Optymalizacja oprogramowania pod kątem wykorzystania zasobów procesorów wielordzeniowych

Marek Machliński (241308)
Daniel Król (241399)
Prowadzący: mgr inż. Tomasz Serafin

01.06.2019

Spis tresci

1 Wstęp teoretyczny

1.1 Historia wprowadzania procesorów wielordzeniowych

Produkcja procesorów od zawsze miała na celu m.in. zwiększenie dostępnej mocy obliczeniowej i zmniejszenie wielkości jednostki. Przez dłuższy czas pierwsza z tych własności była osiągana za pomocą budowania coraz to bliższych optymalnemu układów wykorzystując nowo odkryte technologie, które zarówno pomagały zwiększać taktowanie jak i minimalizować układ. Jednak w pewnym etapie takie podejście przestało być najbardziej opłacalnym pod względem wydajnościowym i kosztowym. Jednym z czynników było osiągnięcie fizycznej granicy rozmiaru architektury, ponieważ przy pomniejszaniu krzemowych elementów dla technologii 7nm (dokładnie 6,72 nm) bramka ma długość 16 atomów, dla 5 nm jest to 12 atomów, więc teoretyczną granicą (wątpliwą do osiągnięcia przez naukowców) jest technologia 0,5 nm, dla którego długość ramki wynosiłaby 1 atom. Wiąże się to również ze znacznie mniejszym zużyciem energii podczas pracy (krótsze ścieżki), co zmniejsza problem przegrzewania się w procesorach jednordzeniowych. Produkowanie takich urządzeń byłoby niezwykle trudne nawet w momencie dostania się do potrzebnej technologii, a jednocześnie znacznie bardziej kosztowne od sposobu w jaki obecnie zwiększa się wydajność procesorów, mianowicie przez dodawanie większej ilości rdzeni, które mogą zachowywać się jak osobne procesory.

Idea równoległości polega na rozdzielaniu programu na poszczególne mniejsze części wykonywalne, które mogą działać jednocześnie, a ich zasoby są od siebie niezależne. Pomysł został wcześniej wykorzystany np. przy tworzeniu sumatorów, gdzie użycie sumatorów prefiksowych pozwalało na jednoczesne wykonywanie wielu obliczeń, a wyniki ich działania były otrzymywane znacznie szybciej w porównaniu do wcześniejszych ich wersji. Dało to nowy pogląd na rozwiązywanie problemów optymalizacyjnych, ponieważ celem nie było już dążenie do uzyskania jak najmniejszej złożoności pojedynczego procesu, lecz minimalizowanie złożoności procesu o największej złożoności spośród procesów powstałych po rozbiciu głównego zadania na mniejsze, niezależne procedury. Jest to charakterystyczną cechą procesorów wielordzeniowych, które w obecnych czasach wykorzystuje się już powszechnie w większości urządzeń.

Pierwszym procesorem wielordzeniowym ogólnego przeznaczenia był procesor Power 4 produkowany przez IBM. Został wprowadzony na rynek w roku 2001, natomiast pierwszymi procesorami wielordzeniowymi architektury x86 były Opteron od AMD i Pentium Extreme Edition od Intela wprowadzone dopiero w 2005 roku. Obecnie procesory osiągają taktowania rzędu ok. 4,5 GHz, lecz ich rozwój nie charakteryzuje się już wyraźnym wzrostem taktowania, lecz zwiększaniem ilości tranzystorów i rdzeni. Przykładowymi jednostkami w których skupiono się na zwiększeniu wydajności wykorzystując wiele rdzeni jest zaprezentowany w 2007 roku przez Intela układ scalony

Intel Polaris wyposażony w 80 rdzeni, który osiągnął wydajność 1,01 TFLOPS (bilion operacji zmiennoprzecinkowych na sekundę) lub wykonany na Uniwersytecie Kalifornijskim pierwszy 1000-rdzeniowy procesor KiloCore, który oferował moc obliczeniową na poziomie 1,78 TFLOPS. Obecnie na rynku możemy znaleźć jednostki posiadające 6 czy 8 rdzeni, a także, wykorzystywane najczęściej do obsługi serwerów, 16-rdzeniowe. Według prawa Moore'a istnieje prawidłowość w stosunku liczby rdzeni procesorów a liczbą tranzystorów w jednym układzie - według niego ma ona podwajać się co 2 lata.

1.2 Porównanie wydajności względem różnych procesów dla procesorów o różnej ilości rdzeni

Procesory wielordzeniowe z reguły są szybsze od swoich odpowiedników w wersji jednordzeniowej, jednak nie jest to regułą. Aby jednostka wielordzeniowa mogła wykorzystać w pełni swój potencjał powinny być spełnione warunki dotyczące procesu, który będzie przez nią obsługiwany. Podstawową własnością jest możliwość wydzielenie z zadania niezależnych jego części, których dopiero wspólne wyniki będą "łączone" w kolejny etap działania programu. W przeciwnym razie taki program mógłby działać jedynie na jednym rdzeniu, co znacznie spowolniłoby jego wykonanie. Przykładem sprzyjających warunków są obliczenia wykonywane na macierzach, które można robić jednocześnie na poszczególnych jej częściach, a wynikiem będzie macierz otrzymana z połączenia wcześniej dokonanych obliczeń na poszczególnych jej elementach.

Ważnym aspektem jest Hyper-Threading, który pozwala każdemu rdzeniowi zachowywać się jak dwa procesory logiczne, które dzielą między sobą zasoby pamięci podręcznej i jednostek wykonawczych. Gdy jeden z konkurujących ze sobą procesów pozostawia niewykorzystane zasoby, proces przypisany do drugiego procesora logicznego może ich użyć, co w sprzyjających okolicznościach może prowadzić do sumarycznego wzrostu wydajności od kilku do kilkunastu procent. Mimo skomplikowanego i stosunkowo drogiego procesu produkcji procesorów wspierających HT używano go w czterordzeniowych procesorach Intel Core i7 czy jednordzeniowym układzie Intel Atom.

Porównując procesor dwurdzeniowy do jednordzeniowego możemy zauważyć, że ten pierwszy wykona zadanie szybciej, jeśli będzie mógł wykonywać jakieś procesy jednocześnie, ponieważ w tej samej sytuacji wersja jednordzeniowa musiałaby przełączać się za każdym razem pomiędzy zadaniami. Kolejnym aspektem jest zredukowany koszt, ponieważ jeszcze przed spopularyzowaniem procesorów dwurdzeniowych użytkownicy mogli uzyskać dwukrotnie większą moc obliczeniową instalując dwie jednostki procesora na jednej płycie głównej zamiast przymusu kupowania całego drugiego komputera. Z drugiej strony podczas wykonywania nieskomplikowanych operacji typowych dla dzisiejszych przeciętnych użytkowników komputera, którzy sprawdzają pocztę, przeglądają strony

internetowe oparte na tekście czy tworzą dokumenty tekstowe, nie istnieje potrzeba używania większej ilości rdzeni, więc marnują oni niepotrzebnie moc pobieraną przez ich sprzęt. Procesory dwurdzeniowe w tym przypadku zyskują największą przewagę, gdy trzeba wykonywać operacje operujące na grafice lub obrazie. Kolejną wadą rozwiązania dwurdzeniowego jest konieczność pisania programów, które będą w stanie wykorzystywać drugi rdzeń, co znacznie komplikuje proces wytwarzania takiego oprogramowania, ponieważ to programista musi powiedzieć programowi, kiedy powinien on użyć drugiego rdzenia.

W przypadku procesorów czterordzeniowych możemy mówić o najlepszym dopasowaniu do wielozadaniowości, ponieważ oferują one jednocześnie dużą moc obliczeniową w jednym momencie, która zarazem odznacza się dużą integralnością. Procesory odznaczają się także wyprzedzaniem "wyścigu" z oprogramowaniem do którego są przystosowywane, ponieważ rozwój oprogramowania następuje dużo szybciej niż technologii wytwarzania procesorów, więc te drugie muszą nadrabiać swoim zaawansowaniem i w pewien sposób być przygotowane na to, co jeszcze nie nastąpiło w aspekcie oprogramowania. Dzięki temu nawet przestarzałe jednostki, czyli na tej części rynku - 2 lata po premierze, mogą sprostać programom pisanym dużo później niż zostały wydane procesory obsługujące je. Biorąc pod uwagę ich powszechne użycie w przenośnym sprzęcie (laptopy, telefony) należy zwrócić uwagę na zużycie baterii, które może pobierać mniej mocy i produkować mniej ciepła. Z drugiej strony zmniejszają one czas życia baterii szybciej niż wersje jednordzeniowe. Kolejnym aspektem jest kompatybilność sprzętowa, ponieważ procesory czterordzeniowe mogą być instalowane tylko w niektórych płytach głównych, co może powodować problemy przy potencjalnej chęci przeniesienia starego CPU do nowej płyty. Z kolei dostosowanie płyty do posiadanego procesora może spowodować konieczność zakupu sprzętu kompatybilnego z nową płytą, co generuje kolejne koszty.

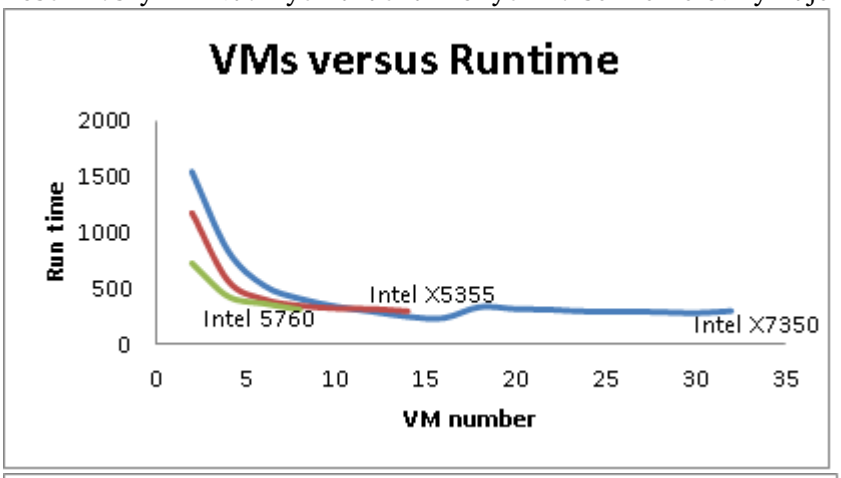
Na początku wszystkie programy były zaprojektowane dla działania na jednym rdzeniu, a systemy operacyjne mogły jednocześnie wykonywać tylko jeden program, co czasami mogło skutkować zawieszeniem systemu (np. w sytuacji, gdy drukowaliśmy jeden dokument, nie mogliśmy robić jednocześnie zupełnie nic innego). Dopiero później systemy zaczęły obsługiwać wielozadaniowość, dzięki której system mógł zmieniać aktualnie wykonywany program, co dawało wrażenie działania wielu programów jednocześnie, gdzie tak naprawdę ciągle wszystko było obsługiwane przez procesor, który jednocześnie wykonywał tylko jeden wątek.

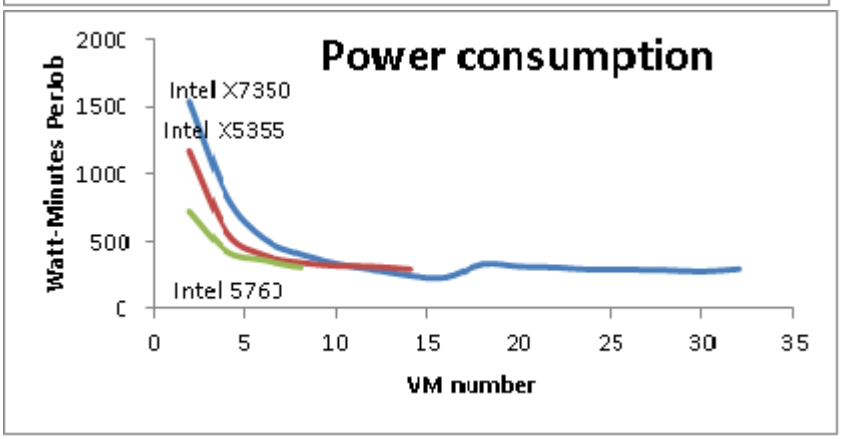
Podczas porównywania efektywności działania programu na wielu rdzeniach należy brać pod uwagę kilka czynników:

- średni procent pozytywnie zakończonych programów
- czas w jakim użytkownik kończy żądanie i system zaczyna odpowiadać

- czas wykonania programu
- moc potrzebna do obsłużenia programu
- opóźnienie czasu odpowiedzi pamięci podczas dostępu do danych
- procent czasu w którym kilka rdzeni próbuje odwołać się do pamięci RAM jednocześnie

Jest to jeden z zestawów parametrów, który może oddawać skuteczność danej jednostki wielordzeniowej. Dla przykładu pewna drużyna z firmy Intel stworzyła serwer bazujący na trzech procesorach: 16-rdzeniowy Quad-Core Intel Xeon CPU X7350, 8-rdzeniowy Quad-Core Intel Xeon CPU X5355 i 4-rdzeniowy Dual-Core Intel Xeon CPU 5160. Serwer ten miał za zadanie uzyskać jak największą wydajność pod względem szybkości i zużycia mocy. Okazało się, że zestawiając czas działania do ilości maszyn wirtualnych uruchumionych na serwerze otrzymuje się następujące wykresy:





Można zaobserwować znaczący spadek wydajności w momencie, gdy ilość maszyn wirtualnych zaczyna przekraczać ilość rdzeni danego procesora. Dla jednostki Intel X7350 czas działania jest prawie stały do momentu, w którym zostaje uruchomiona 16. maszyna - wtedy wykres zaczyna wzrastać. Kosztem tego na drugim wykresie możemy zaobserwować wzrost poboru mocy w podobnych momentach wykresu, dzięki czemu możemy zauważyć zależność pomiędzy tymi dwoma statystykami.

1.3 Sposoby optymalizacji na procesorach wielordzeniowych

1.3.1 Oprogramowanie

Pierwszym paradygmatem optymalizacji pracy na procesorach wielordzeniowych jest używanie optymalnego oprogramowania. Nie można oczekiwać wzrostu wydajności po samym przeniesieniu programu wykonywanego na komputerze z pojedynczym rdzeniem na komputer wielordzeniowy. Aby uzyskać pożądany efekt należy przede wszystkim przepisać cały program z myślą o pracy na wielu procesach i wątkach. System operacyjny zarządza programami jak osobnymi procesami. Każdy proces posiada powiązany kontekst który sprawia wrażenie w perpektywie programu, że ma on dostęp do wszystkich zasobów komputera takich jak CPU, pamięć, układy wejścia/wyjścia. Kiedy proces jest zablokowany z jakieś powodu (np. czeka na wprowadzenie danych w układzie wejścia/wyjścia), to system operacyjny zapisuje jego obecny kontekst, a następnie przyznaje zasoby kolejnemu procesowi. System żongluje kolejnością procesów ze względu na priorytety procesów w taki sposób, aby sprawić wrażenie jak najbardziej responsywnego dla użytkownika. Program wielowątkowy może być napisany w sposób, aby rózne jego sekcje mogły działać jednocześnie i niezależnie. Jest to podobne do działania wielu procesów, jednak wątki różnią się tym, że zużywają dużo mniej zasobów (są "lżejsze"), a w szczególności dzielą tę samą przestrzeń adresową, co pozwala systemowi operacyjnemu na szybkie żonglowanie między nimi, a także używanie tych samych danych. Wielowątkowość jest szczególnie dobrze dopasowana do procesorów wielordzeniowych, ponieważ program napisany w wersji sekwencyjnej może zostać podzielony na osobne wątki, gdzie ich równoległe wykonanie na wielu rdzeniach znacznie przyspieszy proces wykonania.

Ważnym aspektem programów wielowątkowych jest sam proces ich pisania, który z powodu ich skomplikowania podczas tworzenia, debugowania i utrzymywania jest bardzo kosztowny, ponieważ wymaga doświadczonych pod tym względem programistów. Z tego powodu niezbyt często pisze się aplikacje korzystające z benefitów procesorów wielowątkowych, chyba że są to wymagające programy. W większości przypadków klienci decydują się na zakup gotowego oprogramowania zoptymalizowanego pod kątem wielowątkowych procesorów z powodu mniejszych kosztów. Daje to gorsze osiągi oprogramowania, ale nie wymaga tak dużego nakładu pieniędzy jak napisanie aplikacji korzystającej z zasobów procesorów wielowątkowych. Najczęściej powtarzającymi się miejscami w programach do optymalizacji pod tym względem są:

- zadania zależne od warstwy sprzętowej, takie jak interakcja z użytkownikiem, przetwarzanie obrazów, akceptowanie/odrzucanie opcji. Dla przykładu jeśli program musi oczekiwać na jakiś sygnał, gdy w tle następuje wyświetlanie obrazu 60 razy na sekundę, to można wykorzystać

pojedynczy wątek, który będzie obsługiwał ten typ zdarzeń, aby zaoszczędzić zasoby

- aplikacja, w której jednocześnie dzieje się wizualizacja efektów obliczeń (np. przesunięcie postaci w grze), a także właśnie te obliczenia, które jeśli byłyby wykonywane w ramach jednego wątku wraz z wizualizacją, to znacząco spowalniałyby całe wykonanie programu
- każdy element programu, który występuje w nim w kilku niezależnych egzemplarzach, np. w systemie kamer każda z kamer może być osobnym procesem, który zostaje uruchomiony dopiero gdy dana kamera zostanie włączona

1.3.2 Dostrajanie do ogólnej wydajności systemu

Mogłoby wydawać się, że wydzielenie z programu ilości wątków równej ilości rdzeni na danym komputerze powinno najkorzystniejsze rezultaty. Nic bardziej mylnego, ponieważ prawdziwe aplikacje nie są skonstruowane w tak prosty sposób, a moc obliczeniowa procesora musi wciąż zasilać działanie systemu operacyjnego i wszystkich procesów w tle. Dlatego nie zawsze najlepszym rozwiązaniem może być przypisanie każdemu wątkowi osobnego rdzenia. Jedyną metodą wyznaczenia optymalnej liczby rdzeni zarezerwowanej dla danych wątków są testy symulujące realistyczne warunki.

1.3.3 Przenośność oprogramowania

Kolejnym problemem, którego możemy doświadczyć dopiero po pewnym czasie, może być potrzeba uruchomienia oprogramowania na komputerze o innej ilości rdzeni niż poprzednio zakładano. Powoduje to, że należałoby tworzyć aplikacje, które wraz ze wzrostem liczby rdzeni, przez które mogą być one obsługiwane powinny wzrastać osiągi takiej aplikacji. Z czasem ten problem stał się na tyle powszechny, że obecnie biblioteki oprogramowania mogą automatycznie wykrywać ilość rdzeni na jakiejś pracują, aby móc dopasować do niej ilość tworzonych wątków. Pozwala to na wcześniej opisane uruchomienie i szybsze działanie tej samej aplikacji na ośmiu rdzeniach niż na czterech bez zmian w kodzie, co zaoszczędza wielu problemów podczas projektowania większych aplikacji.

2 Opis projektu

2.1 Geneza i założenia

Pierwszym krokiem w realizacji tematu "optymalizacji oprogramowania pod kątem wykorzystania zasobów procesorów wielordzeniowych" był wybór zadania, które mogłoby odpowiednio wiz-

ualizować rozpatrywany problem. Szukając zadania, które nawet abstrahując od programowania w swoich założeniach operowało na wielu obiektach lub zadaniach wykonywanych jednocześnie, trafiliśmy na problem jedzących filozofów. Jednak po wstępnej analizie doszliśmy do wniosku, że mimo że ten problem porusza zagadnienia związane z wielowątkowością, to niezbyt dobrze wyrażałby całość tematu ze względu na brak wydzielenia wymagających procesów, co znacznie utrudniłoby opis postępu optymalizacji i analizę wyników. Problem ten nadawałby się do przedstawienia dostępu do wspólnych zasobów procesom, jednak w naszym przypadku okazało się to nieprzydatne. Kolejnym pomysłem, który z kolei doszedł do fazy realizacji, było przetwarzanie obrazów, a dokładnie tworzenie ich negatywów. Jest to proces często przytaczany w zagadnieniach opartych o wykorzystanie zasobów procesorów wielordzeniowych, ponieważ jest szczególnie kosztowny wraz z operacjami na coraz to większych obrazach, a jednocześnie łatwo skalowalny w celach testów, ponieważ tworzenie bardziej wymagającego obiektu testowego polega jedynie na zwiększeniu wymiarów bitmapy. Kolejnym aspektem przemawiającym za słusznością wyboru algorytmu jest stosunkowo mała złożoność pojedynczego obiektu, na którym przeprowadzane były operacje, mianowicie wystarczyło powtarzać pewne zadanie dla każdej z barw RGB piksela w bitmapie.

Realizując ten projekt mieliśmy na celu pokazanie, w jaki sposób można uwydatnić działanie oprogramowania na procesorach wielordzeniowych w porównaniu z procesorami jednordzeniowymi. Na początku należało zdobyć wiedzę związaną z warstwą sprzętową odpowiadającą za cały proces optymalizacji, a mianowicie wielordzeniowym CPU. Wiedząc, w jaki sposób odbywa się zarządzanie aplikacjami potrafiącymi je wykorzystywać, postanowiliśmy stworzyć trzy programy tworzące negatyw obrazu na różne sposoby odnosząc się do korzystania z wielowątkowości:

- **sekwencyjnie** - najbardziej podstawowy sposób opierający się na korzystaniu z zasobów zakładając, że korzystamy tylko z procesora jednordzeniowego, przez co tworzy on tylko jeden wątek, który przechodzi po każdym z pikseli po kolei przetwarzając go do formy wynikowej. Ta część programu kończy się razem z momentem przetworzenia ostatniego piksela, następnie prezentowane są wyniki.

- **równolegle z predefiniowanym dzieleniem procesów między rdzenie** - program na początku dzieli bitmapę na w przybliżeniu równe części, których ilość jest równa ilości rdzeni (w naszym przypadku cztery), a następnie każdy z rdzeni przetwarza osobny proces zajmujący się daną częścią bitmapy. Na koniec wyniki z wszystkich rdzeni są łączone w jeden dając wyjściową bitmapę.

- **równolegle z przydzielaniem rdzeniom procesów w czasie biegu programu** - program na początku dzieli bitmapę na mniejsze bitmapy składające się z pojedynczych wierszy wejściowej

bitmapy, a następnie kolejkuje je do wykonania jako osobne procesy przez zbiór rdzeni (pierwszy wiersz do negacji pikseli zostanie obsłużony przez pierwszy w tej chwili wolny rdzeń). Na końcu wszystkie wyniki procesów są łączone w jeden, z którego powstaje bitmapa wynikowa.

Głównym celem projektu było porównanie efektywności tych trzech metod ze względu na czas wykonania w dziedzinie rozmiaru bitmapy. Takie zestawienie dałoby informację, który ze sposobów przetwarzania jest najbardziej efektywny lub czy któregoś ze sposobów warto używać tylko w specyficznych sytuacjach. Następnie przeszlibyśmy do przedstawienia wniosków, z jakich powodów potencjalnie mogły występować różnice w tych algorytmach i odnieść je do architektury opartej o właściwości procesorów wielordzeniowych. Takie zestawienie pozwala na odpowiedzenie na pytania związane z opłacalnością jednostek wielordzeniowych, a także pozwala lepiej zrozumieć obecny rozwój w ich stronę.

2.2 Implementacje rozwiązań problemu tworzenia negatywu obrazu

Implementacja sekwencyjnego rozwiązania była najprostszą drogą rozwiązania problemu ze strony programisty, ponieważ nie musiał on w żaden sposób przejmować projektowaniem aplikacji zawierającej skomplikowane mechanizmy wielowątkowości. Na początku program wczytywał obraz jako bitmapę na podstawie nazwy pliku, aby wczytać go do pamięci programu. Sam algorytm przetwarzania pojedynczego piksela nie sprawił żadnego problemu (polegał jedynie na ustawieniu wszystkich wartości koloru RGB danego piksela na różnicę maksymalnej wartości koloru (255) i obecnej jego wartości). Przeprowadzenie tej operacji gwarantowało otrzymanie poprawnego koloru piksela, więc po tym można było przejść już do przetwarzania kolejnego piksela. Po przetworzeniu ostatniego piksela operacja była automatycznie zakończona, ponieważ pracowaliśmy na oryginalnym obrazie i nie było potrzeby sumowania żadnych wyników. Ostatnią częścią programu (powtarzającą się w finalnym etapie każdej z implementacji programu) było zaprezentowanie efektów w formie wyświetlenia obrazu, a także zapisanie danych na temat danego uruchomienia do pliku (wymiary obrazu, czas przetwarzania i wykorzystana metoda).

Kolejną implementacją rozwiązania był program możliwy do uruchomienia w dwóch trybach definiowanych za pomocą jednego z parametrów wejściowych. Po wybraniu nazwy pliku i pierwszej z dwóch wartości parametru program sprawdzał, czy bitmapa posiada liczbę wierszy mniejszą od liczby rdzeni procesora. W takim przypadku przydzielał za pomocą maski bitowej pojedyncze rdzenie procesora do każdego z wierszy. W przypadku liczby wierszy przekraczającej liczbę dostępnych rdzeni program uruchomiony z tym parametrem dzielił wszystkie wiersze z wczytanej bitmapy na poszczególne rdzenie procesora za pomocą maski, co w naszym przypadku skutkowało podzieleniem

wierszy na 4 mniejsze bitmapy, gdzie każda z nich była obsługiwana przez osobny proces obsługiwany przez pojedynczy rdzeń. Samo przetwarzanie w poszczególnych częściach bitmapy odbywało się wtedy na takiej samej zasadzie jak w wersji sekwencyjnej programu z tą różnicą, że po zakończeniu przetwarzania każda z części była zapisywana osobno, a po zakończeniu pracy wszystkich rdzeni następowało łączenie otrzymanych obrazów w bitmapę wynikową. Dane na temat uruchomienia (wymiary obrazu, czas wykonania i jego tryb) zostają zapisane do pliku.

Dla drugiej wartości parametru powyższego trybu programu na początku następowało to samo sprawdzenie, czy liczba wierszy bitmapy jest mniejsza od ilości rdzeni procesora i w przypadku twierdzącej odpowiedzi następuje ten sam scenariusz co dla pierwszej wartości parametru. Natomiast w przeciwnym przypadku program tworzył liczbę procesów równą liczbie wierszy, a następnie przydzielał każdemu z tych procesów przetwarzających po jednym wierszu osobny rdzeń. Po zakończeniu przetwarzania danego wiersza program przydzielał zwolniony rdzeń dla kolejnego wiersza do momentu wyczerpania puli wierszy. Po zakończeniu przetwarzania wszystkich wierszy następowało łączenie wcześniej otrzymanych mniejszych bitmap w jedną, wynikową. Tak jak wcześniej program również mierzył czas wykonania i zapisywał razem z resztą swoich parametrów do pliku.

3 Testy aplikacji i analiza wyników

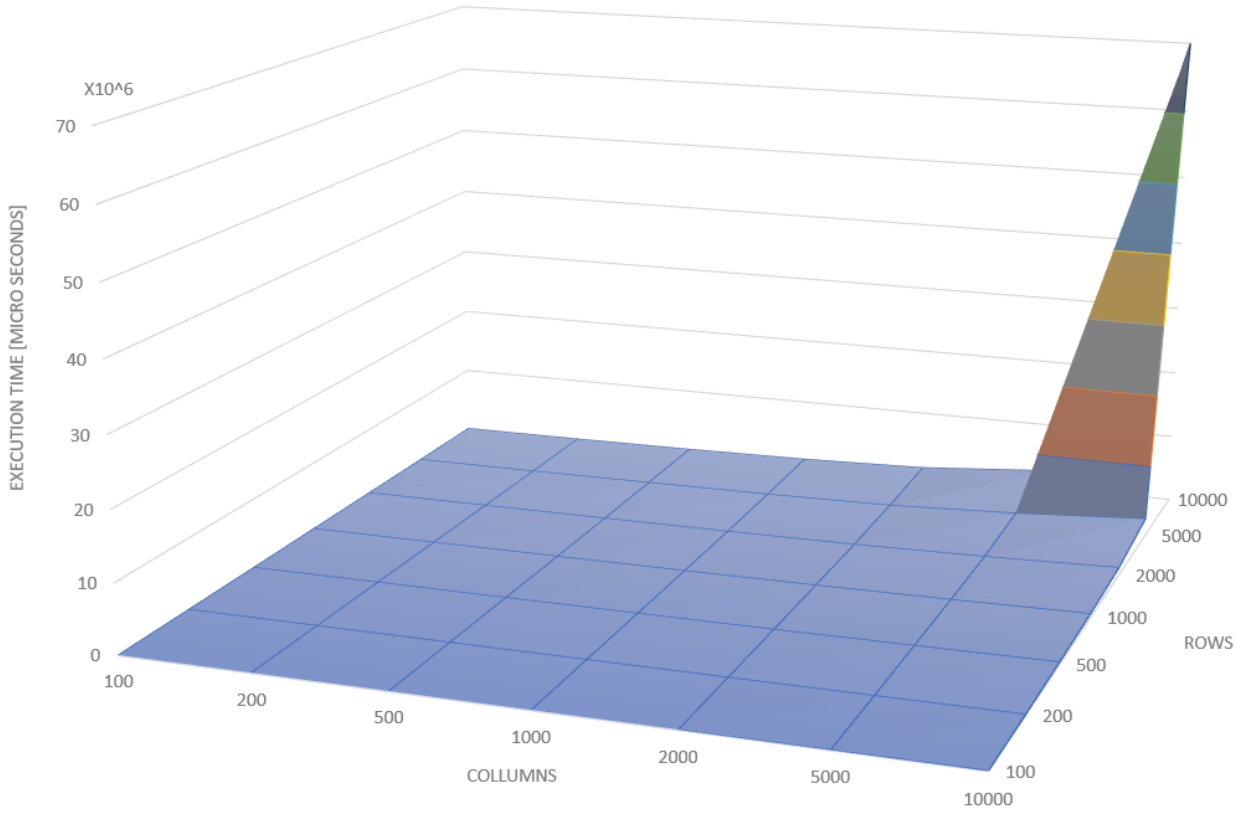
Do automatycznej obsługi programów realizujących w sumie trzy tryby programów stworzyliśmy kolejny program, który uruchamiał wcześniej opisane programy z wcześniej zadanymi parametrami po 100 razy każdy, pobierał dane z każdego wyniku uruchomienia, a następnie uśredniał wartości ich czasów. Dzięki temu otrzymaliśmy zestawienie wyników dla 7 różnych wymiarów bitmap, gdzie każda z nich była przetwarzana w negatyw na 3 różne sposoby, a każde z tych przetworzeń było powtórzone 100 razy. Tak zapisane dane zapisaliśmy w formie podatniejszej na wizualizacje, aby następnie stworzyć wykresy i tabele otrzymanych zależności.

W celu sprawdzenia jak każdy z algorytmów radzi sobie ze wzrostem wymiarów obrazu stworzyliśmy tabele dla każdego z nich, które zestawiały średni czas wykonania z ilością wierszy i kolumn bitmapy. Dodatkowo na ich podstawie wygenerowaliśmy trójwymiarowe wykresy, które ułatwiają porównanie otrzymanych danych:

1 Tabela metody sekwencyjnej

Height\Width	100	200	500	1000	2000	5000	10000
100	249	1532	2451	7074	14739	28323	60293
200	715	2772	7973	17041	28899	58974	124161
500	3021	7581	18058	32816	64305	158260	286800
1000	8679	15280	34844	63489	115329	288869	561450
2000	15050	26559	64385	138625	229438	551788	1107195
5000	32192	56158	177514	304878	551428	1353016	2584664
10000	63973	116831	293924	560528	1079983	2588585	70145445

2 Wykres metody sekwencyjnej

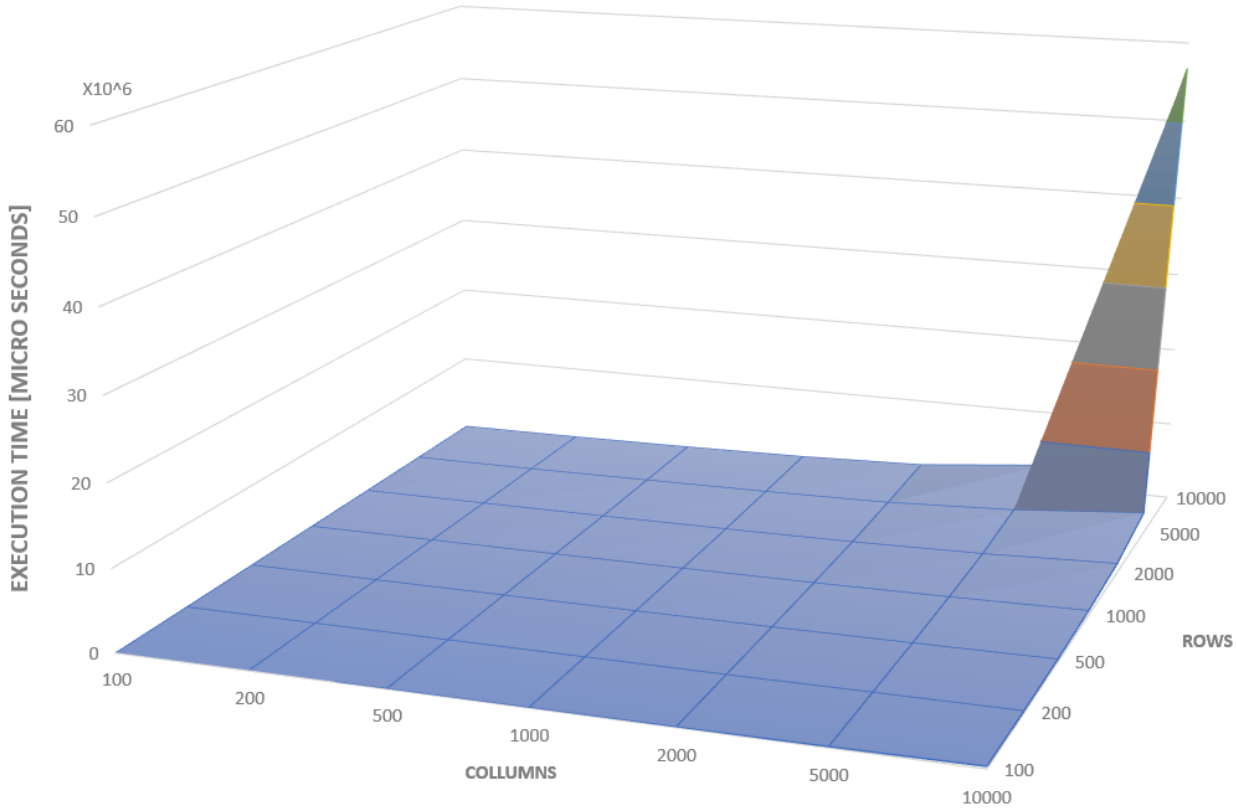


W metodzie sekwencyjnej możemy zauważyć, że oferuje ona dość przewidywalne wyniki i prawdopodobnie łatwo byłoby oszacować wyniki dla kolejnych rozmiarów bitmap. Charakteryzuje się ona najbardziej gwałtownym spadkiem wydajności w obrębie największych zmian parametrów tak jak byśmy tego oczekiwali. Jest to prawdopodobnie spowodowane brakiem możliwości optymalizacji zadanego procesu przez sprzęt w tej metodzie.

3 Tabela metody równoległej z przydzielaniem predefiniowanym

Height\Width	100	200	500	1000	2000	5000	10000
100	10991	5005	4705	10289	15912	28523	54783
200	10130	8371	7383	14687	25937	58738	109483
500	16100	16359	16913	31820	53900	179175	282887
1000	30010	33767	29441	61708	113200	312627	543347
2000	60030	61235	56585	125791	226643	559233	1060674
5000	134924	142168	156973	291407	546085	1316073	2674381
10000	264658	200043	296624	566830	1143922	2702359	56739167

4 Wykres metody równoległej z przydzielaniem predefiniowanym



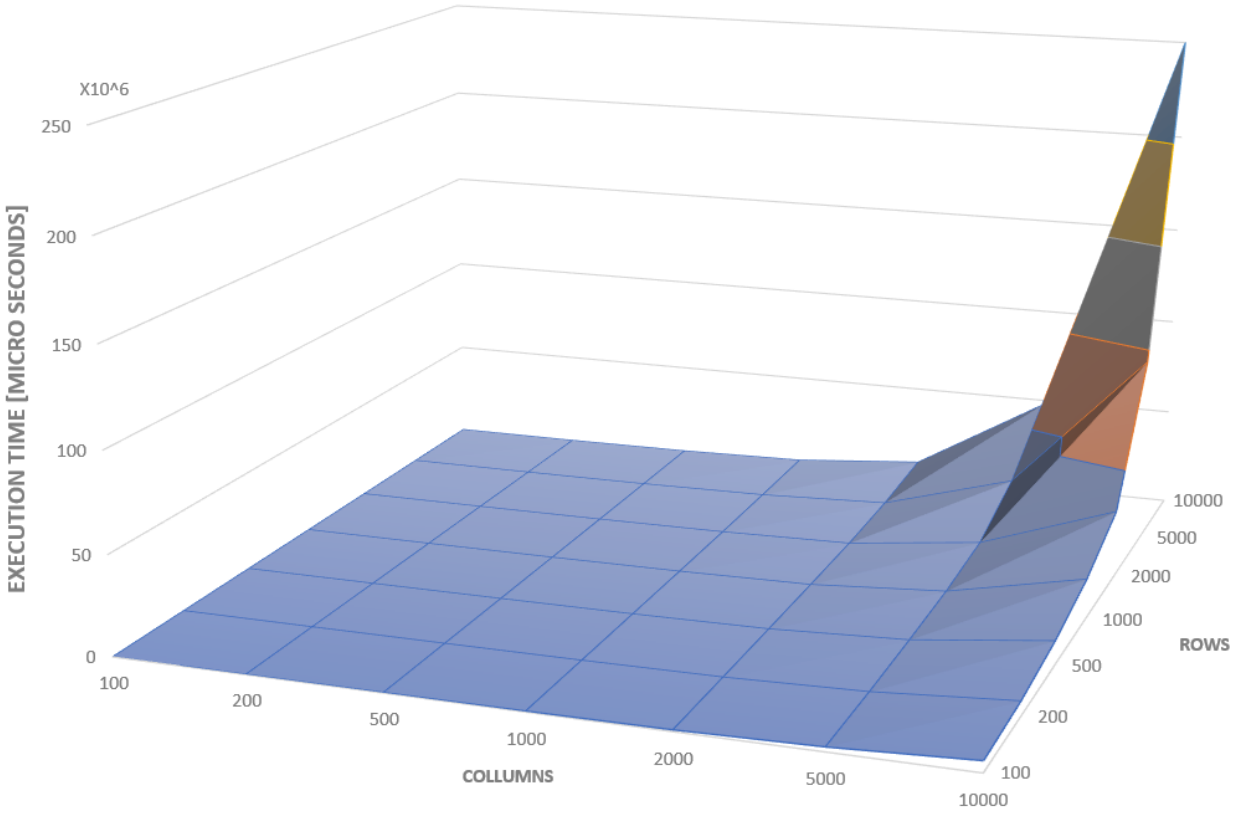
W metodzie równoległej z predefiniowanym przydzielaniem (przed rozpoczęciem algorytmu) wierszy procesom, a tych procesów konkretnym rdzeniom możemy zauważyć znaczne różnice dla małych rozmiarów obrazów w porównaniu do poprzedniej metody, co prawdopodobnie wynika z konieczności tworzenia dodatkowych procesów, a następnie łączenia otrzymanych z nich wyników przed zakończeniem programu. Nadaje to wynikom charakter pomiaru z dodanym błędem stałym, który w tym przypadku odpowiada czasowi wcześniej wymienionych czynności do przygotowania. Jednak wraz ze wzrostem wielkości obrazów ta inwestycja zaczyna się opłacać, ponieważ dzięki

wykorzystaniu większej mocy obliczeniowej algorytm osiąga krótsze czasy niż dla wersji sekwencyjnej.

5 Tabela metody równoległej z przydzielaniem w biegu programu

Height\Width	100	200	500	1000	2000	5000	10000
100	45199	38002	38710	39770	42248	50939	64990
200	75467	75291	77561	81418	90475	119570	164677
500	186543	191327	205087	230167	283643	426150	630849
1000	379452	397597	455860	557459	747820	1263277	1998439
2000	790402	922614	1143900	1477291	2061087	3884408	7714340
5000	2246665	2743027	4050611	5990007	10023267	23334130	47157823
10000	5480725	7350447	12063483	20119852	33839834	94220818	253689257

6 Wykres metody równoległej z przydzielaniem w biegu programu



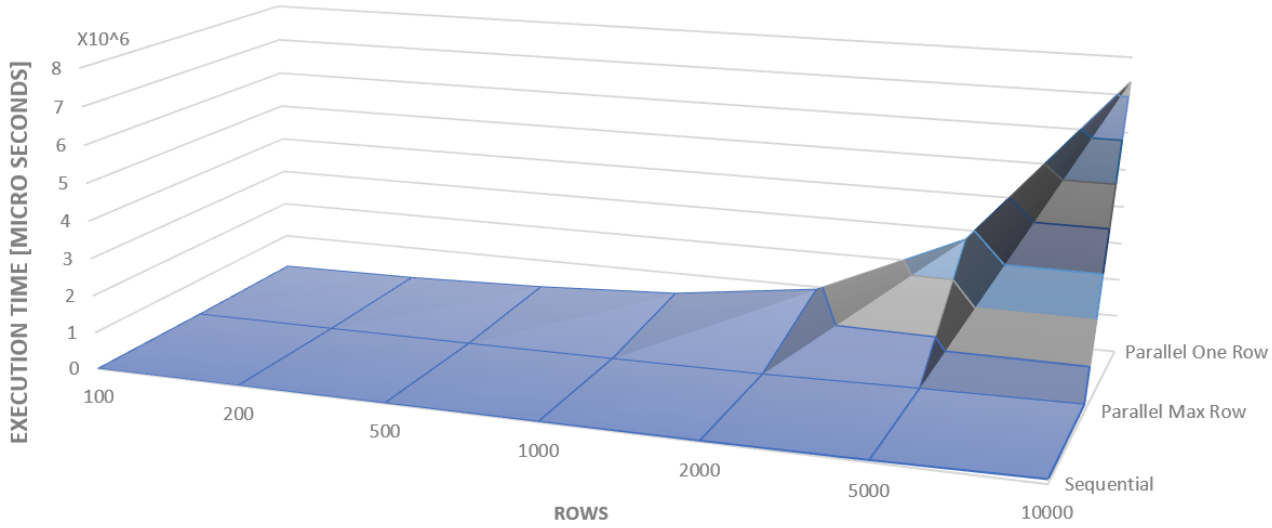
W metodzie równoległej z przydzielaniem wierszy do procesów (a tych do rdzeni) podczas biegu programu możemy zaobserwować znacznie słabszą wydajność względem poprzednich podejść. Jest to spowodowane tworzeniem olbrzymiej liczby procesów dla każdego z obrazów, co jest kosztowną operacją. Koszt stworzenia jednego procesu jest stosunkowo duży w porównaniu do zadania wykonywanego przez ten proces (przetworzenie pojedynczego wiersza), więc zgodnie z naszymi przewidywaniami ta metoda okazała nieefektywna. Zastosowanie jej nie daje przewagi przy żadnej

wielkości obrazu co można stwierdzić przez kształt powierzchni na wykresie, który znacznie szybciej zaczyna się "podnosić", aby ostatecznie osiągnąć maksymalny czas wykonania spośród wszystkich metod.

7 Tabela porównania wszystkich metod dla 200 kolumn

Rows	100	200	500	1000	2000	5000	10000
Sequential	1532	2772	7581	15280	26559	56158	116831
Parallel Max Row	5005	8371	16359	33767	61235	142168	200043
Parallel One Row	38002	75291	191327	397597	922614	2743027	7350447

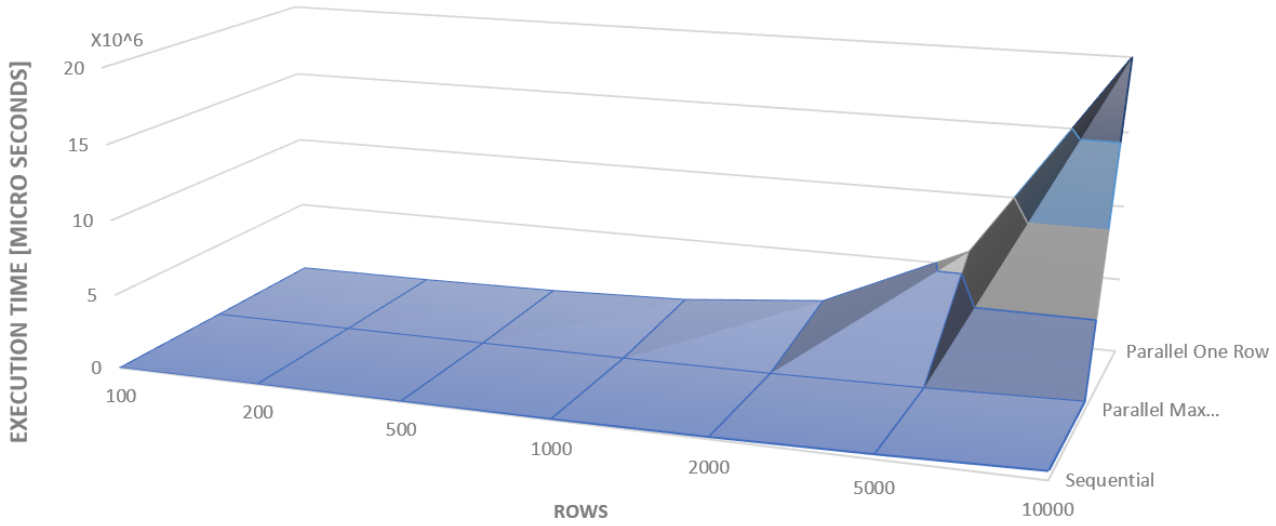
8 Wykres porównania wszystkich metod dla 200 kolumn



9 Tabela porównania wszystkich metod dla 1000 kolumn

Rows	100	200	500	1000	2000	5000	10000
Sequential	7074	17041	32816	63489	138625	304878	560528
Parallel Max Row	10289	14687	31820	61708	125791	291407	566830
Parallel One Row	39770	81418	230167	557459	1477291	5990007	20119852

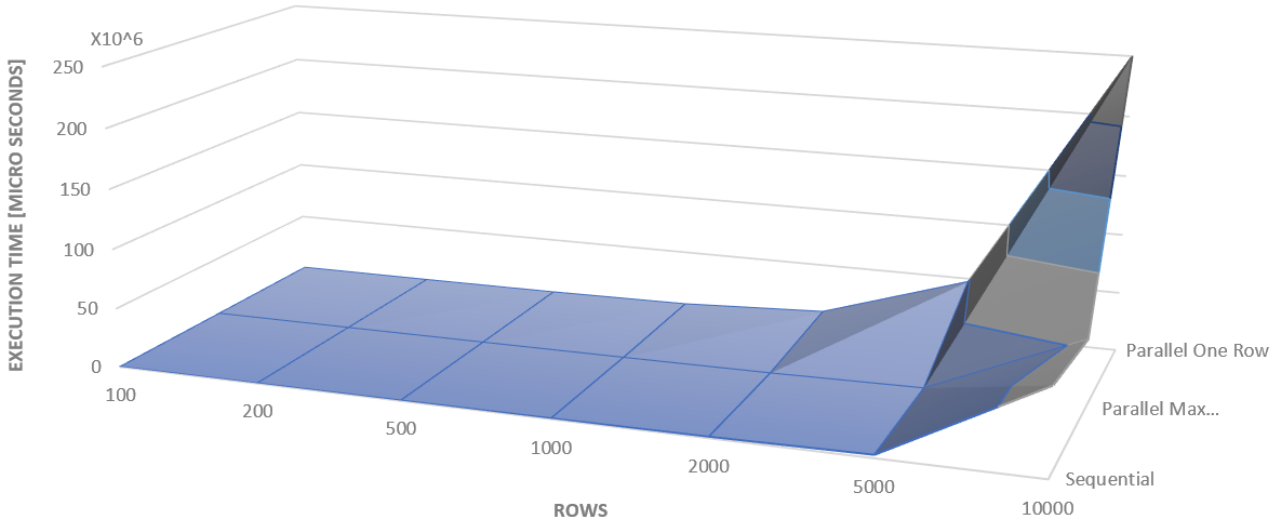
10 Wykres porównania wszystkich metod dla 1000 kolumn



11 Tabela porównania wszystkich metod dla 10000 kolumn

Rows	100	200	500	1000	2000	5000	10000
Sequential	60293	124161	286800	561450	1107195	2584664	70145445
Parallel Max Row	54783	109483	282887	543347	1060674	2674381	56739167
Parallel One Row	64990	164677	630849	1998439	7714340	47157823	253689257

12 Wykres porównania wszystkich metod dla 10000 kolumn



Porównując wyniki wszystkich trzech metod dla 200, 1000 i 10 000 kolumn możemy zauważyć ważną zależność: korzystając z metody sekwencyjnej osiągamy wyniki drugie w kolejności co do wydajności, jednak koszt wykonania tego algorytmu przez programistę był znacznie mniejszy od pozostałych, ponieważ nie zawierał elementów wielowątkowości. W stosunku do swoich efektów wymaga on stosunkowo niedużej ilości czasu do implementacji, a jego kod na pewno byłby łatwiejszy do późniejszej modyfikacji czy rozbudowy. Korzystając z metody równoległej z przydzielaniem procesów w czasie biegu programu osiągamy zdecydowanie najgorsze wyniki, które w żaden sposób nie są rekompensowane. Ta metoda ma swoje odpowiedniki w tańszych i efektywniejszych wersjach, więc prawdopodobnie nie jest warta implementacji w żadnym przypadku związanym z tym zagadnieniem. Jedyną zaletą, jaka potencjalnie może działać w jej obronie jest fakt, że przy tak dużej ilości procesów możemy często wykonywać zadania poboczne pomiędzy poszczególnymi procesami jeśli byłaby taka potrzeba, jednak w naszym przypadku to nie zaszło. Metodą która uzyskała najlepsze rezultaty okazało się podejście równoległe z predefiniowanym przydzielaniem rdzeni, które tworzy minimalną liczbę procesów wymaganą do pełnego użycia procesora, dzięki czemu nie marnuje niepotrzebnie zasobów. Jest to prawdopodobnie podejście, które powinno być wykorzystywane najczęściej w tego typu problemach, ponieważ oferuje największą efektywność. Istnieją także sytuacje, w których warto rozważyć metodę sekwencyjną, szczególnie jeśli z danej operacji nie korzystamy zbyt często, ponieważ w takim przypadku wydajność spadnie nieznacznie, a niski koszt utrzymania takiego kodu może być bardziej opłacalny.