
        firstMultiple = i + i;
      } else if (m % i) {
        firstMultiple = (firstMultiple * i) + i;
      } else {
        firstMultiple = (firstMultiple * i);
      }

      for (int j = firstMultiple; j <= n; j += j) {
        result[j - m] = false;
      }
    }
  }

  ppstop = clock();
  pewtime = omp_get_wtime();

  printf("Sito równoległe funkcyjne bez lokalności dostępu do danych\n");
  printf("Czas procesorow przetwarzania sekwencyjnego: %f sekund\n",
         ((double)(ppstop - ppstart) / CLOCKS_PER_SEC));
  printf("Czas trwania obliczen sekwencyjnych - wallclock: %f sekund\n",
         pewtime - pswtime);
}
```

Sito równoległe funkcyjne bez lokalności dostępu do danych (k4a)

Jest to implementacja kodu z zadania poprzedniego, w której dodatkowo sprawdzana jest wartość tablicy `result` w komórce, do której program chciałby wpisać wartość `false`. Oznacza to, że możemy uniknąć wielu nadmiernych unieważnień pamięci oraz ograniczyć ilość prób niepotrzebnych zmian wartości tablicy wynikowej.

```
#include <math.h>
#include <omp.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "consts.h"

int main() {
  long int m = M_VAL, n = N_VAL;
  clock_t ppstart, ppstop;
  double pswtime, pewtime;

  bool *result = (bool *)malloc((n - m + 1) * sizeof(bool));
  memset(result, true, (n - m + 1) * sizeof(bool));

  bool *primeArray = (bool *)malloc((sqrt(n) + 1) * sizeof(bool));
  memset(primeArray, true, (sqrt(n) + 1) * sizeof(bool));

  pswtime = omp_get_wtime();
  ppstart = clock();

  for (int i = 2; i * i * i * i <= n; i++) {
    if (primeArray[i] == true) {
      for (int j = i * i; j * j <= n; j += i) {
        primeArray[j] = false;
      }
    }
  }

  int sqrtn = sqrt(n);

#pragma omp parallel for
  for (int i = 2; i <= sqrtn; i++) {
    if (primeArray[i]) {
      int firstMultiple = (m / i);

      if (firstMultiple <= 1) {
        firstMultiple = i + i;
      } else if (m % i) {
        firstMultiple = (firstMultiple * i) + i;
      } else {
        firstMultiple = (firstMultiple * i);
      }

      for (int j = firstMultiple; j <= n; j += j) {
        if (result[j - m])
          result[j - m] = false;
      }
    }
  }

  ppstop = clock();
  pewtime = omp_get_wtime();
```

```
  printf("Sito równoległe funkcyjne bez lokalności dostępu do danych\n");
  printf("Czas procesorow przetwarzania sekwencyjnego: %f sekund\n",
         ((double)(ppstop - ppstart) / CLOCKS_PER_SEC));
  printf("Czas trwania obliczen sekwencyjnych - wallclock: %f sekund\n",
         pewtime - pswtime);
}
```

Sito równoległe domenowe z potencjalną lokalnością dostępu do danych (k5)

Poniższy kod również przedstawia algorytm sita Erastothenesa, jednak dodatkowo stosuje taktykę podziału obszaru roboczego na bloki. Dzięki takiemu rozwiązaniu możemy uniknąć false sharingu, a co za tym idzie znacznie przyspieszyć wykonywanie się programu. Najlepszy efekt osiągniemy, jeżeli wielkość bloku dostosujemy do długości linii pamięci. Dzięki temu unikniemy niepotrzebnych odczytów w linii pamięci przez różne wątki, a co za tym idzie jej unieważniania.

Kod 5. Sito równoległe domenowe z potencjalną lokalnością dostępu do danych*

```
#include <math.h>
#include <omp.h>
#include <stdio.h>
#include <string.h>
#include <time.h>

#include "consts.h"

int main() {
  long int m = M_VAL, n = N_VAL;
  clock_t ppstart, ppstop;
  double pswtime, pewtime;

  bool *result = (bool *)malloc((n - m + 1) * sizeof(bool));
  memset(result, true, (n - m + 1) * sizeof(bool));
  bool *primeArray = (bool *)malloc((sqrt(n) + 1) * sizeof(bool));
  memset(primeArray, true, (sqrt(n) + 1) * sizeof(bool));

  pswtime = omp_get_wtime();
  ppstart = clock();

  int blockSize = BLOCKSIZE;
  int numberOfBlocks = (n - m) / blockSize;
  if ((n - m) % blockSize != 0) {
    numberOfBlocks++;
  }
#pragma omp parallel for num_threads(THREADS_COUNT)
```

```
    for (int i = 0; i < numberOfBlocks; i++) {
      int low = m + i * blockSize;
      int high = m + i * blockSize + blockSize;
      if (high > n) {
        high = n;
      }
      for (int j = 2; j * j <= high; j++) {
        if (primeArray[j]) {
          int firstMultiple = (low / j);
          if (firstMultiple <= 1) {
            firstMultiple = j + j;
          } else if (low % j) {
            firstMultiple = (firstMultiple * j) + j;
          } else {
            firstMultiple = (firstMultiple * j);
          }
          for (int k = firstMultiple; k <= high; k += j) {
            result[k - m] = false;
          }
        }
      }
    }

    ppstop = clock();
    pewtime = omp_get_wtime();

    printf(
        "Sito równoległe domenowe z potencjalną lokalnością dostępu do
  danych\n");
    printf("Czas procesorow przetwarzania rownoleglego: %f sekund\n",
           ((double)(ppstop - ppstart) / CLOCKS_PER_SEC));
    printf("Czas trwania obliczen rownoleglych - wallclock: %f sekund\n",
           pewtime - pswtime);
  }
```

Wyniki eksperymentu

Aby przeprowadzić eksperyment obliczeniowo-pomiarowy, wykorzystaliśmy funkcję "Microarchitecture Exploration" dostępną w programie Intel VTune. Ta funkcja umożliwia analizę efektywności przetwarzania. W ramach tego trybu zbierane są dane podczas działania procesora za pomocą jednostek monitorujących wydajność.
Programy sekwencyjne zostały uruchomione jednokrotnie, a ich równoległe odpowiedniki w dwóch wariantach. Z racji, że procesor posiada 6 Performance-cores oraz 8 Efficient-cores postanowiliśmy uruchomić program dla ograniczonej części rdzeni typu Performance (Nie jesteśmy jednak pewni jakich rdzeni procesor używa w danym momencie, gdyż nie zostało to opisane), czyli dla 4 procesorów oraz dla połowy maksymalnej liczby procesorów logicznych, która wynosi 10.

Przetwarzane przez nas przedziały, to:

- $< 2, MAX >$, gdzie $MAX = 10^8$
- $< 2, MAX/2 >$, gdzie $MAX = 10^8$
- $< MAX/2, MAX >$, gdzie $MAX = 10^8$

Wykorzystane w tabelach oznaczenia:

- Czas przetwarzania - Elapsed Time w VTune wyrażony w sekundach
- Instructions Retired (IT) - liczba wywoływanych instrukcji kodu asemblera
- ClockTicks (CT) - liczba cykli procesorów w trakcie wykonywania kodu
- Ograniczenie wejścia (F-EB) - udział procentowy w ograniczeniu efektywności przetwarzania części wejściowej procesora
- Ograniczenie wyjścia (B-EB) - udział procentowy w ograniczeniu efektywności przetwarzania części wyjściowej procesora
- Ograniczenie systemu pamięci (MB) - udział procentowy w ograniczeniu efektywności przetwarzania systemu pamięci
- Ograniczenie jednostek wykonawczych (CB) - udział procentowy w ograniczeniu efektywności przetwarzania jednostek wykonawczych procesora
- Efektywne wykorzystanie rdzeni fizycznych procesora - procentowe wykorzystanie dostępnych zasobów w postaci rdzeni procesora

Liczby pierwsze wyznaczane sekwencyjnie poprzez dzielenie oraz metodą sita (k1, k3, k3a)

W poniższej tabeli znajdują się wyniki przetwarzania dla metod sekwencyjnych.

Tabela 1: Tabela wartości parametrów dla algorytmu k1

Nr	MIN	MAX	Kod	Wątki	ET[s]	IR	CT	R[%]	F-EB[%]	B-EB[%]	MB[%]	CB[%]	EPCU[%]
1	2	10^8	k1	1	40.741s	404,678,5	200,761,7	30.6%	48.3%	3.6%	0.2%	3.5%	6.9%
2	(10^8)/2	10^8	k1	1	18.213s	59,333,80	38,434,40	23.5%	43.4%	7.7%	5.0%	2.7%	3.0%
3	2	(10^8)/2	k1	1	16.187s	66,157,10	32,924,50	28.7%	44.4%	5.9%	5.9%	0.0%	2.9%

Tabela 2: Tabela wartości parametrów dla algorytmu k3

Nr	MIN	MAX	Kod	Wątki	ET[s]	IR	CT	R[%]	F-EB[%]	B-EB[%]	MB[%]	CB[%]	EPCU[%]
10	2	10^8	k3	1	1.340s	1,207,700	3,125,000	6.4%	22.5%	78.1%	72.7%	5.3%	3.3%
11	(10^8)/2	10^8	k3	1	0.592s	602,500,0	1,382,400	6.6%	22.5%	77.3%	76.5%	0.8%	3.3%
12	2	(10^8)/2	k3	1	0.618s	599,100,0	1,394,200	6.9%	25.0%	73.7%	69.6%	4.2%	3.0%

Tabela 3: Tabela wartości parametrów dla algorytmu k3a

Nr	MIN	MAX	Kod	Wątki	ET[s]	IR	CT	R[%]	F-EB[%]	B-EB[%]	MB[%]	CB[%]	EPCU[%]
13	2	10^8	k3a	1	0.249s	862,100,0	486,000,0	22.7%	37.5%	27.3%	20.0%	7.3%	2.6%
14	(10^8)/2	10^8	k3a	1	0.605s	606,900,0	1,419,400	6.6%	21.0%	70.8%	69.0%	1.9%	3.3%
15	2	(10^8)/2	k3a	1	0.166s	442,200,0	254,800,0	33.8%	52.8%	25.4%	22.1%	3.3%	2.1%

Z powyższych tabeli wynika, że algorytm wyznaczający liczby pierwsze metodą sekwencyjną sposobem dzielenia jest 30 razy wolniejszy od algorytmu sita Erastothenesa. Jesteśmy w stanie uzyskać jeszcze lepsza wydajność stosując lokalność dostępu do danych, co widzimy w tabeli 3. Algorytm ten jest około 164 razy szybszy od sposobu dzielenia oraz około 5 razy szybszy od zwykłego sita. Wyniki te zostały uzyskane dla obliczeń w zakresie $< 2, MAX >$. Dodatkowo możemy zauważyć, że ograniczenie systemu pamięci znacząco się zmniejsza w przypadku wersji kodu k3a.

Liczby pierwsze wyznaczane równolegle poprzez dzielenie oraz metodą sita (k2, k4, k4a, k5)

W poniższej tabeli znajdują się wyniki przetwarzania dla metod równoległych podejściem funkcyjnym (k2, k4, k4a) oraz domenowym (k5).

Tabela 4: Tabela wartości parametrów dla algorytmu k2

Nr	MIN	MAX	Kod	Wątki	ET[s]	IR	CT	R[%]	F-EB[%]	B-EB[%]	MB[%]	CB[%]	EPCU[%]
4	2	10^8	k2	10	7.189s	404,811,3	238,965,1	31.1%	33.6%	5.3%	0.2%	5.1%	36.1%
5	(10^8)/2	10^8	k2	10	4.075s	86,346,80	73,911,10	26.8%	39.2%	7.6%	3.7%	3.9%	16.6%
6	2	(10^8)/2	k2	10	8.314s	50,536,60	36,520,20	30.8%	36.0%	6.5%	3.8%	2.7%	3.7%
7	2	10^8	k2	4	7.080s	88,687,80	59,532,50	26.0%	41.9%	9.0%	5.6%	3.4%	10.3%
8	(10^8)/2	10^8	k2	4	7.081s	80,952,20	60,514,10	23.1%	41.4%	9.8%	5.2%	4.6%	10.8%
9	2	(10^8)/2	k2	4	10.106s	118,530,0	75,203,50	26.1%	42.4%	9.2%	5.7%	3.5%	9.6%

Tabela 5: Tabela wartości parametrów dla algorytmu k4

Nr	MIN	MAX	Kod	Wątki	ET[s]	IR	CT	R[%]	F-EB[%]	B-EB[%]	MB[%]	CB[%]	EPCU[%]
16	2	10^8	k4	10	0.092s	87,900,00	174,800,0	10.5%	32.6%	47.7%	38.9%	8.9%	2.5%
17	(10^8)/2	10^8	k4	10	0.062s	56,900,00	109,600,0	13.1%	65.2%	71.0%	46.6%	24.5%	2.9%
18	2	(10^8)/2	k4	10	0.075s	56,000,00	117,600,0	9.1%	44.5%	34.9%	18.4%	16.5%	1.9%
19	2	10^8	k4	4	0.098s	82,500,00	182,600,0	17.3%	54.5%	68.4%	39.1%	29.3%	1.6%
20	(10^8)/2	10^8	k4	4	0.094s	56,900,00	129,200,0	6.2%	25.0%	30.4%	16.8%	13.7%	1.5%
21	2	(10^8)/2	k4	4	0.053s	56,800,00	108,400,0	17.0%	43.9%	58.4%	35.3%	23.0%	2.0%

Tabela 6: Tabela wartości parametrów dla algorytmu k4a

Nr	MIN	MAX	Kod	Wątki	ET[s]	IR	CT	R[%]	F-EB[%]	B-EB[%]	MB[%]	CB[%]	EPCU[%]
22	2	10^8	k4a	10	0.037s	77,566,80	137,314,2	17.1%	31.0%	100.0%	81.0%	38.1%	4.6%
23	(10^8)/2	10^8	k4a	10	0.057s	56,400,00	109,300,0	14.5%	50.6%	57.7%	42.3%	15.4%	2.2%
24	2	(10^8)/2	k4a	10	0.059s	57,400,00	117,100,0	11.4%	42.4%	47.9%	40.9%	7.0%	2.6%
25	2	10^8	k4a	4	0.083s	80,000,00	155,700,0	10.5%	41.5%	57.3%	38.6%	18.8%	2.3%
26	(10^8)/2	10^8	k4a	4	0.059s	57,100,00	112,300,0	13.7%	49.7%	58.3%	23.2%	35.1%	2.3%
27	2	(10^8)/2	k4a	4	0.065s	56,500,00	105,000,0	11.5%	53.7%	42.3%	25.1%	17.2%	2.4%

Tabela 7: Tabela wartości parametrów dla algorytmu k5

MAX	Kod	Wątki	ET[s]	IR	CT	R[%]	F-EB[%]	B-EB[%]	MB[%]	CB[%]	EPCU[%]
10^8	k5	10	0.411s	3,687,400	3,794,800	25.1%	38.3%	42.6%	29.4%	13.2%	8.4%
10^8	k5	10	0.180s	1,951,200	1,991,000	23.0%	37.7%	42.5%	30.9%	11.6%	10.4%
(10^8)/2	k5	10	0.173s	1,764,100	1,763,100	24.7%	40.7%	40.7%	31.3%	9.3%	9.4%
10^8	k5	4	0.615s	3,712,600	3,543,500	18.4%	38.8%	58.1%	36.0%	22.1%	6.7%
10^8	k5	4	0.430s	1,965,900	2,067,900	17.6%	37.9%	58.4%	32.3%	26.1%	5.5%
(10^8)/2	k5	4	0.233s	1,757,400	1,482,200	16.9%	38.1%	55.2%	41.7%	13.5%	7.9%

W tabelach 4, 5 oraz 6 zostało wykorzystane podejście funkcyjne. Oznacza to, że procesy otrzymują całą tablicę wykreśleń i fragment zbioru liczb pierwszych, których wielokrotności są usuwane. W tym podejściu najlepszy czas uzyskaliśmy przy użyciu 10 wątków dla instancji $< 2, 10^8 >$ algorytmu sita równoległego k4a. Czas ten jest około 273 razy mniejszy od najdłuższego czasu wykonywania, należącego do metody dzielenia równoległego (k2) oraz około 2 razy mniejszy od czasu wykonywania algorytmu k4. Najszybszy z algorytmów wykazuje się jednak znaczącym zwiększeniem ograniczenia systemu pamięci.

Porównując powyżej opisane wyniki do podejścia domenowego, gdzie procesy otrzymują fragment tablicy wykreśleń i całą tablicę liczb pierwszych możemy zauważyć znacznie lepsze wykorzystanie rdzeni procesora. Jest to przedstawione w kolumnie EPCU. Dodatkowo zaobserwowany został zmniejszony narzut pamięci w stosunku do podejścia funkcyjnego. Nie zostały jednak zauważone zmniejszone czasy wykonywania algorytmów.

Zbadaliśmy różne warianty szeregowania pętli for i mimo lepszego wykorzystania wątków w szeregowaniu dynamic, lepsze czasy osiągnęliśmy stosując szeregowanie guided.

Podsumowanie wyników

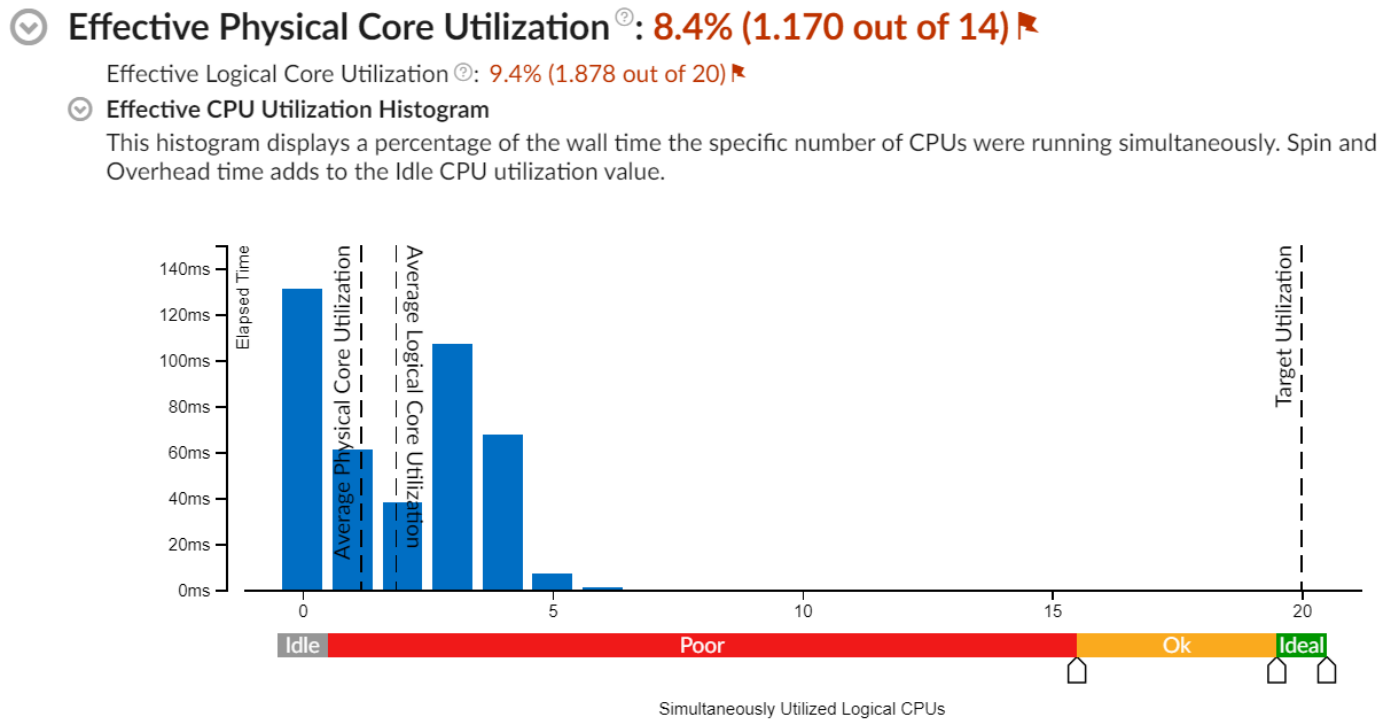
Podsumowując możemy zauważyć około 6-krotne przyspieszenie działania dla algorytmu równoległego względem sekwencyjnego wykorzystującego dzielenie do obliczenia liczb pierwszych z zakresu $< 2, 10^8 >$.

Równoległy algorytm wykorzystujący sito Erastothenes'a osiągnął czas około 36 razy lepszy od wariantu sekwencyjnego dla instancji problemu szukającego liczb pierwszych w zakresie $< 2, 10^8 >$.

Wnioski

Zauważyliśmy, że we wszystkich obliczeniach występuje niska efektywność zużycia wątków procesora. We wszystkich przypadkach ilość wykorzystanych wątków była mniejsza lub równa ilości rdzeni typu Performance procesora. Podejrzewamy, że jest to spowodowane nietypową architekturą nowych procesorów firmy Intel (Rysunek 1).

Rysunek 1: Efektywność zużycia wątków procesora



Najszybciej efektywnym podejściem równoległym okazało się w naszym eksperymencie podejście domenowe. Algorytm ten nie jest najszybszy ze wszystkich badanych, jednak zapewnia najlepsze wykorzystanie rdzeni procesora dla wszystkich badanych instancji (zarówno dla 4 jak i 10 wątków). Ograniczenia, które mogą wpływać na efektywność, to duża liczba komunikacji i synchronizacji.

Największą prędkością wyznaczania liczb pierwszych w podanym zakresie $< m, n >$ okazał się być algorytm k4a. Jest to algorytm sita równoległego funkcyjnego zmodyfikowany odpowiednio, aby uniknąć nadmiernych unieważnień pamięci.