Przetwarzanie Równoległe - Laboratorium

Projekt nr. 1 - Wyznaczanie Liczb Pierwszych

Adrian Rzucidło 148127

Jan Metzler 148137

Termin wymagany: 25.04.2023

Termin rzeczywisty: 07.05.2023

Adresy kontaktowe:

adrian.rzucidlo@student.put.poznan.pl

jan.metzler@student.put.poznan.pl

Grupa L5, termin zajęć: wtorek 8.00(nieparzyste)

Niniejsza wersja sprawozdanie jest wersją pierwszą – niepoprawianą.

Opis projektu

Celem projektu była realizacja zadania polegającego na napisaniu algorytmu pozwalającego na wyznaczenie wszystkich liczb pierwszych w podanym zakresie. Algorytm miał zostać zrealizowany za pomocą kilku różnych metod, różniących się między sobą wykorzystaniem równoległości oraz metodą wyznaczania liczb pierwszych.

Aby osiągnąć wyniki jak najbardziej zbliżone do optymalnych próbowaliśmy różnych podejść do rozwiązywania równoległego - w sprawozdaniu zamieściliśmy zarówno wersję końcową kodu jak i wersje przejściowe.

Projekt został zrealizowany na platformę Windows, w języku C++, z wykorzystaniem biblioteki Open MP oraz wbudowanych bibliotek języka.

Całość kodu źródłowego znajduje się w repozytorium dostępnym przez portal github pod adresem [JanMetz/Parallel-Computing-OMP (github.com)]

Prezentacja przygotowanych wariantów kodu

Słowniczek pojęć:

- mMinNum - dolna granica przedziału, którego sprawdzenia życzył sobie użytkownik
- mMaxNum - górna granica przedziału, którego sprawdzenia życzył sobie użytkownik
- mThreadsNum - maksymalna dostępna na maszynie ilość wątków

1. Przetwarzanie sekwencyjnie, wersja z dodawaniem:

Kod:

```
28
29   std::vector<int> ErastotenesSieve::findPrimesSequential_add(const int max)  const
30   {
31       std::vector<bool> isPrime(max - 1, true);
32
33       for (int divider = 2; divider <= max; ++divider)
34       {
35           if (isPrime[divider - 2] == false)
36               continue;
37
38           removeMultiples(divider, isPrime, 2);
39       }
40
41       std::vector<int> startingPrimes;
42       fillPrimesList(isPrime, startingPrimes, 2);
43
44       return startingPrimes;
45   }
```

1. Kod sekwencyjny, dodawanie

Wyjaśnienie:

Wykorzystywany jest algorytm sita Eratostenesa. Z wektora wartości typu bool, w której wartość true na indeksie X oznacza, że liczba X jest pierwsza, a wartość *false*, że nie jest, wykreślane są kolejne wielokrotności każdej liczby z zakresu od 2 do pierwiastka kwadratowego z maksymalnej liczby jakiej poszukujemy. Następnie tabela ta jest interpretowana w celu otrzymania z niej czytelnej dla użytkownika listy liczb pierwszych. Interpretacja polega na translacji indeksów, pod którymi znajdują się wartości true na konkretne liczby pierwsze będące wartościami indeksów.

Opis przetwarzania:

Przetwarzanie odbywa się sekwencyjnie, wszystko jest wykonywane przez jeden wątek.

2. Przetwarzanie sekwencyjne, wersja z dzieleniem:

Kod:

```cpp
bool ErastotenesSieve::isPrime_div(const int number)
{
    const int limit = floor(sqrt(number));
    int potential_div = 2;

    while (potential_div <= limit)
    {
        if (number % potential_div == 0)
            return false;

        ++potential_div;
    }

    return true;
}

std::vector<int> ErastotenesSieve::findPrimesSequential_div() const
{
    std::vector<int> primes;
    for (int i = mMinNum; i < mMaxNum; ++i)
    {
        if (ErastotenesSieve::isPrime_div(i))
            primes.emplace_back(i);
    }

    return primes;
}
```

2. Kod sekwencyjny, dzielenie

Wyjaśnienie:

Wykorzystywany jest algorytm polegający na sprawdzeniu każdej liczby z zadanego przedziału pod kątem podzielności w celu determinacji jej przynależności do zbioru liczb pierwszych. Każda liczba z zakresu jest sprawdzana przez funkcję *isPrime_div*. Funkcja udziela odpowiedzi czy zadana liczba jest pierwsza czy nie jest. Sprawdzanie odbywa się poprzez monitorowanie reszty z dzielenia zadanej liczby przez liczby z przedziału od 2 do pierwiastka kwadratowego z zadanej liczby. Jeżeli reszta z dzielenia zadanej liczby przez którąkolwiek z liczb z wyznaczonego przedziału wynosi 0 to zwracana jest wartość wskazująca na brak przynależności sprawdzanej liczby do zbioru liczb pierwszych. Jeżeli reszta z dzielenia w żadnym wypadku nie wyniesie 0, zwracana jest wartość potwierdzająca przynależność sprawdzanej liczby do zbioru liczb pierwszych. Jeżeli dana liczba przynależy do zbioru liczb pierwszych to jest umieszczana w wektorze wynikowym *primes*.

Opis przetwarzania:

Przetwarzanie odbywa się sekwencyjnie, wszystkie operacje są wykonywane przez jeden wątek.

Kolejne wersje kodu różnią się jedynie dyrektywami OMP dlatego tylko pierwsza wersja zawierać będzie wyjaśnienie kodu. Wersje następne będą zawierały jedynie opis przetwarzania.

3.1 Przetwarzanie równoległe metodą dodawania, podejście domenowe wer. 1.

Kod:

```
96  std::vector<int> ErastotenesSieve::findPrimesDomain() const
97  {
98      const std::vector<std::vector<int>> ranges{getRanges()};
99      const std::vector<int> startingPrimes{findPrimesSequential_add(static_cast<int>(sqrt(mMaxNum)))};
100
101     std::vector<std::vector<int>> primesMulti(mThreadsNum);
102
103     #pragma omp parallel
104     {
105         const int threadID = omp_get_thread_num();
106
107         const int lowerLimit = ranges[threadID][0];
108         const int upperLimit = ranges[threadID][1];
109
110         std::vector<bool> isPrime((upperLimit - lowerLimit + 1), true);
111         for (const auto &prime : startingPrimes)
112         {
113             int i = 0;
114             while (prime * i < lowerLimit)
115                 ++i;
116
117             const int offset = prime * i - lowerLimit;
118             for (int multiple = 0; multiple + offset < isPrime.size(); multiple += prime)
119                 isPrime[offset + multiple] = false;
120         }
121
122         std::vector<int> primes;
123         fillPrimesList(isPrime, primes, lowerLimit);
124
125         primesMulti[threadID] = primes;
126     }
127
128     std::vector<int> primes;
129     for (const auto &prime : startingPrimes)
130         if (prime >= mMinNum)
131             primes.emplace_back(prime);
132
133     for (int i = 0; i < mThreadsNum; ++i)
134         primes.insert(primes.end(), primesMulti[i].begin(), primesMulti[i].end());
135
136     return primes;
137 }
138
```

3. Kod domenowy 1.

Wyjaśnienie:

Każdy wątek otrzymuje do sprawdzenia pod kątem przynależności do zbioru liczb pierwszych wycinek całego zbioru, którego sprawdzenia życzył sobie użytkownik. Z takiego wycinka usuwane są wielokrotności liczb pierwszych. Liczby, które nie zostaną wykreślone, są liczbami pierwszymi.

W kodzie występuje parę zmiennych.

Zmienna *ranges* jest stała i zawiera wektor wektorów opisujących zakresy liczb, którymi mają się zajmować poszczególne wątki. Zmienna *startingPrimes* zawiera wektor liczb pierwszych z zakresu od 2 do pierwiastka z *mMaxNum*, na podstawie których wykonywane jest wykreślanie wielokrotności. Zmienna *primesMulti* jest współdzielonym wektorem wektorów, do którego każdy wątek przypisuje do określonej komórki wektor liczb pierwszych wyznaczonych dla jego zakresu znajdujący się w zmiennej prywatnej *primes*.

Zmienne *startingPrimes, mMaxNum, mMinNum, primesMulti* oraz *ranges* są współdzielone przez wątki.

W obszarze równoległym każdy wątek ma swoje własne, prywatne zmienne: *threadID, lowerLimit, upperLimit, prime* oraz *isPrime*.

ThreadID określa identyfikator wątku. *LowerLimit* określa minimalną liczbę z zakresu, którym dany wątek ma się zajmować. Analogicznie, *upperLimit* określa górną granicę zakresu, którym dany wątek ma się zajmować. *IsPrime* to wektor, w którym każdy wątek przechowuje swoją listę liczb pierwszych. Prime to wektor, w którym znajdują się wszystkie liczby pierwsze odkryte przez dany wątek.

Na końcu, poza sekcją równoległą, tworzony jest wektor wynikowy *primes*. Są do niego dodawane liczby pierwsze znajdujące się w *startingPrimes* (ponieważ wykreślane są wszelkie, w tym również pierwsza, wielokrotności każdej z znajdujących się tam liczb). Następnie następuje dodawanie do wektora wynikowego *primes* wartości zgromadzonych w kolejnych komórkach współdzielonego przez wątki w poprzednim etapie przetwarzania wektora wektorów *primesMulti*. Poza sekcją równoległą przetwarzanie odbywa się w trybie sekwencyjnym.

Opis przetwarzania:

Dyrektywa *pragma omp parallel* sprawia, że blok kodu następujący bezpośrednio po niej staje się regionem współbieżności. Oznacza to, że każdy wątek wykonuje równolegle oraz niezależnie od siebie znajdującą się w nim część kodu.

W pierwszych 20 linijkach kodu równoległego nie występuje bezpośrednia synchronizacja ani operacje zapisu do pamięci

współdzielonej - każdy wątek ma swoją prywatną kopię tablicy *isPrime* i swój prywatny wektor *primes*. W ostatnich linijkach kodu równoległego wymagana jest synchronizacja dostępu, gdyż następuje w nich zapis wyznaczonego, prywatnego wektora *primes* do współdzielonego między wątkami wektora wektorów *primesMulti*. Synchronizacja jest wymagana aby uniknąć problemów z zapisem, gdy wiele wątków próbuje równocześnie zmodyfikować ten sam wektor.

3.2. Przetwarzanie równoległe metodą dodawania, podejście domenowe wer. 2.

Kod:

```cpp
std::vector<int> ErastotenesSieve::findPrimesDomain() const
{
    const std::vector<std::vector<int>> ranges{getRanges()};
    const std::vector<int> startingPrimes{findPrimesSequential_add(static_cast<int>(sqrt(mMaxNum)))};

    std::vector<std::vector<int>> primesMulti(mThreadsNum);

    #pragma omp parallel firstprivate(ranges, startingPrimes) shared(primesMulti)
    {
        const int threadID = omp_get_thread_num();

        const int lowerLimit = ranges[threadID][0];
        const int upperLimit = ranges[threadID][1];

        std::vector<bool> isPrime((upperLimit - lowerLimit + 1), true);
        for (const auto &prime : startingPrimes)
        {
            int i = 0;
            while (prime * i < lowerLimit)
                ++i;

            const int offset = prime * i - lowerLimit;
            for (int multiple = 0; multiple + offset < isPrime.size(); multiple += prime)
                isPrime[offset + multiple] = false;
        }

        std::vector<int> primes;
        fillPrimesList(isPrime, primes, lowerLimit);

        primesMulti[threadID] = primes;
    }

    std::vector<int> primes;
    for (const auto &prime : startingPrimes)
        if (prime >= mMinNum)
            primes.emplace_back(prime);

    for (int i = 0; i < mThreadsNum; ++i)
        primes.insert(primes.end(), primesMulti[i].begin(), primesMulti[i].end());

    return primes;
}
```

4. Kod domenowy 2.

Opis przetwarzania - różnice względem wersji 1:

Dyrektywa *firstprivate* sprawia, że każdy wątek tworzy swoje prywatne kopie zmiennych wymienionych jako jej argumenty. Kopie te są inicjalizowane wartościami oryginalnych zmiennych. Pozwala to na upewnienie się o braku synchronizacji między wątkami przy dostępie do tych zmiennych. Jako prywatne kopie zostały określone zmienne *ranges oraz startingPrimes*.

Dyrektywa *shared* sprawia, że wątki uznają jej argumenty jako zmienne współdzielone. W naszym przypadku taką zmienną jest *primesMulti*, które zostaje przez to udostępniona do zapisu i odczytu przez każdy

wątek. Wszelkie działania na tej zmiennej wykonywane są na tym samym obszarze pamięci.

Kolejne wersje kodu różnią się jedynie dyrektywami OMP, dlatego tylko pierwsza wersja zawierać będzie wyjaśnienie kodu. Wersje następne będą zawierały jedynie opis przetwarzania.

4.1. Przetwarzanie równoległe metodą dodawania, podejście funkcyjne wer. 1.

Kod:

```
150  std::vector<int> ErastotenesSieve::findPrimesFunctional() const
151  {
152      const std::vector<int> startingPrimes{findPrimesSequential_add(static_cast<int>(sqrt(mMaxNum)))};
153
154      std::vector<std::vector<bool>> isPrimeMulti(mThreadsNum);
155      for (auto &row : isPrimeMulti)
156          row = std::vector<bool>(mMaxNum - mMinNum + 1, true);
157
158      #pragma omp parallel for firstprivate(startingPrimes) shared(isPrimeMulti)
159      for (int i = 0; i < startingPrimes.size(); ++i)
160      {
161          const int prime = startingPrimes[i];
162          const int threadID = omp_get_thread_num();
163          for (int multiple = prime; multiple - mMinNum < isPrimeMulti[threadID].size(); multiple += prime)
164              isPrimeMulti[threadID][multiple - mMinNum] = false;
165      }
166
167      std::vector<int> primes;
168      for (const auto& prime : startingPrimes)
169          if (prime >= mMinNum)
170              primes.emplace_back(prime);
171
172      const auto combined = combinePrimesLists(isPrimeMulti, mMinNum);
173      primes.insert(primes.end(), combined.begin(), combined.end());
174
175      return primes;
176  }
177
```

5. Kod funkcyjny 1.

Wyjaśnienie działania kodu:

Każdy wątek otrzymuje swoją kopię pełnego zbioru, którego sprawdzenia życzył sobie użytkownik. Z takiego zbioru usuwane są wielokrotności konkretnych, przypisanych dynamicznie do wątku liczb pierwszych. Przydział liczb pierwszych, których wielokrotności należy wykreślać do poszczególnych wątków odbywa się automatycznie. Po zakończeniu procesu usuwania wielokrotności wszystkie prywatne kopie zbioru są scalane w jedną korzystając z funkcji logicznej *AND*. Z takiego zbioru wyznaczane są konkretne liczby pierwsze.

W kodzie występuje parę zmiennych.

Zmienna *startingPrimes* zawiera wektor liczb pierwszych z zakresu od 2 do pierwiastka z *mMaxNum*, na podstawie których wykonywane jest wykreślanie wielokrotności.

Zmienna *isPrimeMulti* jest współdzielonym między wątkami wektorem wektorów, na którego podwektorze znajdującym się w komórce pod indeksem równym swojemu numerowi identyfikacyjnemu (pobieranemu

za pomocą funkcji *omp_get_thread_num()*) dany wątek wykonuje operacje wykreślania.

Na końcu poza sekcją równoległą tworzony jest wektor wynikowy identyfikowany zmienną *isPrime*. Jego zawartość jest wynikiem wykonania funkcji logicznej *AND* na odpowiadających komórkach w wektorach znajdujących się w zmiennej *isPrimeMulti*.

Po wykonaniu scalenia poszczególnych wektorów znajdujących się w zmiennej *isPrimeMulti* zmienna *isPrime* jest przetwarzana tak, aby osiągnąć z niej wektor wynikowy *primes*, w którym znajdują się wyłącznie wyznaczone dla zakresu liczby pierwsze.

Poza sekcją równoległą przetwarzanie odbywa się w trybie sekwencyjnym.

Opis przetwarzania:

Dyrektywa *omp parallel* for sprawia, że pętla for wykonywana jest równolegle przez wiele wątków. Podział pracy polega na przydzieleniu do każdego wątku określonej części całości iteracji do wykonania. Dodatkowe parametry podziału można modyfikować za pomocą dyrektywy *schedule*.

Dyrektywa *firstprivate* sprawia, że każdy wątek tworzy swoje prywatne kopie zmiennych wymienionych jako jej argumenty. Kopie te są inicjalizowane wartościami oryginalnych zmiennych. Pozwala to na upewnienie się o braku synchronizacji między wątkami przy dostępie do tych zmiennych. Jako prywatna kopia została określona zmienna *startingPrimes*.

Dyrektywa *shared* sprawia, że wątki uznają jej argumenty jako zmienne współdzielone. W naszym przypadku taką zmienną jest *isPrimeMulti*. Zostaje ona przez to udostępniona do zapisu i odczytu przez każdy wątek. Wszelkie działania na tej zmiennej wykonywane są na tym samym obszarze pamięci.

4.2. Przetwarzanie równoległe metodą dodawania, podejście funkcyjne wer. 2.

Kod:

```
150  std::vector<int> ErastotenesSieve::findPrimesFunctional() const
151  {
152      const std::vector<int> startingPrimes{findPrimesSequential_add(static_cast<int>(sqrt(mMaxNum)))};
153
154      std::vector<std::vector<bool>> isPrimeMulti(mThreadsNum);
155      for (auto &row : isPrimeMulti)
156          row = std::vector<bool>(mMaxNum - mMinNum + 1, true);
157
158      #pragma omp parallel for schedule(guided) firstprivate(startingPrimes) shared(isPrimeMulti)
159      for (int i = 0; i < startingPrimes.size(); ++i)
160      {
161          const int prime = startingPrimes[i];
162          const int threadID = omp_get_thread_num();
163          for (int multiple = prime; multiple - mMinNum < isPrimeMulti[threadID].size(); multiple += prime)
164              isPrimeMulti[threadID][multiple - mMinNum] = false;
165      }
166
167      std::vector<int> primes;
168      for (const auto& prime : startingPrimes)
169          if (prime >= mMinNum)
170              primes.emplace_back(prime);
171
172      const auto combined = combinePrimesLists(isPrimeMulti, mMinNum);
173      primes.insert(primes.end(), combined.begin(), combined.end());
174
175      return primes;
176  }
177
```

6. Kod funkcyjny 2.

Opis przetwarzania - różnice względem wersji 1:

Dyrektywa *schedule* określa sposób podziału pracy w pętli *for* dla kolejnych wątków. Sposób podziału, którego użyliśmy, sprawia, że każdy wątek otrzymuje pewien kawałek danych. Po zakończeniu przetwarzania na tym kawałku “zgłasza” swoją gotowość do otrzymania kolejnej partii, a następnie, o ile są wciąż dostępne kawałki do przydzielenia, cały proces się powtarza. Rozmiary przydzielanych kawałków wraz z postępowaniem iteracji maleją.

Kolejne wersje kodu różnią się jedynie dyrektywami OMP dlatego tylko pierwsza wersja zawierać będzie wyjaśnienie kodu. Wersje następne będą zawierały jedynie opis przetwarzania.

5.1. Przetwarzanie równoległe metodą dzielenia wer. 1.

Kod:

```
139 std::vector<int> ErastotenesSieve::findPrimesDiv() const
140 {
141     std::vector<std::vector<bool>> isPrimeMulti(mThreadsNum);
142     for (auto &row : isPrimeMulti)
143         row = std::vector<bool>(mMaxNum - mMinNum + 1, true);
144
145     #pragma omp parallel for shared(isPrimeMulti)
146     for (int num = mMinNum; num < mMaxNum + 1; ++num)
147         isPrimeMulti[omp_get_thread_num()][num - mMinNum] = ErastotenesSieve::isPrime_div(num);
148
149     return combinePrimesLists(isPrimeMulti, mMinNum);
150 }
151
```

7. Kod równoległy, dzielenie 1.

Wyjaśnienie działania kodu:

Przetwarzanie odbywa się równolegle. Podział pracy jest zapewniony za pomocą dyrektywy *pragma omp parallel for.* Każdy wątek określa czy dana liczba jest pierwsza, czy nie, wywołując funkcję zajmującą się określeniem przynależności do zbioru liczb pierwszych za pomocą metody dzielenia. Funkcja ta jest opisana w opisie znajdowania liczb pierwszych metodą dzielenia w trybie sekwencyjnym.

W kodzie występuje parę zmiennych.

Zmienna *isPrimeMulti* jest współdzielonym między wątkami wektorem wektorów. Dany wątek wykonuje operacje mające na celu określić przynależność do zbioru liczb pierwszych na liczbach reprezentowanych przez kolejne komórki podwektora znajdującego się w komórce wektora *isPrimeMulti* pod indeksem równym swojemu numerowi identyfikacyjnemu, pobieranemu za pomocą funkcji *omp_get_thread_num()*.

Na końcu poza sekcją równoległą tworzony jest wektor wynikowy identyfikowany zmienną *isPrime*. Jego zawartość jest wynikiem wykonania funkcji logicznej *AND* na odpowiadających komórkach w wektorach znajdujących się w zmiennej *isPrimeMulti*.

Po wykonaniu scalenia poszczególnych wektorów znajdujących się w zmiennej *isPrimeMulti* zmienna *isPrime* jest przetwarzana tak, aby

osiągnąć z niej wektor wynikowy *primes*, w którym znajdują się wyłącznie wyznaczone dla zakresu liczby pierwsze.

Poza sekcją równoległą przetwarzanie odbywa się w trybie sekwencyjnym.

Opis przetwarzania:

Dyrektywa *omp parallel for* sprawia, że pętla for wykonywana jest równolegle przez wiele wątków. Podział pracy polega na przydzieleniu do każdego wątku określonej części całości iteracji do wykonania. Dodatkowe parametry podziału można modyfikować za pomocą dyrektywy *schedule*.

Dyrektywa *shared* sprawia, że wątki uznają jej argumenty jako zmienne współdzielone. W naszym przypadku taką zmienną jest *isPrimeMulti*. Zostaje ona przez to udostępniona do zapisu i odczytu przez każdy wątek. Wszelkie działania na tej zmiennej wykonywane są na tym samym obszarze pamięci.

5.2. Przetwarzanie równoległe metodą dzielenia wer. 2.

Kod:

```
139  std::vector<int> ErastotenesSieve::findPrimesDiv() const
140  {
141      std::vector<std::vector<bool>> isPrimeMulti(mThreadsNum);
142      for (auto &row : isPrimeMulti)
143          row = std::vector<bool>(mMaxNum - mMinNum + 1, true);
144
145      #pragma omp parallel for schedule(guided) shared(isPrimeMulti)
146      for (int num = mMinNum; num < mMaxNum + 1; ++num)
147          isPrimeMulti[omp_get_thread_num()][num - mMinNum] = ErastotenesSieve::isPrime_div(num);
148
149      return combinePrimesLists(isPrimeMulti, mMinNum);
150  }
151
```

8. Kod równoległy, dzielenie 2..

Opis przetwarzania - różnice względem wersji 1:

W tej wersji kodu używamy dyrektywy *schedule*, która określa sposób podziału pracy w pętli for dla kolejnych wątków. Sposób podziału, którego użyliśmy, sprawia, że każdy wątek otrzymuje pewien kawałek danych. Po zakończeniu przetwarzania na tym kawałku “zgłasza” swoją gotowość do otrzymania kolejnej partii, a następnie, o ile są wciąż dostępne kawałki do przydzielenia, cały proces się powtarza. Rozmiary przydzielanych kawałków wraz z postępowaniem iteracji maleją.

Prezentacja wyników przetwarzania

Opis eksperymentu:

Wszystkie powyższe wersje kodu zostały przetestowane przy pomocy oprogramowania VTUNE, dla przedziałów:

a) 2 - 1000000000
b) 2 - 500000000
c) 500000000 - 1000000000

Ze względu na bardzo długi czas przetwarzania, przedziały dla kodów wykonywanych metodą dzielenia(2. i 5.) zostały zmniejszone 10 krotnie.

Opis parametrów platformy:

Procesor:

- Nazwa: Intel i7 – 1260P
- Liczba rdzeni fizycznych: 12
- Liczba rdzeni logicznych: 16
- Oznaczenie typu procesora: P
- Wielkość pamięci cache procesora: 18 MB

System operacyjny:

- Nazwa: Microsoft Windows
- Wersja: 11

Opis programu profilującego Intel VTUNE:

Oprogramowanie Intel VTUNE służącym do badania wydajności oprogramowania, program ten działa w oparciu o Performance Monitoring Units (PMU), który bada wydarzenia odbywające się w procesorze, po przeprowadzeniu analizy tych wydarzeń oprogramowania prezentuje wiele przydatnych parametrów opisujących wydajność aplikacji na różnych płaszczyznach (np. płaszczyźnie sprzętu, pamięci itp.).

Legenda tabeli:

a) Kod wersji (równoznaczny kodom z powyższych punktów) wraz z kodem instancji(literki równoznaczne z przedziałami powyżej)
b) Czas przetwarzania - w sekundach
c) Liczba instrukcji asemblera
d) Liczba cykli procesorów w czasie wykonywania badanego kodu
e) Udział procentowy wykorzystanych zasobów procesora do przetwarzania kodu
f) „Ograniczenie wejścia” - udział procentowy w ograniczeniu efektywności przetwarzania części wejściowej
g) „Ograniczenie wyjścia” - udział procentowy w ograniczeniu efektywności przetwarzania części wyjściowej procesora
h) „Ograniczenie systemu pamięci” - udział procentowy w ograniczeniu efektywności przetwarzania systemu pamięci
i) „Ograniczenie jednostek wykonawczych” - udział procentowy w ograniczeniu efektywności przetwarzania jednostek wykonawczych procesora
j) Efektywne wykorzystanie rdzeni fizycznych procesora – „effective physical core utilization”
k) Przyspieszenie przetwarzania równoległego dla badanego wariantu kodu równoległego (iloraz czasu przetwarzania najlepszego dostępnego przetwarzania sekwencyjnego(w tym samym przedziale liczb) oraz czasu przetwarzania równoległego, ze względu na inny, wspólny przedział wariant 5. porównano z wariantem 2.)
l) Prędkość przetwarzania liczona jako liczba przetestowanych liczb w jednostce czasu z uwzględnieniem wielkości badanego zbioru liczb
m) Efektywność przetwarzania równoległego jako iloraz przyspieszenia przetwarzania równoległego i liczby użytych w przetwarzaniu procesorów fizycznych

Tabela:

a)	b)	c)	d)	e)	f)	g)	h)	i)	j)	k)	l)	m)
1.a	18,509	97266624000	63672960000	24,0%	3,0%	57,7%	45,4%	12,3%	7,7%	x	54027770	x
1.b	8,277	48212736000	292705920000	26,6%	3,4%	62,5%	46,8%	15,7%	8,5%	x	60408360	x
1.c	17,208	97251648000	603158400000	27,2%	3,3%	58,8%	43,5%	15,3%	8,1%	x	29056252	x
2.a	70,537	377,794560000	284776128000	25,2%	2,2%	72,5%	0,5%	72%	7,8%	x	1417695	x
2.b	25,914	140901696000	106097472	24,9%	2,2%	71,8%	0,5%	71,3%	8%	x	1929458	x
2.c	44,5	236860416000	179013120000	24,7%	2,2%	72,3%	0,4%	71,9%	7,5%	x	1123595	x
3.1 a	12,795	83830656000	121712448000	20,7%	9,5%	69%	45,4%	23,6%	16,2 %	1.44	78155529	0.12
3.1 b	2,08	25274496000	194213760000	36,4%	10,1 %	52,9%	43,8%	9,1%	15,3 %	3.979	240384614	0.331
3.1 c	2,918	32093568000	360222720000	26,7%	8,4%	60,6%	40,6%	20%	18%	5.89	171350239	0.490
3.2 a	3,016	149740032000	102226176000	38,7%	4,7%	54,3%	42,6%	11,7%	58%	6.136	331564986	0.511
3.2 b	1,122	73998912000	322333440000	59,8%	6%	25,4%	17,4%	8%	47%	7.37	445632796	0.614
3.2 c	2,04	158171520000	634033920000	55,3%	6,1%	30,9%	18,3%	12,6%	54%	8.43	245098039	0.7
4.1 a	34,92	113831328 0000	879443136000	39,1%	9%	50%	40,4%	9,7%	47,3 %	0.53	28636884	0.04
4.1 b	15,587	562945344000	366073344000	45,5%	10,3 %	43%	32,3%	10,8%	43,8 %	0.531	32078013	0.04

4.1 c	7,838	98397312000	291682560000	49,1%	30,3 %	13,8%	8,6%	5,2%	7,6%	2.19	63791783	0.18
4.2 a	47,8	1140721920000	123512563200 0	28,5%	6,9%	63,%	55,8%	7,7%	49,5 %	0.387	20920502	0.032
4.2 b	15,462	562965312000	379768896000	43,5%	9,7%	44,1%	34,3%	9,8%	44,3 %	0.535	32337343	0.535
4.2 c	7,719	98599488000	292980480000	53,4%	32,2 %	14,8%	10,3%	4,5%	8,1%	2.22	64775229	0.185
5.1 a	15,3	398488896000	445328832000	27,3%	6,7%	64,7%	0,5%	64,3%	58%	4.61	6535947	0.384
5.1 b	6,36	151377408000	166368384000	28,4%	7,4%	66,1%	0,4%	65,7%	55%	4.074	7861635	0.339
5.1 c	8,05	247430976000	303311424000	28,1%	8,7%	63%	0,2%	62,7%	65%	5.52	6211180	0.46
5.2 a	13,05	412571328000	500310720000	28,5%	14,9 %	54,3%	0,9%	53,5%	68%	5.405	7662835	0.450
5.2 b	4,766	158223936000	188310720000	29,9%	15,6 %	52,6%	1%	51,6%	66%	5.56	10490977	0.463
5.2 c	7,55	254272512000	315741504000	27,7%	14,2 %	55,4%	0,5%	55%	71%	5.89	6622516	0.490

9. Tabela wynikowa

Wnioski

Wniosek nr. 1:

Wariantem kodu wyznaczającym liczby pierwsze z zadanego zakresu najszybciej jest wariant wykorzystujący przetwarzanie równoległe w domenowej konfiguracji podziału pracy, która odbywała się z wykorzystaniem metody dodawania. Wariant ten osiągnął również najwyższy poziom równoległości oraz najlepiej wykorzystywał zasoby systemu.

Powodem takiego stanu rzeczy jest niski stopień wymaganej między wątkami synchronizacji - wszystkie operacje dzieją się na prywatnych zmiennych, a do zmiennej współdzielonej dostęp wymagany jest wyłącznie aby przypisać do jej określonej części dane ze zmiennej prywatnej. Przypisywanie takie odbywa się tylko raz dla jednego wątku - stosunek zapisów do obszaru synchronizowanego do wyliczeń przeprowadzanych przez dany wątek samodzielnie jest więc bardzo niski.

Wniosek nr. 2:

Wyznaczanie liczb pierwszych metodą dodawania jest wielokrotnie szybsze niż metodą dzielenia.

Jest tak, ponieważ metoda dodawania nie wymaga wykonywania operacji modulo, która wykonanie jest wysoce kosztowne czasowo dla procesora.

Wniosek nr. 3:

Wariant kodu wykonujący przetwarzanie równoległe funkcyjne metodą dodawania okazał się wysoce nieoptymalny - zarówno pod kątem czasu przetwarzania jak i z perspektywy wykorzystania procesora i podziału pracy.

Powodem takiego zachowania może być fakt konieczności synchronizacji zawartości tablicy, na której wykonywane były obliczenia między wątkami. Stosunek zapisów do współdzielonego obszaru pamięci do wyliczeń przeprowadzanych przez dany wątek samodzielnie jest bardzo wysoki.

Wniosek nr. 4:

Najczęściej testowane metody osiągały najlepsze wartości czasu dla przedziału b., ze względu na to, że przedział ten zawierał najmniejsze liczby.

Dodatkowe informacje

Po zgłębieniu tematu zdecydowaliśmy się na użycie kontenera std::vector z biblioteki wbudowanej STL zamiast klasycznych tablic w stylu czystego języka C. Decyzję tę podjęliśmy ze względu na znikome różnice w wydajności oraz na znacznie zwiększony komfort użytkowania.

Zamiast drugiej tabeli prezentującej najlepsze wyniki zdecydowaliśmy się na oznaczenie najlepszych wyników kolorem zielonym w głównej tabeli prezentującej dane.

Wersja kodu dostarczona wraz z sprawozdaniem zawiera wersje kodów o numerach X.2 dla kodów równoległych, ze względu na to, iż te kody zostały naszymi ostatecznymi wersjami.

Instrukcja użycia kodu: program należy włączać z następującymi argumentami

1. Początek przedziału (liczba całkowita większa równa 2)
2. Koniec przedziału (liczba całkowita większa od początku przedziału)
3. Sposób prezentacji danych, możliwe wartości:
    a) 1 - oznacza brak prezentacji wyników
    b) 2 - oznacza prezentację ilości znalezionych liczb pierwszych
    c) 3 - oznacza prezentację ilości oraz wszystkich znalezionych liczb pierwszych
4. Sposób realizacji przetwarzania, możliwe wartości:
    a) seq_a - wersja kodu 1.
    b) seq_d - wersja kodu 2.
    c) dom - wersja kodu 3.2
    d) fun - wersja kodu 4.2
    e) div - wersja kodu 5.2

Przykładowe uruchomienie programu “ ./es.exe 2 1000 2 seq_a “.