Przetwarzanie Równoległe - laboratorium

PROJEKT 1 OMP studia dzienne, kierunek Informatyka
Wersja pierwsza

Paulina Pacura L8 142179 - środa 9:45 paulina.pacura@student.put.poznan.pl
Michał Cichy L4 132203 - poniedziałek 9:45 michal.a.cichy@student.put.poznan.pl

Termin wymagany oraz rzeczywisty: 27 Kwietnia 2020

Krótki opis realizowanego zadania

Problem znajdowania liczb pierwszych w podanym przedziale M..N w celu analizy efektywności przetwarzania równoległego realizowanego w komputerze równoległym z procesorem wielordzeniowym z pamięcią współdzieloną. Projekt przedstawia różne wersje rozwiązania problemu znajdowania liczb pierwszych oraz analizę ich efektywności. Zrealizowane wersje algorytmów to:
Sekwencyjne: wyszukiwanie liczb pierwszych w podanym zakresie metodą sprawdzania czy jest podzielna przez liczby mniejsze, gdzie ich zakres można ograniczyć do pierwszej w kolejności liczby, która jest dzielnikiem badanej liczby i liczb mniejszych od pierwiastka kwadratowego z badanej liczby, oraz wyszukiwanie liczb pierwszych metodą wykreślania ze zbioru liczb do zbadania liczb będących liczbami złożonymi (jest to metoda Sita, które służy do odsiania nieinteresujących nas liczb złożonych).
Równoległe: drugi opisany powyżej algorytm przekształcany jednak tak, aby uzyskać różne warianty zrównoleglenia realizując podejście domenowe z wykreśleniem wszystkich wymaganych wielokrotności podzbioru liczb pierwszych oraz podejście funkcyjne z wykreśleniem wielokrotności mających ograniczony zakres wartości, ale będących wielokrotnościami wszystkich wymaganych liczb. Pierwszy z algorytmów również poddawany jest zrównoleglaniu.
Następnie w oparciu o tak napisane kody dokonywana jest analiza miar efektywności obliczeń z wykorzystaniem wskazanego oprogramowania dokonującego pomiarów w wyznaczonych konfiguracjach. Kod testowany jest dla wyszczególnionych w zadaniu instancji z badaniem działania przy wykorzystaniu wskazanych ilości wątków. Wartości zakresów w programie zostały dobrane tak, by zapewnić przetwarzanie trwające odpowiednio długo dla uzyskania poprawności metody zbierania informacji o przebiegu przetwarzania przez program oceny jakości, lecz nie dłużej niż około 2 minuty dla dowolnej liczby procesorów i wątków (zadanie wspomina o 1 minucie jednak dla sekwencyjnej wersji algorytmu dzielącego przedłużyliśmy ten czas).

1 Opis wykorzystywanego systemu obliczeniowego

Procesor	Intel Core
Liczba procesorów fizycznych	2
Liczba procesorów logicznych	4
Liczba uruchamianych wątków	4
Oznaczenie typu procesora	i5-7200 x86
Wielkość pamięci podręcznych	3 MB Intel® Smart Cache
Wersja systemu operacyjnego	Windows 10
Oprogramowanie do przygotowania kodu	Visual Studio 2019
Oprogramowanie do przeprowadzenia testów	Intel® Parallel Studio XE Cluster Edition for Windows 2020

W każdym programie zastosowano OpenMP Support oraz wykorzystano wersję Release.

2 Prezentacja przygotowanych wariantów kodów

2.1 Wersje sekwencyjne

2.1.1 SEQ_1 - dzielenie

```
1  namespace SEQUENTIAL {
2          vector<int> primesDivide(int a, int b, bool output = true) {
3                  vector<int> results;
4                  results.reserve(b - a + 1);
5                  for (int i = a; i <= b; i++) {
6                          if (isPrime(i)) {
7                                  results.push_back(i);
8                          }
9                  }
10                 return results;
11         }
12 }
```

Figure 1: SEQ_1 - kod funkcji *primeDivide(int a, int b)* pierwszej wersji sekwencyjnej

```
1  bool isPrime(int number) {
2          int divider = 2;
3          int upperBound = int(sqrt(number));
4          while (divider <= upperBound) {
5                  if (number % divider == 0) {
6                          return false;
7                  }
8                  divider++;
9          }
10         return true;
11 }
```

Figure 2: SEQ_1 - kod funkcji *isPrime(int number)* pierwszej wersji sekwencyjnej

Metoda: dzielenie badanej liczby przez liczby pierwsze i badanie wartości reszty z dzielenia. Obraz przedstawiony na Figure 1: *primeDivide(int a, int b)* przedstawia funkcję tworzącą wektor zawierający liczby pierwsze odnalezione we wskazanym przedziale, gdzie kluczowym momentem klasyfikacyjnym dla danej liczby jest wywołanie funkcji *isPrime(int number)* wewnątrz pętli while (line 31-32), która to funkcja widoczna jest na obrazie Figure 2. W funkcji *isPrime(int number)* jako górna granica podzielnika danej liczby wybierany jest jej pierwiastek kwadratowy i do takiego zakresu badane są kolejne od 2 wzrastające liczby całkowite, gdzie w przypadku natrafienia na resztę z dzielenia opracowywanej liczby przez kolejny dzielnik równą zero zwracana jest informacja przecząca o przynależności liczby do zbioru liczb pierwszych. W przypadku braku znalezienia takiej reszty z dzielenia zwracana jest informacja o przynależności liczby do zbioru liczb pierwszych.

2.1.2 SEQ_2 - wykreślanie (Sito Eratostenesa)

```
namespace SEQUENTIAL {
        vector<int> primesSieve(int a, int b) {
                int upperBound = int(sqrt(b));
                vector<int> firstPrimes = SEQUENTIAL::primesDivide(2, upperBound, false);
                CHECK* checked = new CHECK[b + 1]{ false };
                for (int i = 0; i < firstPrimes.size(); i++) {
                        int multiplicity = firstPrimes[i] * 2;
                        while (multiplicity <= b) {
                                checked[multiplicity] = true;
                                multiplicity += firstPrimes[i];
                        }
                }
                vector<int> primes;
                primes.reserve(b - a + 1);
                for (int i = a; i <= b; i++) {
                        if (!checked[i]) {
                                primes.push_back(i);
                        }
                }
                delete[] checked;
                return primes;
        }
}
```

Figure 3: SEQ_2 - kod programu drugiej wersji sekwencyjnej

Metoda: sito Eratostenesa czyli usuwanie ze zbioru liczb wielokrotności liczb od najmniejszej. Algorytm przesiewa z zadanego przedziału liczby w taki sposób, że zostają tylko te, które nie są wielokrotnością liczb wcześniejszych, zatem są to liczby pierwsze. W przedstawionej na rysunku Figure 3 funkcji ustawiany jest limit górnej granicy jako pierwiastek ostatniej liczby ze wskazanego zakresu. Następnie dla tej liczby pobierana jest tablica liczb pierwszych z wykorzystaniem napisanej wcześniej funkcji *primeDivide(int a, int b)* aby uzyskać zbiór na podstawie którego dla każdej liczby obliczamy jej podwojenie i je, oraz każdą kolejną wielokrotność dodajemy do tablicy sprawdzonych już liczb niebędących liczbami pierwszymi. Następnie, mając wypełnioną już tablicę sprawdzonych tworzony jest wektor liczb pierwszych na podstawie porównywania obecności kolejnych liczb całkowitych ze wskazanego zakresu w tablicy sprawdzonych, i w przypadku braku takiej obecności dodawania sprawdzanej liczby do wektora liczb pierwszych.

2.2 Wersje równoległe

2.2.1 Dyrektywy i klauzule Open MP

W kodzie programów równoległych wykorzystaliśmy poznane we wcześniejszym projekcie dyrektywy OpenMP, poprzez które w sposób jawny wskazuje się, które fragmenty programu powinny być wykonywane równolegle.

» *omp_get_max_threads()* - zwraca górną granicę dostępnej liczby wątków.
» *omp_set_num_threads(x)* - ustawia ilość wątków, które wezmą udział w wykonaniu regionów równoległych.
» *omp_get_thread_num()* - pobiera numer bieżącego wątku.
» *pragma omp parallel* - definiuje region równoległy: obszar kodu, który wykonany będzie równolegle przez wiele wątków.
» *pragma omp for* - powoduje, że praca wykonywana wewnątrz regionu równoległego podzielona będzie pomiędzy wątki.
» *pragma omp for schedule(dynamic)* - każdemu wątkowi przypisywana jest określona parametrem (domyślnie =1) liczba iteracji. Po wykonaniu obliczeń wątki otrzymują kolejną część iteracji do wykonania.

2.2.2 Problemy poprawnościowe

Wyścig
Wyścig to termin, którego używa się dla oznaczenia sytuacji,kiedy kilka (conajmniej dwa) wątków wykonuje operację na zasobach dzielonych. Taka sytuacja prowadzi do nadpisywania danych w przypadkowej kolejności i w rezultacie uzyskania niedeterministycznych, prawdopodobnie niepoprawnych wyników: ostateczny wynik operacji jest zależny od momentu jej realizacji. Wystąpienie wyścigu może być konsekwencją braku ostrożności w dostępie do współdzielonych zmiennych. Aby zapobiec warunkom wyścigu należy wykorzystać mechanizm niedopuszczający do sytuacji, gdy więcej niż jeden proces zyskuje dostęp do zasobów dzielonych w tym samym czasie - wzajemne wykluczanie pozwala na zachowanie własności bezpieczeństwa. Dostępne są dyrektywy umożliwiające synchronizację dostępu wątków do sekcji krytycznej oraz możliwy jest odpowiedni podział pracy wątków. Problemy z poprawnością opisane zostały w dalszych podrozdziałach, osobno dla każdego fragmentu kodu. Na końcu rozdziału przedstawiona została tabela ukazująca zestawienie problemów poprawnościowych i efektywnościowych dla wszystkich programów.

Własność żywotności
Polega na zapewnieniu możliwości spełnienia warunku koniecznego do wykonania każdego procesu w programie, aby nie dopuścić do sytuacji zagłodzenia procesu z powodu braku wystąpienia sytuacji pozwalającej na uzyskanie dostępu do zasobu. Problem ten nie występuje w przedstawionych poniżej programach z powodu braku sytuacji gdzie wątek mógłby nie doczekać się dostępu do zasobu.

2.2.3 Problemy efektywnościowe

False sharing
Zjawisko fałszywego współdzielenia danych (false sharing) polega na wielokrotnym unieważnianiu linii pamięci w pamięci podręcznej procesorów, które stają się nieaktualne w wyniku zapisu przez różne procesy, zmiennej znajdującej się w tej samej linii i dany proces musi ponownie pobierać linię z pamięci dla zachowania spójności - konieczność sprowadzenia do pamięci podręcznej procesora realizującego kod wątku aktualnej kopii danych z unieważnionej linii. Eksperyment przeprowadzony w pierwszym zadaniu z Przetwarzania Równoległego wykazał, że długość linii pamięci w wykorzystywanym do testów systemie to 64B. Mając to na uwadze, obserwowaliśmy działanie zaimplementowanych algorytmów. Spostrzeżenia zanotowane zostały w kolejnych podrozdziałach, osobno dla każdego przedstawionego kodu. Na końcu rozdziału przedstawiona została tabela ukazująca zestawienie problemów poprawnościowych i efektywnościowych dla wszystkich programów.

2.2.4 *PAR_DZIEL* - Dzielenie, wersja równoległa

Jako zrównoleglenie algorytmu dzielenia wykorzystaliśmy wprowadzenie dodatkowej tablicy do przechowywania prywatnych obliczonych liczb pierwszych procesu. W regionie równoległym każdy proces oblicza swoją część liczb pierwszych ze wskazanego przedziału, a następnie zapisuje swoje wyniki na odpowiednie miejsca we właściwej wynikowej tablicy przechowującej liczby pierwsze. Do samego zadecydowania czy liczba jest liczbą pierwszą, tak jak w wersji sekwencyjnej wykorzystana została funkcja **isPrime(int a)**.

Wersja naiwna

Wersja naiwna najpierw w sposób sekwencyjny tworzy i rezerwuje miejsce w wektorze do przechowywania wynikowych liczb pierwszych. Następnie w regionie równoległym w pętli sprawdzane są kolejne liczby z podanego zakresu przy użyciu funkcji **isPrime(int a)**. W tej wersji kodu występuje ryzyko wyścigu: równoczesne dodawanie nowych elementów do wektora przez wątki bez synchronizacji powoduje szansę powstawania niepoprawnych wyników. W programie występuje także ryzyko problem False Sharingu: współdzielony wektor wyników modyfikowany jest równocześnie przez wątki, co w przypadku elementów dzielących te same linie pamięci doprowadzić może do unieważnień linii. (Kod przedstawia Figure 4, następna strona)

```
namespace NAIVE {
        vector<int> PARALLEL_primesDivide(int a, int b, bool output = true) {
                vector<int> results;
                results.reserve(b - a + 1);
#pragma omp parallel
                {
#pragma omp for
                        for (int i = a; i <= b; i++) {
                                if (isPrime(i)) results.push_back(i);
                        }
                }
                return results;
        }
```

Figure 4: *PAR_DZIEL_NAIVE* - kod wersji naiwnej równoległego algorytmu dzielenia

Wersja dobra

```
namespace BEST {
        vector<int> PARALLEL_primesDivide(int a, int b, bool output = true) {
                vector<int> results;
                results.reserve(b - a + 1);

                int threadsCount = omp_get_max_threads();
                vector<vector<int>> privateResults(threadsCount, vector<int>());
                for (int i = 0; i < threadsCount; i++) {
                        privateResults[i].reserve(b - a + 1);
                }
#pragma omp parallel
{
#pragma omp for
                        for (int i = a; i <= b; i++) {
                                if (isPrime(i)) {
                                        privateResults[omp_get_thread_num()].push_back(i);
                                }
                        }
                }
                for (int i = 0; i < threadsCount; i++) {
                        results.insert(results.end(), privateResults[i].begin(), privateResults[i].end());
                }
                return results;
        }
```

Figure 5: *PAR_DZIEL_OK* - kod ostatecznej wersji równoległego algorytmu dzielenia

W wersji ostatecznej inaczej zorganizowane zostało zarządzanie wynikami. Program sprawdza aktualną ustawioną ilość wątków funkcją *omp_get_max_threads()* oraz dla każego wątku wyznaczana jest oddzielny wektor do przechowywania prywatnych wyników obliczania liczb pierwszych. Następnie, znów w bloku równoległym, w pętli for tak jak w wersji naiwnej wyznaczane są liczby pierwsze, jednak uzyskane wyniki zapisywane sa w oddzielnych tablicach. Po wykonaniu współbieżnej części funkcji prywatne wektory zostają scalone w jeden końcowy wektor w odpowiedniej kolejności, dzięki czemu wyeliminowany został problem wyścigu a także false sharingu: usunięto współdzielony wektor wyników, modyfikowany równocześnie przez wątki, stąd brak problemu unieważnień linii pamięci. Uzyskane wyniki są poprawne.

2.2.5 *PAR_WYK_DOM* - Wykreślanie, wersja równoległa domenowa

Podejście domenowe do algorytmu Sita Eratostenesa charakteryzuje się przydziałem do procesu fragmentu tablicy wykreśleń przy przekazaniu mu całej tablicy liczb pierwszych we wskazanym zakresie.

Wersja naiwna

Wersja naiwna równoległego algorytu sita Eratostenesa w podejściu domenowym wykorzystuje u nas pobranie wektora liczb pierwszych z użyciem równoległej funkcji dzielenia w wersji dobrej (*PAR_DZIEL_OK*), przy czym wektor ograniczony jest górną granicą zakresu będącą pierwiastkiem kwadratowym z górnej granicy właściwego zakresu. Następnie w bloku współdzielonym kolejne wielokrotności każdej znalezionej liczby pierwszej oznaczane są w osobnej strukturze jako liczby nie-pierwsze. Następnie, po zakończeniu bloku równoległegło tworzony jest wektor do przechowania ostatecznych liczb pierwszych i w pętli sprawdzającej liczby z całego zakresu dodawane są te liczby, które nie zostały w bloku równoległym oznaczone jako nie-pierwsze. W takiej wersji kodu wyniki nie wychodzą poprawne, oraz występuje ryzyko zjawisko false sharingu - wszystkie wątki operują na tej samej tablicy, w której zaznaczane są wykreślenia, a zakresy działania wątków na tej tablicy się przeplatają, zatem w przypadku znalezienia się elementów tablicy na wspólnych liniach pamięci wystąpi problem false sharingu.

```
namespace NAIVE {
        vector<int> PARALLEL_primesSieveDomain(int a, int b) {
                int upperBound = int(sqrt(b));
                vector<int> firstPrimes = BEST::PARALLEL_primesDivide(2, upperBound, false);
                CHECK* checked = new CHECK[b + 1]{ false };
#pragma omp parallel
                {
#pragma omp for
                        for (int i = 0; i < firstPrimes.size(); i++) {
                                int multiplicity = firstPrimes[i] * 2;
                                while (multiplicity <= b) {
                                        checked[multiplicity] = true;
                                        multiplicity += firstPrimes[i];
                                }
                        }
                }
                vector<int> primes;
                primes.reserve(b - a + 1);
                for (int i = a; i <= b; i++) {
                        if (!checked[i]) {
                                primes.push_back(i);
                        }
                }
                delete[] checked;
                return primes;
        }
}
```

Figure 6: *PAR_WYK_DOM_NAIVE* - kod wersji naiwnej równoległego algorytmu wykreślania domenowo

Wersja poprawiona

Wersja poprawiona równoległego algorytmu wykreślania w podejściu domenowym tak samo jak naiwna pobiera wektor liczb pierwszych z użyciem (*PAR_DZIEL_OK*), jednak każdy wątek otrzymuje osobną tablicę do przechowywania w niej swoich wyników. W regionie równoległym procesy zapełniają swoją tablicę wyników informacjami o kolejnych odnalezionych wielokrotnościach (czyli liczbach, które nie są pierwsze), a po zakończeniu poza regionem równoległym ich wyniki wykorzystane są do zebrania wszystkich nieoznaczonych liczb z podanego zakresu. Dzięki temu uniknięto problemu wyścigu - procesy operują na osobnych tablicach - oraz false sharingu - nie występuje groźba modyfikacji często i cyklicznie tych samych linii pamięci podręcznych procesorów.

```
namespace BETTER {
        vector<int> PARALLEL_primesSieveDomain(int a, int b) {
                int upperBound = int(sqrt(b));
                vector<int> firstPrimes = BEST::PARALLEL_primesDivide(2, upperBound, false);
                int threadsCount = omp_get_max_threads();
                CHECK** checked = new CHECK * [threadsCount];
                for (int i = 0; i < threadsCount; i++) {
                        checked[i] = new CHECK[b + 1]{ false };
                }
#pragma omp parallel
                {
#pragma omp for
                        for (int i = 0; i < firstPrimes.size(); i++) {
                                int multiplicity = firstPrimes[i] * 2;

                                while (multiplicity <= b) {
                                        checked[omp_get_thread_num()][multiplicity] = true;
                                        multiplicity += firstPrimes[i];
                                }
                        }
                }
                vector<int> primes;
                primes.reserve(b - a + 1);
                for (int i = a; i <= b; i++) {
                        int sum = false;
                        for (int j = 0; j < threadsCount; j++) {
                                sum |= checked[j][i];
                        }
                        if (!sum) {
                                primes.push_back(i);
                        }
                }
                delete[] checked;
                return primes;
        }
}
```

Figure 7: *PAR_WYK_DOM_BETTER* - kod poprawionej równoległego algorytmu wykreślania domenowo

Wersja dobra

Wersja ostateczna została ulepszona w stosunku do wersji poprawionej o dyrektywę *pragma omp for schedule(dynamic)*, dzięki czemu każdemu wątkowi przypisywany jest zakres iteracji, domyślnie o wielkości przydziału równej jeden. Takie rozwiązanie zwiększyło szybkość działania programu w stosunku do wersji poprawionej przy mierzeniu czasu dokonywanemu wstępnie przy użyciu mechanizmów *Time*. Tak jak w wersji poprawionej, zminimalizowano tu ryzyko wystąpienia false sharingu (procesy operują na odrębnych tablicach) i uniknięto błędów prowadzących do wystąpienia wyścigu (wspólny wektor wyniku modyfikowany po zakończeniu części równoległej).

```
namespace BEST {
	vector<int> PARALLEL_primesSieveDomain(int a, int b) {
		int upperBound = int(sqrt(b));
		vector<int> firstPrimes = BEST::PARALLEL_primesDivide(2, upperBound, false);
		int threadsCount = omp_get_max_threads();
		CHECK** checked = new CHECK * [threadsCount];
		for (int i = 0; i < threadsCount; i++) {
			checked[i] = new CHECK[b + 1]{ false };
		}

#pragma omp parallel
		{
#pragma omp for schedule(dynamic)
			for (int i = 0; i < firstPrimes.size(); i++) {
				int multiplicity = firstPrimes[i] * 2;

				while (multiplicity <= b) {
					checked[omp_get_thread_num()][multiplicity] = true;
					multiplicity += firstPrimes[i];
				}
			}
		}
		vector<int> primes;
		primes.reserve(b - a + 1);
		for (int i = a; i <= b; i++) {
			int sum = false;
			for (int j = 0; j < threadsCount; j++) {
				sum |= checked[j][i];
			}
			if (!sum) {
				primes.push_back(i);
			}
		}
		delete[] checked;
		return primes;
	}
}
```

Figure 8: *PAR_WYK_DOM_OK* - kod ostatecznej wersji równoległego algorytmu wykreślania domenowo

2.2.6 *PAR_WYK_FUNC* - Wykreślanie, wersja równoległa funkcyjna

Wersja funkcyjna algorytmu Sita Eratostenesa polega na przydziale do procesu całej tablicy wykreśleń przy przekazaniu mu fragmentu zbioru liczb pierwszych, których wielokrotności są usuwane.

Wersja naiwna

Wersja równoległa algorytmu wykreślania z podejściem funkcyjnym podobnie jak wersja domenowa wykorzystuje najlepszą uzyskaną wersję równoległą algorytmu dzielenia (*PAR_DZIEL_OK*) do pobrania początkowej tablicy liczb pierwszych z zakresu ograniczonego do pierwiastka kwadratowego górnej granicy przedziału zadanego. Na początku założona zostaje struktura do przechowywania zmiennych typu boolean, a następnie w regionie równoległym każdy wątek otrzymuje osobny zakres tablicy liczb pierwszych do sprawdzenia (funkcje *getStart(), getEnd()* odpowiadają za przydział granic zakresu) i wyłącznie dla tego zakresu w pętli wykreśla (oznacza jako *false*) kolejne wielokrotności liczb pierwszych. Po zakończeniu regionu równoległego wykonywana jest pętla po wszystkich liczbach z zakresu i te, które nie zostały wykreślone, dodawane są do wynikowego wektora liczb pierwszych. Wyniki uzyskane przez program są poprawne, brak groźby wystąpienia wyścigu (brak wspólnego modyfikowania tej samej tablicy przez wątki) jednak

wątki operują na tej samej tablicy, w której zaznaczone są wykreślenia. Zakresy działania wątków na tej tablicy się przeplatają, zatem jest ryzyko wystąpienia false sharingu.

```
namespace NAIVE {
        vector<int> PARALLEL_primesSieveFunctional(int a, int b) {
                int upperBound = int(sqrt(b));
                vector<int> firstPrimes = BEST::PARALLEL_primesDivide(2, upperBound, false);
                int threadsCount = omp_get_max_threads();
                CHECK* checked = new CHECK[b + 1]{ false };
#pragma omp parallel
                {
                        int privateA = getStart(a, b, omp_get_thread_num(), threadsCount);
                        int privateB = getEnd(a, b, omp_get_thread_num(), threadsCount);

                        for (int i = 0; i < firstPrimes.size(); i++) {
                                int multiplicity = firstPrimes[i] * 2;
                                while (multiplicity <= privateB) {
                                        checked[multiplicity] = true;
                                        multiplicity += firstPrimes[i];
                                }
                        }
                }
                vector<int> primes;
                for (int i = a; i <= b; i++) {
                        if (!checked[i]) {
                                primes.push_back(i);
                        }
                }
                delete[] checked;
                return primes;
                return vector<int>();
        }
}
```

Figure 9: $PAR\_WYK\_FUNC\_NAIVE$ - kod wersji naiwnej równoległego algorytmu wykreślania funkcyjnie

Wersja dobra

Wersja ostateczna algorytmu wykreślania z podejściem funkcyjnym również wykorzystuje wersję równoległą algorytmu dzielenia (PAR_DZIEL_OK) do uzyskania tablicy liczb pierwszych z ograniczonego zakresu. Na początku zadeklarowana zostaje struktura do przechowywania zmiennych typu boolean, a następnie w części równoległej każdy wątek sprawdza osobny zakres tablicy liczb pierwszych do wykreślania. Zmianą w stosunku do wersji wcześniejszej jest ograniczenie zakresu przeszukiwania: wcześniej każdy wątek sprawdzał od początku całego przedziału do momentu, który został wyliczony jako koniec jego fragmentu, co okazało się mieć znaczący wpływ na dokonane z wykorzystaniem *Time* pomiary czasu: ograniczenie zakresu również "od dołu" znacznie przyspieszyło uzyskiwane pomiary co dało nam podstawy przypuszczać, że podobne efekty osiągnie algorytm przy użyciu programu pomiarowego VTune.

```
namespace BEST {
        vector<int> PARALLEL_primesSieveFunctional(int a, int b) {
                int upperBound = int(sqrt(b));
                vector<int> firstPrimes = BEST::PARALLEL_primesDivide(2, upperBound, false);

                int threadsCount = omp_get_max_threads();
                CHECK* checked = new CHECK[b + 1]{ false };
#pragma omp parallel
                {
                        int privateA = getStart(a, b, omp_get_thread_num(), threadsCount);
                        int privateB = getEnd(a, b, omp_get_thread_num(), threadsCount);

                        for (int i = 0; i < firstPrimes.size(); i++) {
                                int multiplicity = firstPrimes[i] * 2;
                                int expr = ((privateA - firstPrimes[i] * 2) / firstPrimes[i]) * firstPrimes
                                if (expr > 0) {
                                        multiplicity += expr;
                                }

                                while (multiplicity <= privateB) {
                                        checked[multiplicity] = true;
                                        multiplicity += firstPrimes[i];
                                }
                        }
                }
                vector<int> primes;
                for (int i = a; i <= b; i++) {
                        if (!checked[i]) {
                                primes.push_back(i);
                        }
                }
                delete[] checked;
                return primes;
        }
}
```

Figure 10: $PAR\_WYK\_FUNC\_OK$ - kod ostatecznej wersji równoległego algorytmu wykreślania funkcyjnie

2.3 Zagadnienie podziału pracy w programach

Dyrektywa *schedule* służy do definiowania sposobu rozdziału pracy na wątki w pętli *for*. Możliwe rodzaje podziału pracy:

- static– podział dokonany przed uruchomieniem pętli, zatem najmniejszy narzut czasu wykonania
- dynamic– wątki wykonują kolejno „pierwszy wolny" segment wewnątrz instrukcji for, jednak powoduje to opóźnienia z powodu konieczności ubiegania się o przydział
- guided– podobnie jak dynamic, wielkość segmentu może ulegać zmniejszaniu
- runtime– podział zależy od wartości zmiennej środowiskowej OMP_SCHEDULE

Praca dzielona jest zawsze na tyle części, ile jest wątków: nierówny może być tylko przydział pracy do konkretnych wątków. W kodach programów większości mamy x iteracji, które dzielimy na n wątków więc każdy dostaje po x/n zadań; wyjątek stanowi kod PAR_WYK_DOM_OK, gdzie wprowadziliśmy konstrukcję *schedule*.

2.4 Tabela podsumowująca problemy poprawnościowe i efektywnościowe

Program	Poprawny	Dlaczego	False Sharing	Dlaczego
SEQ_1 : Dzielenie	TAK	Program jednowątkowy	NIE	Program jednowątkowy
SEQ_2: Wykreślanie	TAK	Program jednowątkowy	NIE	Program jednowątkowy
PAR_DZIEL_NAIVE (NAIVE::PARALLEL_primes Divide)	NIE	Równoczesne dodawanie nowych elementów do wektora przez wątki bez synchronizacji	TAK	Współdzielony wektor wyników, modyfikowany równocześnie przez wątki, również elementy znajdujące się w tej samej lini pamięci
PAR_DZIEL_OK (BEST::PARALLEL_primes Divide)	TAK	Dopiero po wykonaniu współbieżnej części funkcji prywatne wektory wątków zostają scalone w jeden	NIE	Każdy wątek ma prywatny wektor wyników, który modyfikuje
PAR_WYK_FUNC_NAIVE (NAIVE::PARALLEL_primes SieveFunctional)	TAK	Scalanie wektora wyników po zakończeniu bloku równoległego	TAK	Wszystkie wątki operują na tej samej tablicy, w której zaznaczone są wykreślenia. Zakresy działania wątków na tej tablicy się przeplatają, zatem jest możliwość zajścia false sharingu.
PAR_WYK_FUNC_OK (BEST::PARALLEL_primes SieveFunctional)	TAK	Scalanie wektora wyników po zakończeniu bloku równoległego	NIE	Wątki operują na wyznaczonym zakresie tablicy wykreśleń; zapewniono nie przeplatanie się zakresów
PAR_WYK_DOM_NAIVE(NAIVE::PARALLEL_primesSieveDomain)	TAK	Scalanie wektora wyników po zakończeniu bloku równoległego	TAK	Wszystkie wątki operują na tej samej tablicy, w której zaznaczane są wykreślenia; częste korzystanie wątków ze wspólnych linii adresowych.
PAR_WYK_DOM_BETTER (BETTER::PARALLEL_primesSieveDomain)	TAK	Scalanie wektora wyników po zakończeniu bloku równoległego	NIE	Każdy wątek operuje na osobnej tablicy, które dopiero po zakończeniu równoległości są scalane: brak operowania na wspólnej linii adresowej.
PAR_WYK_DOM_OK (BEST::PARALLEL_primes SieveDomain)	TAK	Scalanie wektora wyników po zakończeniu bloku równoległego	NIE	Każdy wątek operuje na osobnej tablicy, które dopiero po zakończeniu równoległości są scalane: brak operowania na wspólnej linii adresowej.

Figure 11: Zestawienie problemów poprawnościowych i efektywnościowych w poszczególnych programach

3 Prezentaja wyników i omówienie przebiegu eksperymentu obliczeniowo-pomiarowego

Eksperyment wykonany został na urządzeniu i przy użyciu programów opisanyych w pierwszym punkcie tego sprawozdania. Dla każdego testowanego programu test przeprowadzany był dla trzech rodzajów instancji wyznaczających różne zakresy poszukiwania liczb pierwszych. Testowane instancje:
a) 2...MAX
b) MAX/2...MAX
c) 2...MAX/2
Jako MAX w programie przyjęliśmy 10^8.

Wszystkie instancje testowane były kolejno dla następujących ilości wątków:
a) Przetwarzanie sekwencyjne – jeden procesor
b) Przetwarzanie równoległe - liczba procesorów fizycznych: dwa procesory
c) Przetwarzanie równoległe - liczba procesorów logicznych: cztery procesory
Liczba równa połowie procesorów fizycznych to w przypadku użytego urządzenia liczba procesorów logicznych.

W dalszej części przedstawione zostaną wyniki eksperymentu zawierające zestawienie pomiarów dla stworzonych przez nas wersji programu. Do sprawozdania dołączony został arkusz kalkulacyjny zawierający zapis pomiarów dla wszystkich utworzonych kodów.

3.1 Intel Parallel Studio XE - omówienie narzędzia

VTune to narzędzie do analizy tworzonego oprogramowania pozwalające na ocenę jakości kodu pod kątem pracy procesora. Umożliwia dokładne przejrzenie czasu pracy konkretnych wątków, pomaga w odnalezieniu wąskich gardeł i zbiera informacje o wydajności przetwarzania. VTune pomaga w różnego rodzaju profilowaniu kodu, w tym próbkowaniu stosu, profilowaniu wątków i próbkowaniu zdarzeń sprzętowych. Wynik profilera składa się ze szczegółów, takich jak czas spędzony w każdej podprogramie, który można prześledzić do poziomu instrukcji. Czas potrzebny na instrukcje wskazuje na możliwość istnienia wąskiego gardła w danym miejscu kodu. Narzędzie może być również wykorzystane do analizy wydajności wątków. W projekcie korzystano przede wszystkim z danych zebranych przy użyciu Microarchitecture Exploration.

3.2 Wyniki uzyskane dla wersji kodu sekwencyjnych

Poniżej przedstawione zostały wyniki uzyskane w Vtune dla wersji sekwencyjnych. Kod przetestowano dla wszystkich doborów zakresów przedziałów poszukiwań liczb pierwszych w celu zapoznania się ze szczegółami faktycznej pracy procesora dla programów nie działających w trybie równoległym o zróżnicowanej złożoności jako baza pod dalszą analizę programów równoległych.
Dla wersji sekwencyjnej algorytmu dzielącego pozwoliliśmy sobie na przesunięcie zaleconej górnej granicy czasu przetwarzania z uwagi na ogólny stan wykorzystywanego urządzenia.

	SEQ_1			SEQ_2		
	2..MAX	HALF..MAX	2..HALF	2..MAX	HALF..MAX	2..HALF
Elapsed time (Microarchitecture Exp.)	158,2s	93,513s	60,5s	0,047s	0,018s	0,016s
Instructions retired	3,78738E+11	2,36798E+11	1,40953E+11	2,50E+07	2,00E+07	2,25E+07
Clockticks	4,38E+11	2,64153E+11	1,6465E+11	4,00E+07	4,00E+07	3,00E+07
Retiring	51,40%	53,00%	54,10%	0%	0%	0%
Front-end bound	50,40%	5,00%	50,40%	72,50%	0%	0%
Back-end bound	0%	0%	0%	27,50%	100%	100%
Memory bound	0%	0%	0%	0%	0%	0%
Core bound	0%	0%	0%	27,50%	100%	100%
Effective physical core utilization (bottom-up)	36,90%	37,80%	35,30%	6,40%	18,80%	15,40%

Figure 12: Tabela przedstawiająca wyniki wersji sekwencyjnych

Na wykresie poniżej przedstawiono zestawienie procentowego czasu obliczeń algorytmów sekwencyjnych dla każdej instancji. Wykres pokazuje, że algorytm SEQ_2, choć wielokrotnie szybszy, procentowo dłużej przetwarzał największą instancję 2...MAX, niż algorytm SEQ_1. Może to być spowodowane wykorzystaniem przez SEQ_2 wyszukiwaniem liczb pierwszych dla fragmentu zakresu przy użyciu SEQ_1 aby uzyskać wstępną tablicę liczb pierwszych: dla większych zakresów liczb wydajność algorytmu dzielenia mocno się obniża.

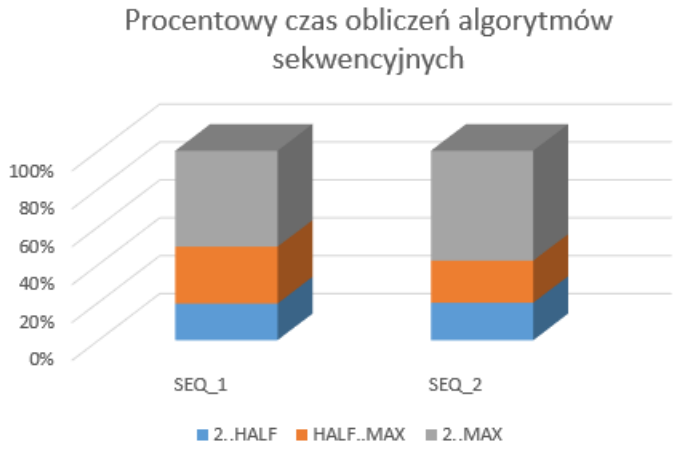

Figure 13: Procentowe zestawienie przetwarzania konkretnych instancji przy użyciu algorytmów sekwencyjnych

3.3 Wyniki uzyskane dla wersji kodów równoległych

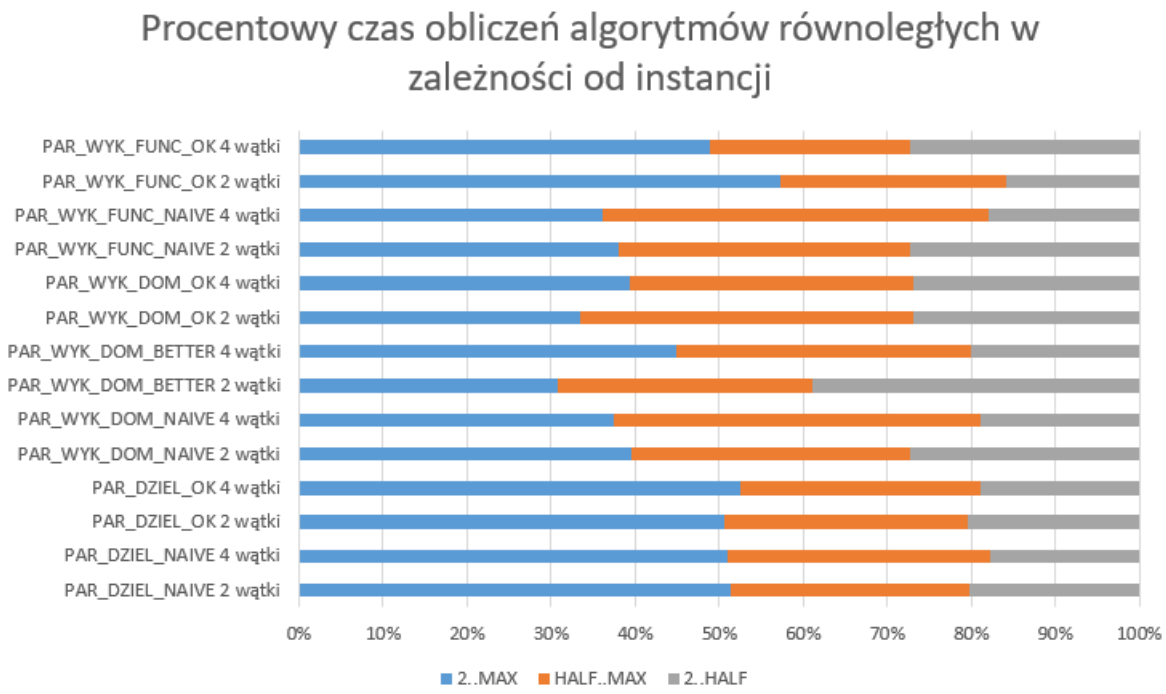

Figure 14: Procentowe zestawienie czasów przetwarzania konkretnych instancji przy użyciu algorytmów równoległych

Kolejny wykres przedstawia zestawienie procentowego czasu obliczeń algorytmów równoległych dla kolejnych wielkości instancji. Niemal wszystkie algorytmy najdłużej, bo około 50% czasu przetwarzały największą instancję 2..MAX. Wyraźnie wyróżnia się tu algorytm PAR_WYK_DOM_BETTER (równoległe wykreślanie w wersji domenowej, wersja poprawiona) przy użyciu 2 wątków: wynik około 30%. Z kolei dziwi stosunkowo długi czas przetwarzania najkrótszej i

najłatwiejszej instancji 2..HALF tego algorytmu. W dalszej części analizy przedstawione zostaną dokładniejsze wyniki odczytane z programu pomiarowego VTune, które prawdopodobnie pozwolą lepiej zrozumieć takie wyniki algorytmu.

3.3.1 Tabela zbiorcza wszystkich zebranych parametrów

			Elapsed time [	Instructions retire	Clockticks	Retiring	Front-end bour	Back-end bo	Memory bou	Core bound	Effective phy	Speed [ilość/s]
SEQ_1	1 wątek	2..MAX	158,2	3,78738E+11	4,384E+11	51,40%	50,40%	0%	0%	0%	36,90%	632111,2389
		HALF..MAX	93,513	2,36798E+11	2,642E+11	53,00%	5,00%	0%	0%	0%	37,80%	534685,0171
		2..HALF	60,5	1,40953E+11	1,647E+11	54,10%	50,40%	0%	0%	0%	35,30%	826446,2479
SEQ_2	1 wątek	2..MAX	0,047	2,50000E+07	4,0000E+07	0%	72,50%	27,50%	0%	27,50%	6,40%	2127659532
		HALF..MAX	0,018	2,00000E+07	4,0000E+07	0%	0%	100%	0%	100%	18,80%	2777777778
		2..HALF	0,016	2,25000E+07	3,0000E+07	0%	0%	100%	0%	100%	15,40%	3124999875
Równoległy dzielenie - naiwne	4 Wątki	2..MAX	75,664	3,78835E+11	5,506E+11	62,20%	42,10%	0,00%	0,00%	0,00%	64,30%	13216632,454
		HALF..MAX	46,676	2,38083E+11	3,588E+11	66,00%	42,80%	0,00%	0,00%	0,00%	60,30%	1071214,329
		2..HALF	26,3	1,41175E+11	2,051E+11	63,30%	40,40%	0,00%	0,00%	0,00%	68,70%	1901140,608
	2 Wątki	2..MAX	113,549	3,7786E+11	4,872E+11	57,20%	49,40%	0,00%	0,00%	0,00%	46,10%	8806677,0469
		HALF..MAX	62,667	2,37308E+11	3,124E+11	57,60%	48,10%	0,00%	0,00%	0,00%	52,50%	797868,0964
		2..HALF	44,516	1,40843E+11	1,852E+11	60,80%	50,70%	0,00%	0,00%	0,00%	42,10%	1123191,616
Równoległy dzielenie - dobry	4 Wątki	2..MAX	81,106	3,79548E+11	5,554E+11	62,90%	42,90%	0,00%	0,00%	0,00%	59,30%	1232954,381
		HALF..MAX	44,256	2,37703E+11	3,59E+11	64,40%	40,60%	0,00%	0,00%	0,00%	66,30%	1129790,311
		2..HALF	29,352	1,41295E+11	2,08E+11	63,50%	40,60%	0,00%	0,00%	0,00%	61,90%	1703461,365
	2 Wątki	2..MAX	108,379	3,77878E+11	4,793E+11	55,80%	49,40%	0,00%	0,00%	0,00%	49,30%	922687,9562
		HALF..MAX	61,743	2,3747E+11	3,12E+11	60,00%	51,30%	0,00%	0,00%	0,00%	51,50%	809808,3993
		2..HALF	43,739	1,40998E+11	1,865E+11	59,80%	49,70%	0,00%	0,00%	0,00%	43,60%	1143144,516
Równoległy wykreślanie domenowe - naiwne	4 Wątki	2..MAX	1,22	3,38000E+09	5,4375E+09	27,70%	14,80%	52,40%	35,70%	16,70%	37,10%	81967211,48
		HALF..MAX	1,425	2,55500E+09	4,3325E+09	31,90%	11,70%	55,80%	31,80%	24%	26,10%	35087719,3
		2..HALF	0,613	2,03750E+09	3,3875E+09	34,20%	21,40%	42,60%	30,80%	11,90%	36,60%	81566065,25
	2 Wątki	2..MAX	1,373	3,00250E+09	4,5450E+09	34,90%	13,10%	40,40%	29,60%	10,80%	32,30%	72833210,49
		HALF..MAX	1,146	2,58750E+09	4,0225E+09	26,90%	6,50%	65,20%	45,90%	19,30%	41,00%	43630017,45
		2..HALF	0,949	1,45500E+09	2,2075E+09	23,60%	10,50%	65,80%	31,90%	33,90%	17,80%	52687036,88
Równoległy wykreślanie domenowe - better	4 Wątki	2..MAX	2,992	1,24900E+10	8,8400E+09	59,20%	20,10%	19,20%	11,00%	8,20%	21,70%	33422459,22
		HALF..MAX	2,345	1,10850E+10	9,0775E+09	50,20%	21,20%	19,90%	11,30%	8,60%	29,60%	21321961,62
		2..HALF	1,34	6,61250E+09	5,0275E+09	53,60%	16,20%	21,60%	11,40%	10,20%	24,00%	37313431,34
	2 Wątki	2..MAX	2,132	1,09925E+10	7,0675E+09	55,30%	23,40%	18,80%	7,60%	11,20%	32,20%	46904314,26
		HALF..MAX	2,09	9,84250E+09	6,6950E+09	56,60%	23,00%	13,60%	6,60%	7,10%	30,60%	23923444,98
		2..HALF	2,686	1,10025E+10	7,7725E+09	47,30%	25,20%	16,80%	8,60%	8,20%	25,40%	18615040,21
Równoległy wykreślanie domenowy - dobry	4 Wątki	2..MAX	2,015	1,23850E+10	1,1653E+10	51,30%	18,50%	23,20%	12,30%	10,90%	38%	49627790,57
		HALF..MAX	1,736	1,07000E+10	1,0868E+10	48,80%	15,20%	32,70%	20,80%	11,90%	39,60%	28801843,32
		2..HALF	1,38	6,18250E+09	5,5725E+09	46,80%	15,60%	31,30%	17,70%	13,60%	26,10%	36231882,61
	2 Wątki	2..MAX	1,952	1,09400E+10	8,8550E+09	53,30%	12,10%	29,40%	6,70%	12,80%	33,50%	51229507,17
		HALF..MAX	2,319	9,59250E+09	9,0225E+09	58,00%	19,90%	12,50%	5,80%	5,80%	25,80%	21561017,68
		2..HALF	1,578	5,38250E+09	4,8675E+09	56,20%	20,00%	8,80%	4,80%	4,00%	19,90%	31685676,81
Równoległy wykreślanie funkcyjne - naiwny	4 Wątki	2..MAX	2,476	5,12000E+09	1,427E+10	13,50%	6,20%	79%	69,40%	9,50%	50,50%	40387721,32
		HALF..MAX	3,149	5,80250E+09	2,037E+10	12,20%	6,70%	80,90%	73,20%	7,80%	52,00%	15878056,53
		2..HALF	1,236	2,56000E+09	6,58E+10	19,80%	7,90%	70,90%	62,80%	8,10%	42%	40453072,82
	2 Wątki	2..MAX	2,139	3,30250E+09	6,9650E+09	28,40%	13,20%	56,50%	45,90%	10,50%	28,70%	46750817,2
		HALF..MAX	1,956	3,41500E+09	8,5200E+09	18,80%	8%	69,80%	58,80%	10,90%	37,10%	25562372,19
		2..HALF	1,531	1,68750E+09	3,6250E+09	12,80%	13,60%	72,80%	49,50%	23,30%	19,10%	32658391,9
Równoległy wykreślanie funkcyjne - dobry	4 Wątki	2..MAX	1,298	2,69000E+09	7,0025E+09	24,00%	10,80%	67%	51,60%	14,80%	43,90%	77041600,92
		HALF..MAX	0,629	1,66500E+09	4,4675E+09	29,20%	18,20%	53,90%	10,40%	12,50%	54,10%	79491255,96
		2..HALF	0,724	1,28250E+09	3,3500E+09	16,40%	10,40%	75,80%	57,50%	18,30%	37%	69060770,72
	2 Wątki	2..MAX	2,042	2,04000E+09	3,9750E+09	30,60%	27,70%	38,00%	30,30%	7,70%	15,50%	48971595,49
		HALF..MAX	0,958	1,38000E+09	3,0100E+09	16,40%	24%	57,60%	45,70%	11,90%	24,90%	52192066,81
		2..HALF	0,568	1,37750E+09	2,9050E+09	12,00%	5,00%	82,00%	76,00%	6,10%	39,00%	88028165,49

Figure 15: Zbiór parametrów wszystkich utworzonych kodów dla różnych ilości wątków i wielkości instancji

3.4 Analiza i omówienie zebranych wyników

W osobnej tabeli zdecydowaliśmy się dla czytelności umieścić miary przyspieszenia (*acceleration* oraz efektywności przetwarzania równoległego *efficiency*.

- Acceleration: przyspieszenie przetwarzania równoległegodla badanego wariantu kodu równolegego – parametr będący ilorazem czasu przetwarzania najlepszego dostępnego przetwarzania sekwencyjnego (dowolnym algorytmem w tym samym systemie) oraz czasu przetwarzania równoległego, dla którego przyspieszenie jest wyznaczane.
- Efektywność przetwarzania równoległego - iloraz przyspieszenia przetwarzania równoległego i liczby użytych w przetwarzaniu procesorów fizycznych.

		2..MAX			HALF..MAX			2..HALF	
	2..MAX	Acceleration	Efficiency	HALF..MAX	Acceleration	Efficiency	2..HALF	Acceleration	Efficiency
PAR_DZIEL_NAIVE 2 wątki	113,549	0,01074426	0,00537213	62,667	0,010037181	0,00501859	44,516	0,012759457	0,006379729
PAR_DZIEL_NAIVE 4 wątki	75,664	0,016123916	0,004030979	46,67	0,013477609	0,003369402	26,3	0,021596958	0,00539924
PAR_DZIEL_OK 2 wątki	108,379	0,011256793	0,005628397	61,743	0,01018739	0,005093695	43,739	0,012986122	0,006493061
PAR_DZIEL_OK 4 wątki	81,106	0,015042044	0,003760511	44,256	0,014212762	0,003553191	29,352	0,019351322	0,00483783
PAR_WYK_DOM_NAIVE 2 wątki	1,373	0,888565186	0,444282593	1,146	0,54886562	0,27443281	0,949	0,598524763	0,299262381
PAR_WYK_DOM_NAIVE 4 wątki	1,22	1	0,25	1,425	0,441403509	0,110350877	0,613	0,926590538	0,231647635
PAR_WYK_DOM_BETTER 2 wątki	2,132	0,572232645	0,286116323	2,09	0,300956938	0,150478469	2,686	0,211466865	0,105733433
PAR_WYK_DOM_BETTER 4 wątki	2,992	0,407754011	0,101938503	2,345	0,268230277	0,067057569	1,34	0,423880597	0,105970149
PAR_WYK_DOM_OK 2 wątki	1,952	0,625	0,3125	2,319	0,271237602	0,135618801	1,578	0,359949303	0,179974651
PAR_WYK_DOM_OK 4 wątki	2,015	0,605459057	0,151364764	1,736	0,362327189	0,090581797	1,38	0,411594203	0,102898551
PAR_WYK_FUNC_NAIVE 2 wątki	2,139	0,570359981	0,285179991	1,956	0,321574642	0,160787321	1,531	0,370999347	0,185499673
PAR_WYK_FUNC_NAIVE 4 wątki	2,476	0,49273021	0,123182553	3,149	0,199745951	0,049936488	1,236	0,459546926	0,114886731
PAR_WYK_FUNC_OK 2 wątki	2,042	0,597453477	0,298726738	0,958	0,6565762	0,3282881	0,568	1	0,5
PAR_WYK_FUNC_OK 4 wątki	1,298	0,93990755	0,234976888	0,629	1	0,25	0,724	0,784530387	0,196132597

Figure 16: Miary przyspieszenia i efektywności algorytmów równoległych z podziałem na wielkość instancji

Zdecydowaliśmy się na użycie formatowania warunkowego dla zwiększenia czytelności uzyskanych wyników: kolor czerwony oznacza algorytmy najsłabsze, o najdłuższym czasie przetwarzania, natomiast zielony algorytmy najszybsze. Wartość przyspieszenia i efektywności oznaczają jednobarwne paski - im większe pole powierzchni paska, tym lepszy wynik uzyskany przez algorytm dla danej miary.
Z zestawienia odczytać można, że zdecydowanie najlepsze wyniki czasowe osiągnęły dwa algorytmy: PAR_WYK_DOM_NAIVE (równoległe wykreślane domenowe w wersji naiwnej) dla obu sprawdzonych ilości wątków oraz algorytm PAR_WYK_FUNC_OK (równoległe wykreślanie funkcyjne w wersji dobrej) dla ilości wątków 4.

W przypadku pozostałych parametrów, dla zwiększenia czytelności i ułatwienia zaobserwowania zależności wydajnościowych postanowiliśmy przedstawić część z parametrów w postaci graficznej.
W szczególności postanowiliśmy przyjrzeć się procentowemu udziałowi wykorzystanych zasobów procesora do przetwarzania kodu. Z wykresu zaobserwować można wykorzystanie zasobów powyżej 50% dla ponad połowy badanych algorytmów. Z zasobów w znikomym stopniu korzysta algorytm sekwencyjny wykreślania: jego czas przetwarzania również jest najkrótszy.
Małym wykorzystaniem zasobów procesora charakteryzują się też algorytmy, które w poprzednim zestawieniu (przyspieszenie i efektywność) uzyskały dobre rezultaty: PAR_WYK_DOM_NAIVE oraz PAR_WYK_FUNC w wersjach dobrej oraz naiwnej. Zwraca to uwagę na związek między niewielkim wykorzystaniem zasobów procesora a zwiększeniem prędkości przetwarzania.

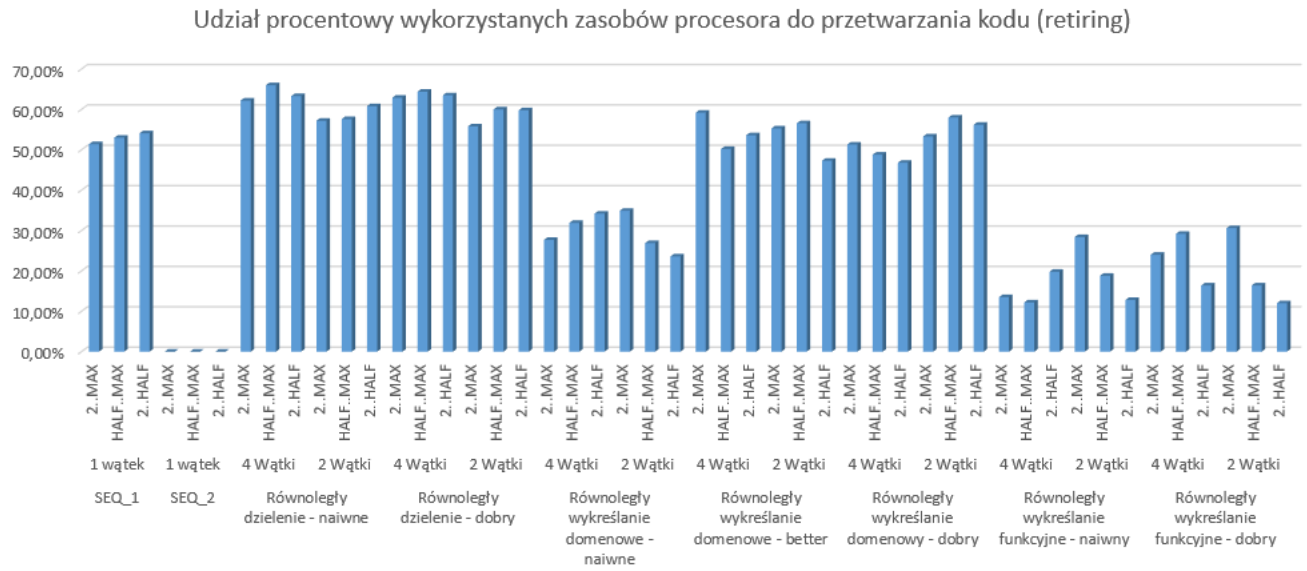

Figure 17: Wykres przedstawiający udział procentowy wykorzystanych zasobów procesora do przetwarzania kodu (retiring)

Postanowiliśmy też przyjrzeć się stosunkowi ilości instrukcji assemblera do liczby cykli procesora. Im większa jest to wartość, tym więcej wykonanych instrukcji na cykl, a więc wykorzystanie procesora powinno być lepsze. Największe wartości osiągnęły te algorytmy, których prędkości i przyspieszenia były w grupie średniej, natomiast wykorzystanie zasobów procesora wysokie (około 50%). Może to wskazywać, że algorytmy z tej grupy (PAR_WYK_DOM_BETTER, PAR_WYK_DOM_GOOD) wykonują wiele operacji obciążających zbyt mocno procesor, przez co ich prędkości nie są satysfakcjonujące.
Zauważyć można również iż algorytm PAL_WYK_DOM_BETTER nie ma zbyt dużych różnic w osiąganych wynikach dla odmiennych instancji (różnice w obrębie 10% między instancjami). Koreluje to z wcześniejszym wykresem Figure 14, gdzie osiągnął dobry udział procentowy czasu przetwarzania największej instancji, jednak dla kolejnych instancji ten osiągany czas nie uległ zmianom. Z kolei w przypadku algorytmu sekwencyjnego wykreślania SEQ_2 wykorzystanie zasobów procesora jest niskie, a ilość wykonanych instrukcji na cykl wysoka. Program mógł wykonywać wiele operacji nie obciążających za mocno procesora, a więc nie tracąc na najlepszej ze wszystkich algorytmów osiągniętej prędkości wykonania.

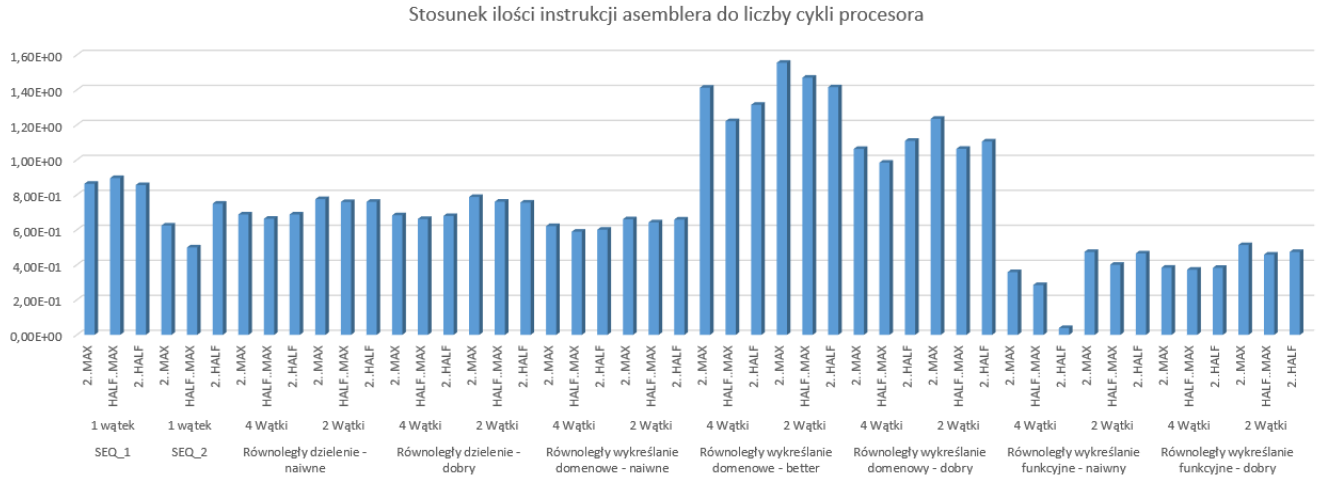

Figure 18: Wykres przedstawiający Stosunek ilości instrukcji asemblera do liczby cykli procesora (Instructions retired/Clockticks)

Z powyższych wyników zdecydowaliśmy się wybrać wyróżnione algorytmy dla analizy udziału procesorów:

- SEQ_2 - sekwencyjna wersja algorytmu wykreślania
- PAR_DZIEL_NAIVE - równoległa wersja algorytmu dzielenia, naiwna,2 wątki
- PAR_WYK_DOM_BETTER - równoległy algorytm wykreślania domenowo, wersja poprawiona, 2 wątki
- PAR_WYK_DOM_NAIVE - równoległy algorytm wykreślania domenowo, wersja naiwna, 4 wątki
- PAR_WYK_FUNC_OK - równoległy algorytm wykreślania funkcyjnie, wersja dobra, 4 wątki

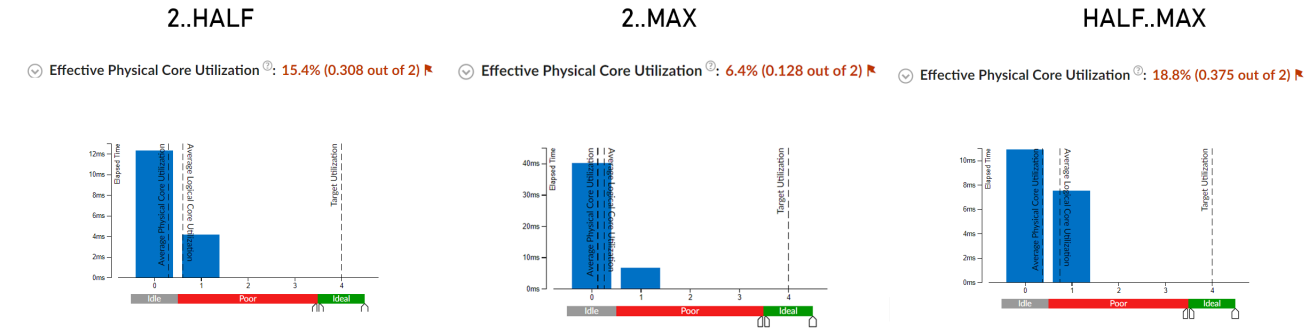

Figure 19: Effective Physical Core Utilization - SEQ_2

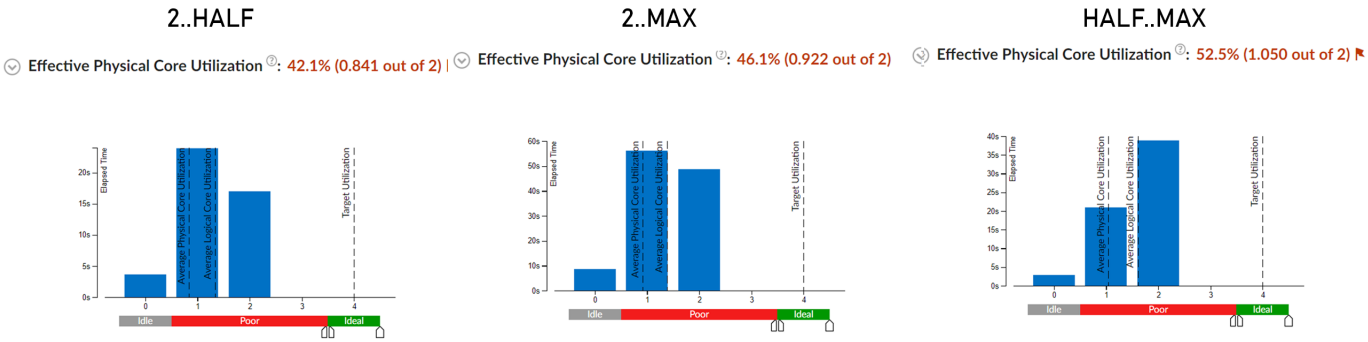

Figure 20: Effective Physical Core Utilization - PAR_DZIEL_NAIVE 2 Wątki

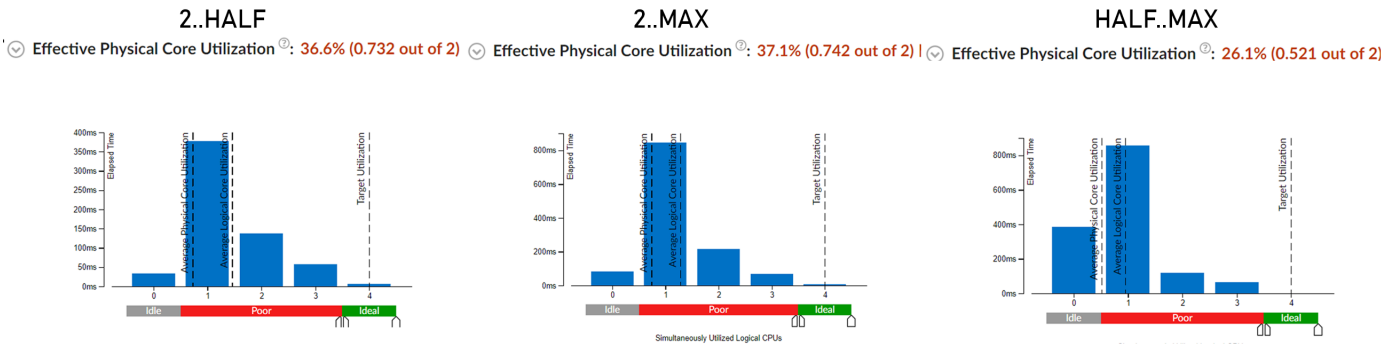

Figure 21: Effective Physical Core Utilization - PAR_WYK_DOM_NAIVE 4 Wątki

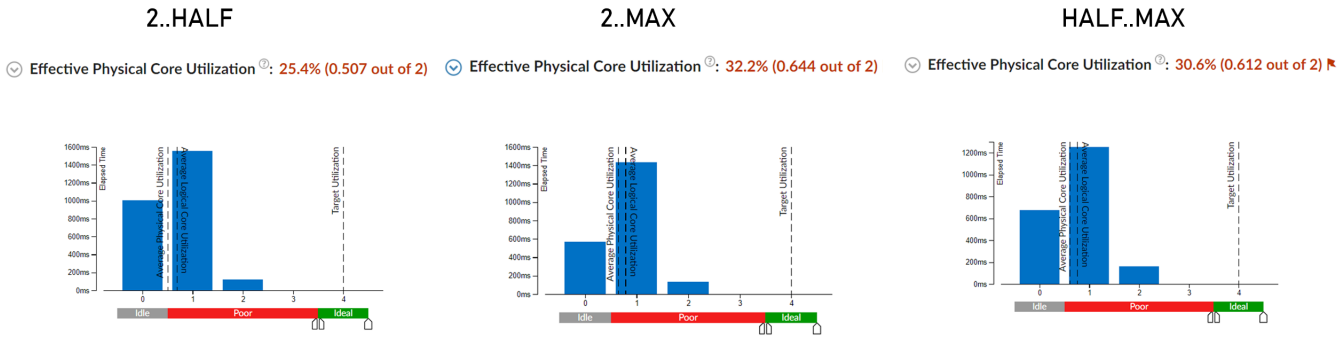

Figure 22: Effective Physical Core Utilization - PAR_WYK_DOM_BETTER 2 Wątki

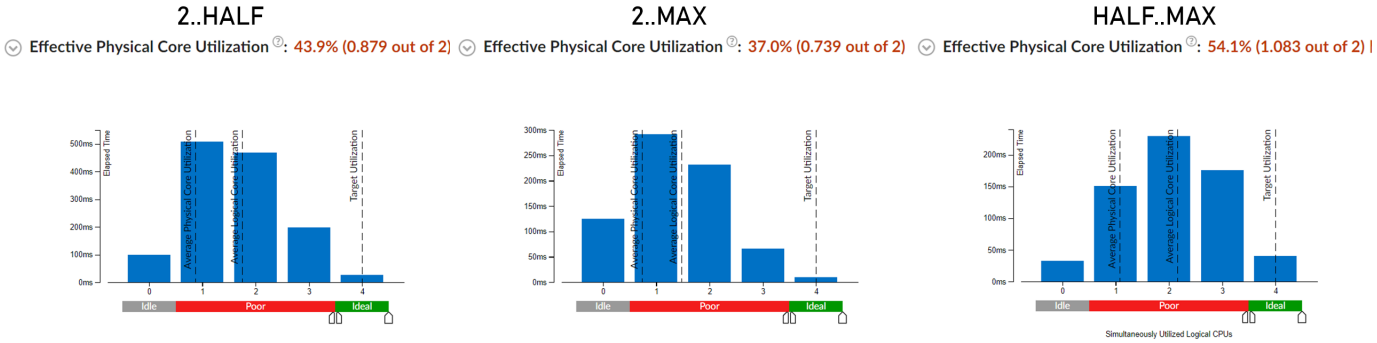

Figure 23: Effective Physical Core Utilization - PAR_WYK_FUNC_OK 4 Wątki

Wykres dotyczący algorytmu SEQ_2 pokazuje, że procentowo przez najwięcej czasu w progamie działało 0 procesorów - program był bezczynny. Nie udało nam się odnaleźć powodów takiej sytuacji; mimo to jest to algorytm, który uzyskał najlepszy czas wykonania ze wszystkich stworzonych w ramach projektu wersji.
Wykres dotyczący wersji PAR_DZIEL_NAIVE pod tym względem wypada lepiej: procentowy udział bezczynności w programie jest nieznaczny (poniżej 10 sekund w najgorszej wersji dla największej instancji). Z kolei przez czas niemal równy działaniu zamierzonej liczby wątków zaangażowany był tylko jeden procesor - wpływ na to może mieć jednak duża ilość kodu dziejąca się poza regionem równoległym.
W przypadku PAR_WYK_DOM_NAIVE przez zdecydowaną większość czasu w kodzie działał jeden procesor lub pojawiała się bezczynność. Kod osiągnął dobre prędkości co budzi skojarzenia z jego wersją sekwencyjną - przez większość czasu działa pojedynczy procesor, stosunek ilości instrukcji assemblera do liczby cykli procesora jest średni w skali wszystkich algorytmów, a prędkość przetworzenia jest spora.
Podobnie wygląda analiza udziału procesorów dla wersji PAR_WYK_DOM_BETTER. Działanie dwóch procesorów na raz dla żadnej instancji nie przekroczyło 200ms, tu jednak praca procesora wg wcześniejszych analiz jest wzmożona, a uzyskany czas średni w skali grupy.
Z kolei w przypadku PAR_WYK_FUNC_OK rozkład pracy procesorów wydaje się bardziej zrównoważony. Czas działania wszystkich wątków na raz nie jest duży, wyróżnia się jednak czas równoległej pracy trzech i dwóch procesorów na raz. Program korzystał z niewielkich zasobów procesora i rozłożony był między działające równolegle wątki nie zakłócające swojej pracy, a prędkości które osiągnął czynią go najszybszym z równoległych algorytmów, które stworzyliśmy dla celów projektu.

4 Wnioski

1. Nie wszystkie użyte podejścia okazały się efektywnie wykorzystywać struktury wewnętrzne procesora. Najlepszym algorytmem pod tym względem okazała się wersja dobra równoległego algorytmu wykreślania funkcyjnego, w której praca rozłożona była na większości dostępnych procesorów, a osiągnięte wyniki czasowe okazały się najlepsze ze wszystkich wersji równoległych. Rozłożenie pracy na większości dostępnych procesorów dotyczy także wersji naiwnej tego algorytmu. Wpływ na to może mieć operowanie na odrębnych fragmentach tablicy wykreśleń przez wątki, gdyż zarządzanie równoległością od strony dyrektyw OMP zarówno tu, jak i w wersji domenowej przetestowano podobnie i w wersji funkcyjnej osiągnięto najlepsze efekty, natomiast w wersji domenowej próby ulepszenia kodu innymi dyrektywami przynosiły nieznaczne tylko w skali wszystkich wyników rezultaty.

2. We wcześniejszym rozdziale przedstawione zostało zestawienie osiągniętych przez algorytmy miar przyspieszenia i efektywności. To zestawienie warto uzupełnić przedstawieniem, jak konkretne, najlepsze wersje algorytmów równoległych "wypadły" w zestawieniu z algorytmami sekwencyjnymi, które miały ulepszyć.
   Wykorzystano formatowanie warunkowe dla ułatwienia szybkiej analizy wizualnej wyników. Algorytm dzielenia

	SEQ_1			PAR_DZIEL_NAIVE		
	1 wątek			4 Wątki		
	2..MAX	HALF..MAX	2..HALF	2..MAX	HALF..MAX	2..HALF
Elapsed time (Microarchitecture Exp.) [s]	158,2	93,513	60,5	75,664	46,676	26,3
Instructions retired	3,78738E+11	2,36798E+11	1,40953E+11	3,78835E+11	2,38083E+11	1,41175E+11
Clockticks	4,3843E+11	2,64153E+11	1,6465E+11	5,50638E+11	3,5876E+11	2,05053E+11
Retiring	51,40%	53,00%	54,10%	62,20%	66,00%	63,30%
Front-end bound	50,40%	5,00%	50,40%	42,10%	42,80%	40,40%
Back-end bound	0%	0%	0%	0,00%	0,00%	0,00%
Memory bound	0%	0%	0%	0,00%	0,00%	0,00%
Core bound	0%	0%	0%	0,00%	0,00%	0,00%
Effective physical core utilization (bottom-up)	36,90%	37,80%	35,30%	64,30%	60,30%	68,70%
Speed [ilość/s]	632111,2389	534685,0171	826446,2479	1321632,454	1071214,329	1901140,608

Figure 24: Zestawienie: algorytm sekwencyjny dzielenia oraz rozbudowująca go wersja równoległa

	SEQ_2			PAR_WYK_DOM_NAIVE			PAR_WYK_FUNC_OK		
	1 wątek			4 Wątki			4 Wątki		
	2..MAX	HALF..MAX	2..HALF	2..MAX	HALF..MAX	2..HALF	2..MAX	HALF..MAX	2..HALF
Elapsed time (Microarchitecture Exp.) [s]	0,047	0,018	0,016	1,22	1,425	0,613	1,298	0,629	0,724
Instructions retired	25000000	20000000	22500000	3380000000	2555000000	2037500000	2,69E+09	1665000000	1282500000
Clockticks	40000000	40000000	30000000	5437500000	4332500000	3387500000	7,003E+09	4467500000	3350000000
Retiring	0%	0%	0%	27,70%	31,90%	34,20%	24,00%	29,20%	16,40%
Front-end bound	72,50%	0%	0%	14,80%	11,70%	21,40%	10,80%	18,20%	10,40%
Back-end bound	27,50%	100%	100%	52,40%	55,80%	42,60%	67%	53,90%	75,80%
Memory bound	0%	0%	0%	35,70%	31,80%	30,80%	51,60%	10,40%	57,50%
Core bound	27,50%	100%	100%	16,70%	24%	11,90%	14,80%	12,50%	18,30%
Effective physical core utilization (bottom-up)	6,40%	18,80%	15,40%	37,10%	26,10%	36,60%	43,90%	54,10%	37%
Speed [ilość/s]	2127659532	2777777778	3124999875	81967211,48	35087719,3	81566065,25	77041601	79491255,96	69060770,72

Figure 25: Zestawienie: algorytm sekwencyjny wykreślania oraz rozbudowujące go wersje równoległe: domenowa i funkcyjna

   udało się ulepszyć praktycznie dwukrotnie pod względem prędkości (Elapsed time), a wraz z nią znacząco wzrosła szybkość wyrażona w ilości przetestowanych liczb na sekundę. Na odwrót jest jednak z algorytmem sekwencyjnym wykreślania, którego wyników żaden z algorytmów równoległych nie zdołał przewyższyć.

3. Choć algorytm równoległy dzielący w wersji naiwnej okazał się o wiele lepszy od wersji sekwencyjnej, jest też jednym z algorytmów osiągających najgorsze wyniki ogólne. Jego prędkość bywa nawet ponad stukrotnie niższa niż najlepsza osiągnięta prędkość równoległa, choć wykorzystując dużo zasobów procesora rozkładał pracę pomiędzy przydzielone wątki.

4. Wśród ograniczeń efektywnościowych w przypadku algorytmów sekwencyjnych oraz równoległych wersji algorytmów dzielenia dominują front-end bound - ograniczenie wejścia (zwykle dla wszystkich instancji ponad 50%). Sytuacja przedstawia się odwrotnie w przypadku pozostałych algorytmów równoległych, gdzie zdecydowaną większość ograniczeń stanowią ograniczenia back-end - ograniczenie wyjścia bound oraz memory bound - ograniczenie systemu pamięci (nie korzystanie z pamięci podręcznej) i core bound - ograniczenie jednostek wykonawczych (oznacza to że duża liczba jednostek wykonawczych procesora zostaje niewykorzystana).