AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INŻYNIERII METALI I INFORMATYKI PRZEMYSŁOWEJ

KATEDRA INFORMATYKI STOSOWANEJ I MODELOWANIA

Analiza interakcji sprzętu i oprogramowania dla mnożenia macierz-wektor dla macierzy rzadkich
Hardware-software interaction analysis for sparse matrix-vector multiplication

Autor: Malwina Paulina Cieśla
Kierunek studiów: Informatyka Techniczna
Opiekun pracy: Dr hab. inż. Krzysztof Banaś

Kraków, rok 2024

Praca została wykonana z wykorzystaniem Infrastruktury PLGrid.
Praca zrealizowana w oparciu o wyniki uzyskane przy wykorzystaniu zasobów obliczeniowych ACK CYFRONET AGH.

Spis treści

1. Wstęp

Mnożenie macierz-wektor dla macierzy rzadkich (ang. *Sparse matrix-vector multiplication, spMV*) jest uznawane za kluczową, fundamentalną i niejednokrotnie najbardziej czasochłonną procedurę dla wielu algorytmów numerycznych, w efekcie czego zostało już szeroko przebadane na wszystkich architekturach nowoczesnych procesorów i akceleratorów. Głównie wykorzystywane jest przy iteracyjnym znajdowaniu rozwiązań układów równań liniowych, gdzie występuje wielokrotne mnożenie macierz-wektor. Rozwiązywanie takich układów może pojawić się w programach aproksymacji równań różniczkowych cząstkowych metodą elementów skończonych, MES (ang. *finite element method* – FEM), a także różnic skończonych, MRS (ang. *finite difference method* – FDM) i objętości skończonych (ang. *finite volume method* – FVM). W takich sytuacjach może także (przy rozwiązywaniu problemów nieliniowych lub zależnych od czasu) występować wielokrotne rozwiązywanie układów równań liniowych, przyczyniające się do wydłużenia czasu trwania obliczeń całego programu. Z tej przyczyny problem wydajnej implementacji procedury mnożenia macierz-wektor dla macierzy rzadkich jest tak ważny.

W niniejszej pracy przeprowadzono analizę interakcji między sprzętem, a oprogramowaniem w kontekście optymalizacji operacji mnożenia macierzy rzadkiej przez wektor. Głównym celem badania było porównanie wydajności kodu uruchamianego w środowisku wielowątkowym z wykorzystaniem OpenMP oraz w środowisku równoległym z użyciem MPI, a także porównanie tych wyników z obliczeniami jednowątkowymi. W ramach analizy zastosowano różne specjalne formaty przechowywania macierzy rzadkich, a także przeanalizowano generowany kod asemblera, aby uzyskać głębsze zrozumienie sposobu wykonywania operacji na poziomie sprzętowym. Ponadto, wykorzystano liczniki sprzętowe do monitorowania kluczowych parametrów wydajności, takich jak liczba operacji zmiennoprzecinkowych, cache misses oraz zużycie pamięci. Ostateczne wyniki analizy zostały przedstawione za pomocą wykresów i tabel, które obrazują miary wydajności, w tym czas wykonania oraz efektywność obliczeń w różnych konfiguracjach wielowątkowych i równoległych.

W pierwszym rozdziale pracy „Mnożenie macierz-wektor dla macierzy rzadkich” opisane zostały podstawowe pojęcia związane z operacją mnożenia macierz-wektor oraz przedstawiono charakterystykę macierzy rzadkich. W rozdziale omówiono także popularne formaty przechowywania macierzy rzadkich, takie jak CSR (Compressed Sparse Row) czy SELL (Sliced ELLPACK), oraz algorytmy mnożenia macierz-wektor, w tym algorytmy dedykowane dla obliczeń

równoległych. W drugim rozdziale „Współczesne mikroprocesory wielordzeniowe i systemy wieloprocesorowe" przedstawiono budowę i funkcjonowanie mikroprocesorów, ze szczególnym uwzględnieniem ich architektury wielordzeniowej i systemów wieloprocesorowych z pamięcią wspólną. Omówione zostały m.in. przetwarzanie rozkazów i pamięci podręczne, a także specyfika systemów wieloprocesorowych. Rozdział „Obliczenia równoległe i wektoryzacja kodu" opisuje koncepcje obliczeń równoległych oraz techniki wektoryzacji kodu, które są kluczowe dla optymalizacji wydajności. Omówione zostały tutaj specyfikacje OpenMP dla systemów z pamięcią wspólną oraz MPI dla systemów z pamięcią rozproszoną, a także zasady wektoryzacji kodu z użyciem wewnętrznych rejestrów wektorowych. W rozdziale „Wydajność obliczeń" przedstawione zostały podstawowe miary wydajności obliczeń, takie jak miary arytmetyczne (zmiennoprzecinkowe) oraz względne miary wydajności obliczeń równoległych. Rozdział analizuje również wpływ kompilatorów optymalizujących na wydajność oraz omawia klasyczne techniki optymalizacyjne stosowane przez kompilatory. Rozdział „Narzędzia wspomagające analizę wydajności obliczeń" opisuje narzędzia do profilowania i liczniki sprzętowe używane do analiz wydajnościowych. Przedstawia również narzędzia takie jak perf oraz interfejs PAPI, które są wykorzystywane do analizy zdarzeń sprzętowych podczas mnożenia macierz-wektor. W pracy rozdziały „Implementacja mnożenia macierz-wektor dla wybranych formatów przechowywania" oraz „Eksperymenty obliczeniowe i analiza wydajności" przedstawiają opis użytej implementacji algorytmów mnożenia macierz-wektor dla wybranych formatów macierzy rzadkich, takich jak CSR i SELL oraz przeprowadzone obliczenia na różnych systemach obliczeniowych, takich jak serwer Honorata oraz superkomputer Ares. Analizowane są wyniki jednowątkowego i wielowątkowego wykonania programu. Eksperymenty obejmują zarówno wykorzystanie OpenMP, jak i MPI. Dodatkowo każda implementacja jest analizowana na poziomie kodu źródłowego oraz w kontekście fragmentów kodów asemblera generowanych dla wszystkich badanych formatów przez kompilator.

2. Mnożenie macierz-wektor dla macierzy rzadkich

2.1 Mnożenie macierz-wektor

Algorytm mnożenia macierz-wektor można przedstawić jako matematyczne równanie y=Ax, w którym należy zastosować macierz A o wymiarach MxN, gdzie wymiary M i N to odpowiednio liczba wierszy i liczba kolumn oraz wektor x, którego rozmiar N przedstawia liczbę wierszy wektora (w mnożeniu macierz-wektor rozmiar wektora jest równy liczbie kolumn macierzy). Po wymnożeniu macierzy A i wektora x otrzymany zostanie wektor y o wymiarze M. Mnożenie to może również zostać zapisane jako realizacja odwzorowania z przestrzeni wektorowej N-elementowej do przestrzeni wektorowej M-elementowej.

Poniżej przedstawiony został podstawowy algorytm mnożenia macierz-wektor implementujący wzór y=Ax zapisany w reprezentacji tablicowej w języku C:

```
for (int i = 0; i < liczba_wierszy; i++){
        for ( int j = 0; j < liczba_kolumn; j++){
                y[i]+=A[i][j]*x[j];
        }
}
```

gdzie:

- macierz A przechowywana jest w sposób standardowy w tablicy dwuwymiarowej,
- wektor y został wstępnie zainicjowany zerami

W podanym algorytmie przy zmianie kolejności pętli zakresy zmiennych „i” oraz „j” nie zostaną zmienione, a element A[i][j] oznaczać zawsze będzie wartość w i-tym wierszu i j-tej kolumnie. Zakresy mogą jednak ulec zmianie kiedy zmienna „i” będzie oznaczać kolumny, a zmienna „j” wiersze. W pracy do zdefiniowania macierzy użyto oznaczeń i-ty wiersz oraz j-ta kolumna.

2.2 Macierze rzadkie

Macierze, które wykorzystywane są w algorytmie mnożenia macierz-wektor można podzielić na macierze: standardowe, inaczej nazywane gęstymi oraz rzadkie. Macierze traktowane są jako gęste, gdy w wartościach znajduje się znikoma ilość elementów zerowych. Macierze rzadkie natomiast są rodzajem macierzy, w którym większość elementów ma wartość zerową. Nie występuje żadna wartość graniczna rozdzielająca macierze na rzadkie i gęste, a stosowanie algorytmów i nazewnictwa wybranych rodzajów macierzy jest używane w zależności od zadania oraz programisty. W następstwie tego rozmiary oraz ilość niezerowych elementów w macierzy jest niejednolita.

Można jednak przyjąć nazywanie macierzy rzadką, kiedy algorytmy korzystające ze specjalnych formatów przechowywania wyłącznie wyrazów niezerowych działają szybciej niż algorytmy używające standardowego przechowywania macierzy.

2.3 Formaty przechowywania macierzy rzadkich

Podczas obliczeń wykonywanych na dużych macierzach często zauważa się, że większość występujących elementów to zera. Przechowywanie wszystkich wartości znacznie zwiększa zajmowane przez macierz miejsce w pamięci, a także wydłuża czas wykonywanych operacji arytmetycznych. W celu minimalizacji tych czynników stworzone zostały dodatkowe formaty do przechowywania macierzy rzadkich. Specjalne algorytmy dla macierzy rzadkich uwzględniają unikalne metody przechowywania, gdzie w pamięci komputera są przechowywane jedynie niezerowe elementy macierzy. Zdarza się jednak, że w niektórych formatach może występować także niewielka część elementów zerowych.

Istnieje kilka schematów przechowywania macierzy rzadkich, a wybór konkretnego zależy często od struktury macierzy, rozmieszczenia niezerowych elementów w macierzy a także od algorytmu, w którym dana macierz jest używana. Jednak główne formaty, które wykorzystywane są w zadaniach to:

- COO (ang. *coordinate format* – format współrzędnych) jest to format, w którym w celu zapisu macierzy używane są trzy tablice: ***row***, ***col*** oraz ***data***. Tablice te przechowują odpowiednio indeks wiersza, kolumny oraz dane niezerowych elementów macierzy. Format ten uznawany jest za ogólny sposób przechowywania macierzy rzadkich, z uwagi na fakt, iż pamięć wymagana do przechowywania danych i indeksów wiersza oraz kolumny w tym

formacie jest stale proporcjonalna do liczby niezerowych elementów obecnych w macierzy. Ponadto w przeciwieństwie do innych istniejących formatów, format COO przechowuje indeks wierszy jak również indeks kolumn.

- CSR (ang. *compressed sparse row format* – skompresowany wierszowo format macierzy rzadkich) jest najbardziej popularnym formatem zapisu macierzy rzadkiej. Format ten jawnie przechowuje indeksy kolumn i wartości niezerowe w tablicach: **col_ind** i **data**. Ponadto wykorzystuje trzecią tablicę **ptr**, która zawiera początkowy indeks w tablicach **coli_ind** i **data** dla każdego wiersza w macierzy rzadkiej (wskaźnik wierszy). Dla macierzy MxN tablica **ptr** ma rozmiar M+1, gdzie zapisywany jest indeks początkowego elementu i-tego wiersza w **ptr[i]**. Z tego powodu ostatnią wartością w tablicy **ptr** jest całkowita liczba niezerowych elementów. Format CSR uważa się za naturalne rozszerzenie formatu COO utworzonego przy użyciu tablic, z których jedna (odpowiadająca indeksom wierszy dla wyrazów macierzy) występuje w pomniejszonym rozmiarze. W ten sposób format CSR może zminimalizować zapotrzebowanie na pamięć tak operacyjną podczas obliczeń, jak i na masową do trwałego przechowywania danych. Dodatkowo wprowadzona tablica **ptr** ułatwia szybkie odczytanie wartości macierzy, a także innych ważnych wartości, takich jak na przykład niezerowa liczba elementów w poszczególnym rzędzie.
- CSC (ang. *compressed sparse column format* – skompresowany kolumnowo format macierzy rzadkich) jest to mniej powszechny format niż CSR. W tym formacie niezerowe wartości występujące w macierzy przechowywane są kolumnami. Indeksy wierszy i wartości niezerowe w sposób jawny przechowywane są w tablicach: **row_ind** i **dane**, a początkowy indeks każdej kolumny przechowywany jest w tablicy **col_ptr** (wskaźnik kolumn). Rozmiar tablicy **col_ptr** wyliczany jest w ten sam sposób jak w CSR, czyli dla macierzy MxN przyjmuje wartość N+1. Z tego powodu ostatnią wartością w tablicy **col_ptr** jest całkowita liczba niezerowych elementów. Format ten, w przypadku niektórych operacji arytmetycznych na macierzach, prowadzi do wyższej wydajności niż format CSR.
- ELL (ELLPACK) jest to format, który stworzony został, by zwiększyć wydajność obliczeń mnożenia macierz-wektor na procesorach graficznych GPU (ang. *Graphics Processing Units*). W momencie gdy dostępna jest macierz o rozmiarach MxN z ilością maksymalnie R wartości niezerowych na wiersz, format ELL tworzy macierz danych w tablicy o wymiarach MxR, w której wiersze posiadające mniej niż R elementów dopełniane są do rozmiaru R zerami. Pozostałe tablice opisywanej struktury danych przechowują indeksy kolumn i również są uzupełniane zerami w taki sam sposób, jak dane. Ponieważ format ELLPACK

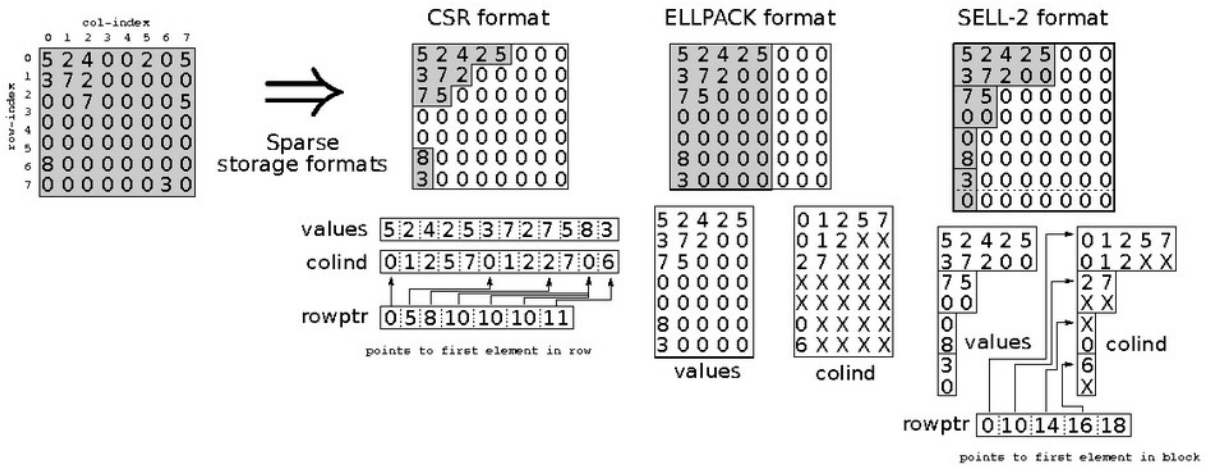
dopełnia wszystkie wiersze do rozmiaru R zerami, może powodować istotne powiększenie ilości zajmowanego miejsca przechowywania danych w stosunku do formatów CSR i CSC. W celu zmniejszenia tego parametru zaproponowano wariant o nazwie Sliced ELPACK.

- SELL (Sliced ELLPACK) jest to format, który jest rozszerzeniem formatu ELL stworzonym przez rozdzielenie wejściowej macierzy na mniejsze bloki wierszy (paski ang. *slices*) o wysokości C. Każdy stworzony blok przechowywany jest w formacie ELL (kolumnowo), gdzie liczba przechowywanych w każdym wierszu niezerowych wartości może się różnić między blokami. Ponadto, w celu dopasowania długości pozostałych wierszy do najdłuższego wiersza w bloku, wiersze są wypełnione zerami. Pozwala to znacznie zmniejszyć rozmiar w porównaniu do formatu ELLPACK. W sytuacji gdy wszystkie rzędy w bloku nie posiadają jednakowej długości, może nadal wystąpić istotne zwiększenie objętości danych w stosunku do np. formatu CSR. Do opisu formatu SELL wykorzystywane są trzy tablice: **val**, **colind** oraz **rowptr** (również znany jako **slice_start**). Wartości, które występują w tablicy **colind** w miejscach, które zostały dopełnione zerami nie przedstawiają żadnych wartości. Tablica **slice_start** działa w identyczny sposób co tablica **ptr** w formacie CSR, jednakże w przypadku SELL uwzględniane są wartości zerowe występujące jako dopełnienia do długości wiersza. Dzięki temu ostatni element tej tablicy nie ukazuje całkowitej liczby niezerowych elementów, lecz całkowitą liczbę elementów występujących w tablicy **val**. W celu przedstawienia algorytmu można również stosować tablicę pomocniczą **cl**, która przedstawia długość bloków wierszy C.

W celu dodatkowego rozszerzenia formatu Sliced ELLPACK, zaproponowano format nazywany „SELL-C-σ”. Format ten został sparametryzowany poprzez dwa parametry: wysokość paska C i zakres sortowania wierszy σ. Sortowanie wierszy macierzy według liczby niezerowych wyrazów w wierszu, w grupach o rozmiarze σ, przeprowadzane jest przed zapisem tak, aby różnice w liczbie wyrazów niezerowych dla wierszy w pojedynczym pasku były jak najmniejsze. Poprzez dobór parametrów C i σ uzyskać można sposób sortowania wierszy macierzy zapewniający wysoką wydajność dla różnorodnych macierzy na wszystkich platformach sprzętowych. Zakres sortowania σ zazwyczaj wybierany jest jako wielokrotność kawałka C. Dodatkowo liczba wierszy macierzy dopełniona musi być do wielokrotności C (również tutaj stosowane jest dopełnienie wartościami zerowymi). Należy również pamiętać, że przy wyborze parametrów sortowania wierszy według liczby niezerowych elementów w ograniczonym obszarze sortowania σ wierszy, zmniejsza się rozmiar schematu i poprawia się wydajność na wszystkich architekturach tylko, jeśli parametr σ nie jest za duży. Ze względu jednak na narzut związany z sortowaniem wierszy macierzy format

„SELL-C-σ” może być wolniejszy od formatu SELL, przez co nie był on wykorzystywany w pracy[1].

Na przykładzie macierzy o wymiarach 8x8 posiadającej 12 niezerowych elementów przedstawione zostało porównanie wymienionych powyżej formatów CSR, ELL, SELL dla C=2. Porównanie to zobrazowano na ilustracji nr 1:



Ilustracja 1: Porównanie formatów przechowywania macierzy rzadkiej

Źródło: [2]

2.4 Mnożenie macierz-wektor dla macierzy rzadkich

2.4.1 Algorytmy mnożenia macierz-wektor dla wybranych formatów przechowywania

Funkcja mnożenia macierz-wektor dla formatu CSR została zrealizowana w sposób sekwencyjny standardowy oraz w sposób zrównoleglony, a dla formatu SELL w sposób sekwencyjny standardowy, zrównoleglony oraz wektorowy sekwencyjny i w sposób wektorowy zrównoleglony. Mnożenie macierz-wektor dla formatu CSR w postaci pseudokodów może być wykonane następująco:

- Mnożenie macierz-wektor przedstawione dla formatu CSR

```
for i = 1, n
  y(i) = 0
  for j = row_ptr(i), row_ptr(i+1) - 1
    y(i) = y(i) + val(j) * x(col_ind(j))
  end;
end;
```

Ilustracja 2: Mnożenie macierz-wektor przedstawione dla formatu CSR

Źródło: Opracowanie własne na podstawie [2]

- Mnożenie macierz-wektor przedstawione dla formatu SELL

```
for i = 1, n/C
  for j = 0, cl[i]
    for k = 0, C
      p = slice_start(i) + j * C + k
      x0 = col_ind(p)
      y(i*C+k) += val(p) * x(x0)
    end;
  end;
end;
```

Ilustracja 3 Mnożenie macierz-wektor przedstawione dla formatu SELL

Źródło: Opracowanie własne na podstawie [2]

- *Mnożenie macierz-wektor przedstawione dla wektorowego formatu SELL*

```
for c=0, c < sell.nr_slice
  tmp = _mm256_load_pd(&y[c<<2])
  offs = sell.sliceStart[c]
  for j =0, j < sell.cl[c]
    val = _mm256_load_pd(&sell.val[offs])
    rhstmp = _mm_loadl_pd(rhstmp ,&x[sell.slice_col[offs ++]])
    rhstmp = _mm_loadh_pd(rhstmp ,&x[sell.slice_col[offs ++]])
    rhs = _mm256_insertf128_pd(rhs , rhstmp ,0)
    rhstmp = _mm_loadl_pd(rhstmp ,&x[sell.slice_col[offs ++]])
    rhstmp = _mm_loadh_pd(rhstmp ,&x[sell.slice_col[offs ++]])
    rhs = _mm256_insertf128_pd(rhs , rhstmp ,1)
    tmp = _mm256_add_pd(tmp , _mm256_mul_pd(val , rhs ))
  end;
  _mm256_store_pd(&y[c<<2] , tmp )
end;
```

Ilustracja 4 Mnożenie macierz-wektor przedstawione dla wektorowego formatu SELL

Źródło: Opracowanie własne

Szczegółowy opis algorytmów znajduje się w dalszej części pracy.

3. Współczesne mikroprocesory wielordzeniowe i systemy wieloprocesorowe

Opis odnosi się do podstawowych cech architektury pojedynczego rdzenia obliczeniowego, który odpowiada klasycznej jednostce przetwarzania w jednym wątku obliczeniowym. Taka jednostka tradycyjnie jest określana jako procesor (*CPU – ang. central processing unit*), stąd również używane są zamiennie nazwy rdzeń i procesor, a czasem również procesor logiczny. We współczesnych układach scalonych, które realizują obliczenia i często przybierają postać mikroprocesorów wielordzeniowych, używa się skrótu procesor.

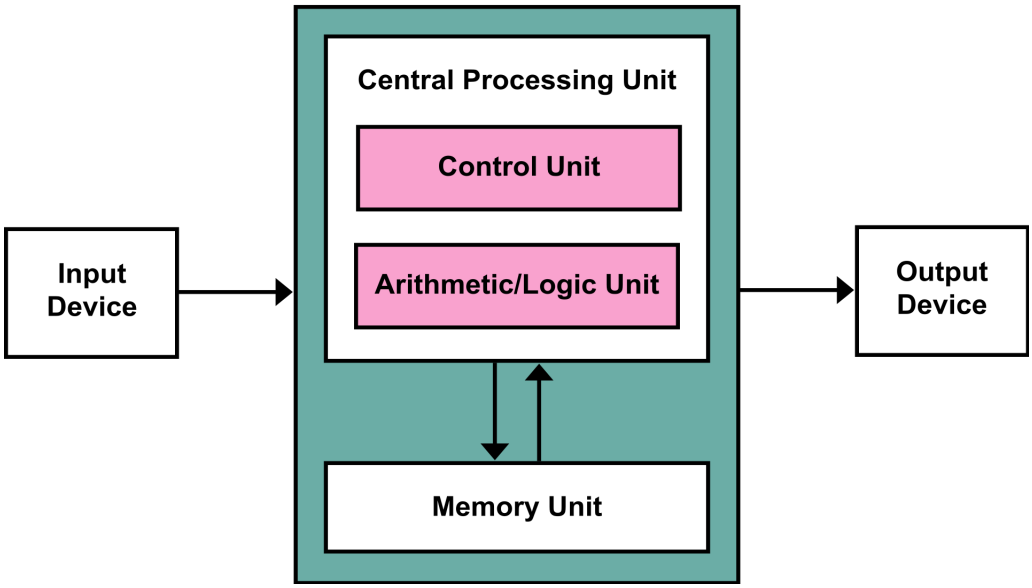
3.1 Budowa i funkcjonowanie pojedynczego rdzenia mikroprocesora

Procesor, jako jednostka centralna komputera, odpowiedzialny jest za wykonywanie instrukcji programu. Podstawowym modelem mikroarchitektury pojedynczego rdzenia jest model architektury von Neumanna.

Maszyna von Neumanna, będąca podstawowym modelem obliczeniowym, składa się z jednostki centralnej (CPU, *ang. Central Processing Unit*), połączonej przy użyciu kanałów komunikacyjnych z układem pamięci głównej (*ang. main memory*). Jednostkę centralną można również przedstawić poprzez dwie pomniejsze jednostki:

- Jednostka arytmetyczno-logiczna ALU (*ang. Arithmetic Logic Unit*) - jest to jednostka odpowiedzialna za wykonywanie operacji arytmetycznych (dodawanie, odejmowanie, mnożenie, dzielenie, inkrementacja, dekrementacja), operacji logicznych (AND, OR, NOT) oraz operacji porównania (cmp ang. comparison).
- Jednostka sterująca (*ang. Control Unit*) - odpowiada za zarządzanie sekwencją operacji, dekodowanie instrukcji oraz sterowanie innymi jednostkami mikroprocesora, m.in. poprzez operacje skoków (bezwarunkowe, warunkowe)[3].

Schemat architektury von Neumanna przedstawiony został na ilustracji nr 5:



Ilustracja 5: Schemat architektury von Neumanna

Źródło: Wikipedia

3.1.1 Przetwarzanie rozkazów

Program w pamięci głównej składa się z ciągu przechowywanych rozkazów (*ang. instructions*). Procesor wykonuje program poprzez pobieranie kolejnych rozkazów do wykonania. Warto zaznaczyć, że pojęcie kolejności w kontekście wykonania nie zawsze odnosi się do kolejności przechowywania rozkazów w pamięci. Procesor pobiera i przetwarza kolejne instrukcje, co stanowi podstawowy mechanizm działania podczas wykonywania programu. Przetwarzanie pojedynczego rozkazu można podzielić na etapy:

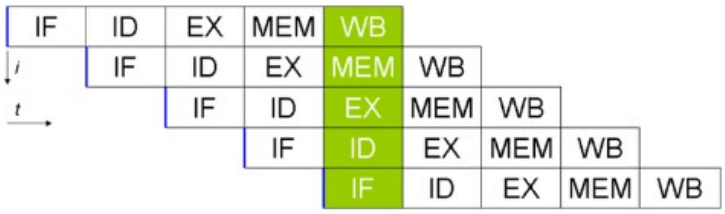
- IF – pobranie rozkazu (*ang. instruction fetch*),
- ID – dekodowanie rozkazu (*ang. instruction decode*),
- EX – wykonanie operacji składających się na rozkaz (*ang. instruction execute*),
- MEM – dostęp do pamięci (*ang. memory access*)
- WB – zapis wyniku realizacji rozkazu (*ang. write-back*)

Poniżej przedstawiony został schemat klasycznego przetwarzania sekwencyjnego:

Ilustracja 6: Schemat klasycznego przetwarzania sekwencyjnego

Źródło: [3]

W celu uzyskania lepszej wydajności programu we współczesnych procesorach wykonywane jest przetwarzanie potokowe. Co istotne w przetwarzaniu potokowym rozpoczęcie wykonania pierwszego etapu rozkazu może nastąpić dopiero po zakończeniu wykonania pierwszego etapu poprzedniego rozkazu przetwarzanego przez procesor. To wszystko może się realizować dzięki temu, że procesor posiada kilka układów, w których instrukcje mogą wykonywać się współbieżnie. Poniżej przedstawiony został schemat klasycznego przetwarzania potokowego:



Ilustracja 7: Schemat klasycznego przetwarzania potokowego

Źródło: [3]

Porównując schematy z ilustracji nr 6 oraz nr 7 można zauważyć, że czas wykonania przetwarzania potokowego, w stosunku do czasu uzyskanego przez przetwarzanie sekwencyjne jest krótszy. Przyjmując, że każdy etap przetwarzania zajmuje ten sam odcinek czasu **t,** to całkowity czas przetwarzania sekwencyjnego można opisać poprzez **liczba_rozkazów * liczba_etapów * t**. W przetwarzaniu potokowym wykonanie pierwszego rozkazu zajmuje: **(liczba_etapów – 1) * t + t**, a każdego kolejnego dodatkowe **t**. Dzięki temu całkowity czas wykonania przetwarzania potokowego zajmuje **(liczba_etapów – 1) * t + liczba_rozkazów * t** co można przedstawić jako wzór: **(liczba_rozkazów + liczba_etapów – 1) * t.** Dzięki temu można zauważyć, że skrócenie czasu przetwarzania potokowego zależy od liczy etapów i liczby występujących rozkazów oraz jest krótszy, od przetwarzania sekwencyjnego, tyle razy ile wykonywanych jest etapów przetwarzania

rozkazu (dla odpowiednio długiej sekwencji rozkazów, liczba_rozkazów >> liczba_etapów).

Kolejne modyfikacje klasycznego przetwarzania potokowego, przetwarzanie SIMD (*ang. Single Instruction Multiple Data*), pozwala na wykorzystywanie również rejestrów wektorowych służących do przechowywania kilku egzemplarzy liczb określonego typu. Związane z tym zestawy rozkazów, będące na liście rozkazów procesora i operujące na danych rejestrach, posiadają specjalne nazwy (np. MMX – *ang. Multimedia Extensions*) odwołujące się do pojęć związanych z renderowaniem grafiki będącym pierwotnym zastosowaniem tych rozkazów.

Rozkazy wektorowe wykonywane są na argumentach będących rejestrami wektorowymi przy wykorzystaniu specjalnych wektorowych potoków przetwarzania. Występowanie tych potoków przetwarzania oznacza istnienie kilku w pełni zsynchronizowanych pojedynczych potoków przetwarzania skalarnego nazywanych ścieżkami SIMD (*ang. SIMD lane*), których kolejność i liczba zależy głównie od typu rejestrów oraz zestawu rozkazów. Wykonanie operacji skalarnej jest natomiast związane z istnieniem tylko jednej ścieżki SIMD. Jednakże pod względem wydajnościowym, wykorzystanie w kodzie rozkazów wektorowych może oznaczać nawet kilkunastokrotne zwiększenie wydajności w stosunku do maksymalnej oferowanej przez procesor.

Współczesne rdzenie procesorów są zbudowane z zaawansowanych elementów, które zwiększają ich wydajność i zdolność do przetwarzania złożonych operacji. Jednym z kluczowych aspektów jest oddzielenie części pobierania, dekodowania i wstępnego przetwarzania rozkazów, znanej jako front-end. Front-end zawiera układy do przewidywania skoków (*ang. branch prediction*), które minimalizują opóźnienia związane z rozgałęzieniami w kodzie, oraz mechanizmy wykonywania poza kolejnością (*ang. out-of-order execution)*, które optymalizują kolejność przetwarzania instrukcji na podstawie dostępnych zasobów.

Procesory te posiadają wiele potoków przetwarzania, które mogą być dedykowane do specyficznych typów rozkazów, w tym rozkazów wektorowych wykorzystywanych w operacjach SIMD. Dzięki temu możliwa jest superskalarność, czyli równoczesne przetwarzanie wielu instrukcji nie tylko w ramach jednego potoku, ale także w wielu potokach jednocześnie.

3.1.2 Pamięci podręczne w rdzeniu

W celu przyspieszenia dostępu do pamięci głównej (RAM, ang. *random access memory*) w architekturze komputerowej stosuje się pamięć podręczną (*ang. cache memory*), która podzielona jest na linie, a rozmiar pojedynczej linii może się różnić w zależności od konkretnego systemu.

Najczęściej spotykane linie posiadają rozmiar 64 lub 128 bajtów. Każda linia może pomieścić kilka zmiennych podwójnej precyzji lub kilkanaście zmiennych całkowitych pojedynczej precyzji.

Pamięć fizyczna (DRAM) podzielona jest na bloki, każdy o rozmiarze odpowiadającym rozmiarowi linii pamięci podręcznej. W najprostszym modelu, pamięci odwzorowanej bezpośrednio (*ang. direct mapped*), każdy blok pamięci fizycznej jest przyporządkowany do pojedynczej linii pamięci podręcznej, a każdy kolejny następujący po nim blok jest przyporządkowany kolejnej linii. Całkowity rozmiar pamięci podręcznej jest znacznie mniejszy od rozmiaru pamięci fizycznej, stąd w pewnym momencie kolejny przyporządkowywany blok zostanie przyporządkowany ponownie pierwszej linii. W końcowym efekcie, pojedynczej linii pamięci podręcznej odpowiadać będzie wiele bloków pamięci fizycznej. We współczesnych mikroprocesorach występują bardziej złożone pamięci sekcyjno-skojarzeniowe (ang. *set associative),* w których pojedynczy blok pamięci DRAM może znajdować się w jednej z kilku linii tworzących sekcję (zbiór).

Mechanizm działania pamięci podręcznej można przedstawić następująco. Procesor, po wygenerowaniu adresu zmiennej, do której chce uzyskać dostęp, sprawdza czy odpowiadająca zmienna znajduje się w pamięci podręcznej, czyli czy aktualna kopia bloku pamięci fizycznej zawierającego wygenerowany adres znajduje się w odpowiadającej mu linii pamięci podręcznej. Jeżeli aktualna zmienna została znaleziona (*ang. cache hit*), procesor realizuje szybki dostęp do zmiennej. Jeżeli natomiast blok zawierający zmienną nie znajduje się w pamięci podręcznej (*ang. cache miss*), zawartość całej linii jest podmieniana, blok pamięci fizycznej jest kopiowany do linii pamięci podręcznej, po czym następuje realizacja dostępu do pamięci podręcznej. Załadowany w ten sposób blok pamięci pozostaje w pamięci podręcznej, dopóki nie zostanie podmieniony przez inny blok odwzorowany w tę samą linię.

Podany wyżej schemat działania przedstawiony został na przykładzie jednopoziomowej pamięci podręcznej. Pamięć podręczna może jednak posiadać wiele poziomów, gdzie najczęściej spotykany jest podział na trzy poziomy:

- L1 o typowym rozmiarze 32 kB,
- L2 o typowym rozmiarze kilka razy większym niż L1, np. 256 kB,
- L3 o rozmiarze od kilku, aż do kilkudziesięciu MB.

Schemat działania przedstawiony dla trzypoziomowej pamięci podręcznej wygląda następująco. Jeżeli przy sprawdzaniu w pamięci L1 nie zostanie znaleziony blok zawierający zmienną (cache miss) poszukiwania przeprowadzane są na niższym poziomie (poziom L2). Jeżeli również na tym

poziomie nie zostanie znaleziony odpowiedni blok sprawdzany jest poziom L3, a następnie, jeżeli nie istnieje więcej poziomów pamięci podręcznej, sprawdzana jest pamięć fizyczna i zawartość całej linii jest podmieniana.

3.2 Mikroprocesory wielordzeniowe

Współczesne mikroprocesory często posiadają wielordzeniową architekturę, co zwiększa ich zdolność do równoczesnego wykonywania wielu instrukcji. W takich architekturach jeden fizyczny procesor zawiera kilka rdzeni wykonawczych, które mogą równocześnie przetwarzać instrukcje. To pozwala na zwiększenie wydajności i lepsze wykorzystanie zasobów sprzętowych.

Początkowo mikroprocesory wielordzeniowe wykorzystywano do przetwarzania sygnałów, na przykład w zastosowaniach związanych z grafiką komputerową. Jednak z czasem zaczęto wykorzystywać wielordzeniowe architektury również w obszarach, takich jak obliczenia komputerowe czy zastosowania serwerowe. W przypadku współczesnych wielordzeniowych mikroprocesorów, pamięć L1 i L2 zazwyczaj jest prywatna dla każdego rdzenia, co pozwala na szybki dostęp do danych przez dany rdzeń. Pamięć L3 natomiast jest wspólna dla wszystkich rdzeni, dzięki czemu ułatwione jest współdzielenie danych pomiędzy nimi. Połączone ze sobą procesory za pomocą odpowiednich układów wewnętrznych posiadają również wspólny kontroler pamięci, zarządzający dostępem do pamięci dla wszystkich rdzeni[3].

3.3 Systemy wieloprocesorowe z pamięcią wspólną

Systemy wieloprocesorowe jak również mikroprocesory wielordzeniowe wykorzystują wielowątkowość. Wielowątkowość dotyczy wielu wątków zarządzanych przez pojedynczą instancję systemu operacyjnego i mających dostęp do współdzielonej przestrzeni adresowej (*ang. shared memory*). Jednym ze sposobów implementacji wielowątkowości jest hyperthreading. Jest to technologia, która pozwala jednemu fizycznemu rdzeniowi procesora symulować dwa procesory logiczne używając lokalnych zasobów. Chociaż hyperthreading może poprawić wydajność w niektórych zastosowaniach, to dla lepiej zoptymalizowanych programów lepiej tej funkcjonalności nie wykorzystywać, ponieważ przypisane przez system operacyjny dwa wątki do danego rdzenia widziane będą przez system jako dwa procesory logiczne, co może prowadzić do spadku wydajności. Po wyłączeniu hyperthreadingu, system operacyjny będzie widział jedynie tyle procesorów logicznych ile jest rdzeni.

Systemy wieloprocesorowe z pamięcią współdzieloną można podzielić na dwa rodzaje:

- UMA (ang. *uniform memory access*) – system, w którym dostęp do pamięci jest jednolity, a z poziomu każdego rdzenia długość drogi do pamięci jest taka sama.
- NUMA (ang. *non-uniform memory access*) – system, w którym każdy rdzeń ma sobie przydzielone kości pamięci, a gdy chce korzystać z innych, to może to zrobić jedynie za pośrednictwem innych rdzeni.

Standardowe planowanie systemu operacyjnego pozwala przypisać wątki do różnych rdzeni w oparciu o złożony algorytm, który stara się zrównoważyć obciążenie rdzeni, zachowując jednocześnie szybkie przełączanie kontekstu dla każdego rdzenia. Czasami jednak jakaś tymczasowa sytuacja powoduje, że standardowe planowanie systemu może prowadzić do nieoptymalnej wydajności. W takich sytuacjach może się okazać, że występuje potrzeba przypisania wątków do rdzeni samodzielnie.

Przypisanie procesów i wątków do rdzeni (*ang. process affinity*) często decyduje o sposobie wykorzystania pamięci. W Linuksie przypisanie fizycznych ramek DRAM do stron pamięci wirtualnej dla dynamicznie alokowanej pamięci często odbywa się na podstawie pierwszego zapisu do komórki pamięci (*ang. lazy page allocation*).

4 Obliczenia równoległe i wektoryzacja kodu

4.1 Idea obliczeń równoległych

Obliczenia równoległe uznawane są za formę wykonywania obliczeń, dzięki której wiele instrukcji wykonywanych jest równocześnie, ewentualnie jedna instrukcja jest wykonywana w danym momencie na wielu egzemplarzach danych. W pierwszym przypadku obliczenia wykonywane są najczęściej bez synchronizacji sprzętowej, na maszynach, zgodnie z klasyfikacją Flynna, typu MIMD (*ang. Multiple Instruction Multiple Data*)[4]. W drugim przypadku mamy do czynienia z przetwarzaniem wektorowym realizowanym współcześnie przez układy, odpowiadające architekturze SIMD (*ang. Single Instruction Multiple Data*) w klasyfikacji Flynna. Układami takim są np. potoki przetwarzania wektorowego współczesnych mikroprocesorów. W dalszej części obliczeniami równoległymi nazywane będą tylko obliczenia na maszynach MIMD, pozostawiając dla przetwarzania przez układy SIMD termin wektoryzacja.

W celu rozwiązania danego problemu w sposób równoległy należy zaprojektować algorytm określający jakie operacje mogą być wykonywane na danych, sposób gromadzenia, przechowywania oraz przetwarzania danych i wyników. Z tych względów rozróżnia się dwa podstawowe typy obliczeń równoległych: z pamięcią wspólną oraz z pamięcią rozproszoną.

4.2 Obliczenia równoległe z pamięcią wspólną

Jednym z popularnych modeli obliczeń równoległych z pamięcią wspólną jest programowanie wielowątkowe. Wielowątkowość odnosi się do realizacji wielu wątków mających dostęp do wspólnej przestrzeni adresowej, dzięki czemu podczas obliczeń równoległych zmienne mają dostęp do tych samych danych. W pracy do obliczeń równoległych wykorzystana została specyfikacja OpenMP.

4.2.1 Specyfikacja OpenMP

Specyfikacja OpenMP obsługuje wieloplatformowe programowanie równoległe z pamięcią współdzieloną. Specyfikacja ta definiuje przenośny, skalowalny model z elastycznym i prostym interfejsem do tworzenia aplikacji równoległych na używanych architekturach procesora[5]. W modelu programowania OpenMP podstawowym elementem są dyrektywy kompilatora, a standard definiuje również szereg funkcji wspomagających realizacje równoległą obliczeń.

W specyfikacji tej używany jest format dyrektyw w języku C „**#pragma omp nazwa_dyrektywy lista_klauzul**”, po którym występuje znak nowej linii (standard osobno definiuje składnię dla języka FORTRAN). Główną dyrektywą jest „**#pragma omp parallel**”, która wyznacza początek wykonania równoległego kodu oraz w sposób jawny nakazuje zrównoleglenie wybranego bloku kodu. Najważniejszymi z istniejących dyrektyw są dyrektywy podziału pracy (ang. *work sharing constructs*), które występują w obszarze równoległym i stosowane są do rozdzielania poleceń realizowanych przez poszczególne wątki. Oprócz opisanej powyżej głównej dyrektywy „**#pragma omp parallel**” w pracy występują również dyrektywy:

- „**#pragma omp for**” - dyrektywa ta nakazuje kompilatorowi podzielenie iteracji pętli for między wątki. W ten sposób pętla staje się pętlą równoległą. Dyrektywa ta musi być napisana tuż przed implementacją pętli for.
- „**#pragma omp critical**” - dyrektywa ta nakazuje kompilatorowi realizację sekcji krytycznej, poprzez zagwarantowanie wykonania danej części kodu przez tylko jeden wątek naraz.

Dodatkowo każda z użytych dyrektyw może posiadać własny zestaw dopuszczalnych klauzul, a najważniejsze z nich określają sposób traktowania zmiennych poprzez wątki w obszarze równoległym. W pracy używane jest pięć klauzul:

- „**num_threads**” - klauzula dyrektywy **parallel** umożliwia jawne określenie liczby wątków wykonujących kod w obszarze obowiązywania dyrektywy,
- „**schedule(sposób_rozdziału_iteracji, *wartość*)**” - klauzula **schedule** dyrektywy **for** umożliwia opisanie sposobu rozdziału iteracji pętli równoległej między wątki oraz liczbę iteracji (wartość rozmiaru porcji iteracji) przydzielanych jednorazowo pojedynczemu wątkowi (w programie podział iteracji zawsze odbywa się w sposób cykliczny, tzw. *round robin*). Przy opisie sposobu rozdziału można wyróżnić trzy najczęściej używane:
  - **static** - iteracje podzielone są na mniejsze zbiory o rozmiarze ***wartość***, a następnie kolejno przydzielane dostępnym wątkom,
  - **dynamic** - każdy wątek posiada przypisaną liczbę iteracji określoną przez parametr ***wartość***, a po wykonaniu obliczeń otrzymuje kolejną porcję iteracji do wykonania, przydzielaną dynamicznie przez układ zarządzający wykonaniem równoległym,
  - **runtime** - w tym przypadku jedna z powyższych opcji oraz parametr ***wartość*** jest ustalana poprzez nadanie odpowiedniej wartości specjalnej zmiennej środowiskowej,

- „**default(none)**”- oznacza deklarację, że domyślnie żadna zmienna nie będzie ani wspólna, ani prywatna. Typ używanej zmiennej określa sam programista,
- „**private**” - klauzula współdzielenia zmiennych umożliwia tworzenie zmiennej lokalnej wątków,
- „**firstprivate**” - klauzula współdzielenia zmiennych umożliwia utworzenie zmiennej lokalnej wątków z kopiowaną wartością początkową z obszaru kodu przed dyrektywą[6].

4.2.2 Wyścig i zależności danych

W programowaniu równoległym z pamięcią wspólną mogą pojawić się problemy ze współbieżnością. Głównym problemem jest wyścig (ang. *race condition*), który występuje w momencie, gdy wątki współbieżnie korzystają z zasobu współdzielonego. W przypadku braku synchronizacji pracy wątków przy dostępie do zasobu, którym może być np. obszar pamięci, wyścig powoduje brak deterministycznego wykonywania programu. W celu zapobiegania wyścigom do kodu programu wprowadza się sekcje krytyczne i wzajemne wykluczanie. W programie sekcja krytyczna przedstawia fragment kodu, w którym używany jest zasób dzielony – w danej chwili zasób powinien być wykorzystywany tylko przez jeden wątek. Do bezpiecznego wykonania wprowadzonej sekcji krytycznej można wykorzystać wzajemne wykluczanie, które realizowane jest w postaci protokołów wejścia do i wyjścia z sekcji krytycznej wykonywanych przez procesy lub wątki. Inną używaną w programowaniu techniką synchronizacji eliminującą występowanie wyścigów może być zastosowanie tzw. operacji atomowych (niepodzielnych). Należy jednak pamiętać, że użycie synchronizacji może doprowadzić do różnych problemów w tym zakleszczenia (w przypadku zakleszczenia wątki czekają na siebie wzajemnie co prowadzi do niemożności kontynuowania działania) lub zagłodzenia wątku (w przypadku zagłodzenia pojedynczy wątek nie może zakończyć działania, mimo że inne wątki nadal pracują) [7].

Synchronizacja pracy wątków stanowi także narzut spowalniający wykonywanie programów. Z tego względu, przy projektowaniu algorytmów, dąży się do eliminacji konieczności stosowania synchronizacji.

4.3 Programowanie systemów z pamięcią rozproszoną – specyfikacja MPI

4.3.1 Idea programowania z przesyłaniem komunikatów

MPI (ang. Message-Passing Interface) jest to specyfikacja interfejsu biblioteki do przekazywania komunikatów, która dotyczy przede wszystkim modeli programowania równoległego dla systemów z pamięcią rozproszoną. Standardowy mechanizm przesyłania komunikatów jest mechanizmem niesymetrycznym, w którym występuje nadawca komunikatu oraz odbiorca. Dane przenoszone są z przestrzeni adresowej jednego procesu do przestrzeni adresowej innego procesu poprzez wspólne operacje na każdym procesie. W sytuacji, w której występuje asymetryczność należy rozdzielić wywołania i w procesie wysyłającym umieścić wywołanie funkcji send(), a w drugim, procesie odbierającym, umieścić wywołanie funkcji receive(). Występujące w specyfikacji funkcjonalności można opisać w następujący sposób:

- wysyłanie komunikatu, wykorzystanie funkcji send(), odbywa się poprzez funkcję: send(cel, identyfikator_komunikatu, dane)
- odbieranie komunikatu, wykorzystanie funkcji receive(), odbywa się poprzez funkcję: receive(źródło, identyfikator_komunikatu, dane)

MPI jako interfejs ustanawia praktyczny, przenośny, wydajny i elastyczny standard przekazywania komunikatów. W środowisku komunikacyjnym z pamięcią rozproszoną, w którym procedury i abstrakcje wyższego poziomu są zbudowane na procedurach przekazywania komunikatów niższego poziomu, korzyści płynące ze standaryzacji są szczególnie widoczne[8].

Podstawowymi funkcjami realizującymi specyfikację MPI są:

- int **MPI_Init**(int *pargc, char ***pargv) – pierwsza funkcja MPI inicjalizująca parametry,
- int **MPI_Comm_size**(MPI_Comm comm, int *psize) - funkcja pozwalająca uzyskać rozmiar komunikatora, określa liczbę procesów tworzących dany komunikator (w przypadku standardowych obliczeń komunikatorem jest obiekt MPI_COMM_WORLD reprezentujący wszystkie procesy wykonujące obliczenia),
- int **MPI_Comm_rank**(MPI_Comm comm, int *prank) - funkcja pozwalająca uzyskać rangę procesu (identyfikator procesu) w ramach komunikatora w MPI

  (0 <= *prank < *psize),
- int **MPI_Finalize**(void) - funkcja pozwalająca wyczyścić struktury danych[9].

4.3.2 Komunikacja punkt-punkt (ang. point-to-point)

W kontekście użytkownika MPI oferuje kilka ważnych gwarancji dotyczących przesyłania komunikatów między procesami. Pierwszą gwarancją, o której należy wspomnieć jest gwarancja postępu przy realizacji przesyłania danych. Po poprawnym zainicjowaniu pary send-receive, co najmniej jedna z tych operacji zostanie zakończona. To oznacza pewność, że komunikaty zostaną dostarczone, pod warunkiem prawidłowego użycia operacji send() i receive(). MPI zapewnia również pewność co do zachowania porządku przyjmowania komunikatów. Komunikaty z tego samego źródła, o tym samym identyfikatorze i w ramach tego samego komunikatora, będą przyjmowane w kolejności, w jakiej zostały wysłane. Należy pamiętać, aby zapewnić spójność w komunikacji między procesami. Warto jednak zauważyć, że MPI nie gwarantuje uczciwości przy odbieraniu komunikatów z różnych źródeł oraz nie gwarantuje konkretnej kolejności odbierania komunikatów od różnych nadawców. Dodatkowo, w trakcie realizacji procedur przesyłania danych może wystąpić błąd związany z przekroczeniem limitów dostępnych zasobów systemowych (np. rozmiaru bufora przechowującego przesyłane dane). Jest to istotne, ponieważ użytkownik powinien być świadomy możliwości wystąpienia tego typu problemów i odpowiednio zarządzać zasobami systemowymi.

Podstawowymi funkcjami realizującymi komunikację punkt-punkt są:

- int **MPI_Send**(void* buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm) – funkcja ta określa wysyłanie z bufora w pamięci nadawcy, lokalizacja, rozmiar i typ bufora wysyłania są określone przez pierwsze trzy parametry operacji wysyłania, a ostatnie trzy parametry operacji wysyłania określają identyfikator (rangę, zmienna dest) odbiorcy, oznaczenie komunikatu (zmienna tag) oraz komunikator, w ramach którego odbywa się przesyłanie wiadomości.
- int **MPI_Recv**(void *buf, int count, MPI_Datatype dtype, int src, int tag,

  MPI_Comm comm, MPI_Status *stat) – w funkcji tej wiadomość jest odbierana, a dane wiadomości są zapisywane w buforze odbierania. Pierwsze trzy parametry operacji odbioru określają lokalizację, rozmiar i typ bufora odbiorczego, następne trzy parametry służą do wybierania przychodzącej wiadomości (gdzie src oznacza identyfikator nadawcy, tag charakteryzuje komunikat a comm odnosi się do komunikatora), ostatni parametr służy do zwracania informacji (statusu) o właśnie odebranej wiadomości.

4.3.3 Komunikacja grupowa

W MPI możliwe jest wykorzystanie komunikacji grupowej (ang. *collective communication*), która wymaga wspólnego wywołania procedur przez wszystkie procesy w danej grupie (inaczej: w ramach danego komunikatora). Umożliwia to wspólną interakcję między procesami, co jest kluczowym aspektem w programach wykorzystujących efektywne programowanie równoległe. Używane w komunikacji grupowej procedury umożliwiają skuteczną wymianę danych między procesami w ramach tej grupy. Procedury te, zawierające istotne informacje, przyjmują jako argumenty bufory danych, które są wysyłane, a także (jeśli to konieczne) wykorzystując bufory przeznaczone na dane do odebrania. W sytuacji, gdy bufory danych wysyłanych i odbieranych są tożsame (np. w sytuacji gdy proces, który rozprasza dane pozostawia swoją część danych w tym samym miejscu), MPI może wymagać użycia argumentu MPI_IN_PLACE zamiast jednego z buforów. To umożliwia optymalizację i efektywną manipulację danymi na miejscu. Wszystkie procedury komunikacji grupowej są blokujące, co oznacza, że wykonanie danej operacji przez proces oczekuje na zakończenie operacji zanim przejdzie dalej. Warto jednak pamiętać, że zakończenie operacji przez jeden proces nie gwarantuje, że pozostałe procesy również zakończyły swoje operacje.

Przykładami takich operacji są m.in.:

- Bariera – funkcja ta blokuje procesy do czasu, gdy wszystkie procesy w grupie nie wywołają tej funkcji.
  int **MPI_Barrier**(MPI_Comm comm)
- Rozgłaszanie jeden do wszystkich – funkcja ta rozgłasza wiadomość z procesu z rangą root do wszystkich procesów grupy, włączając w to samą siebie. Jest wywoływana przez wszystkich członków grupy przy użyciu tych samych argumentów dla comm i root.
  Po powrocie zawartość bufora root jest kopiowana do wszystkich innych procesów.
  int **MPI_Bcast(**void *buff, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
- Zbieranie wszyscy do jednego - każdy proces (w tym proces główny) wysyła zawartość swojego bufora wysyłania do procesu głównego (o randze root). Proces główny odbiera wiadomości i przechowuje je w kolejności rangi. Wynik jest taki, jakby każdy z n procesów w grupie (w tym proces główny) wykonał wywołanie funkcji MPI_Send(), a proces główny wykonał n wywołań funkcji MPI_Recv()
  int **MPI_Gather**(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount, MPI_Datatype rdtype, int root, MPI_Comm comm)

- Rozpraszanie jeden do wszystkich – MPI_Scatter() jest operacją odwrotną do MPI_Gather(), proces główny (root) wykonuje n operacji wysyłania MPI_Send(), a każdy proces wykonuje operację odbierania MPI_Recv()
  int **MPI_Scatter**(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount, MPI_Datatype rdtype, int root, MPI_Comm comm)
- Zbieranie wszyscy do wszystkich (równoważne rozgłaszaniu wszyscy do wszystkich) – funkcję tą można traktować jako MPI_GATHER dla sytuacji, gdzie wszystkie procesy otrzymują wynik nie tylko proces główny. Blok danych wysłany z i-tego procesu jest odbierany przez każdy proces i umieszczany w i-tym bloku bufora rbuf.
  int **MPI_Allgather**(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount, MPI_Datatype rdtype, MPI_Comm comm)

4.4 Programowanie procesorów z rejestrami wektorowymi – wektoryzacja kodu

Wektoryzacja to proces przekształcania algorytmu z implementacji skalarnej, gdzie operacje są wykonywane na pojedynczych argumentach skalarnych na implementację wektorową, gdzie pojedyncza instrukcja może operować na wektorze zawierającym spakowane wartości skalarne. Instrukcje SIMD (Single Instruction, Multiple Data) pozwalają operować na wielu danych w jednej instrukcji i korzystać z rejestrów zmiennoprzecinkowych o długości 128, 256, a nawet 512 bitów. Dzięki zastosowaniu wektoryzacji zmodyfikowany kod może stać się krótszy i bardziej czytelny, co prowadzi do znacznego zmniejszenia złożoności obliczeniowej oraz czasu wykonania programu, ponieważ spakowane instrukcje operują na więcej niż jednym elemencie danych jednocześnie[10].

4.4.1 Wektoryzacja niejawna poprzez opcje kompilacji

Wektoryzacja może być realizowana w sposób niejawny poprzez użycie opcji kompilatora. Kompilator przeglądając kod źródłowy programu, identyfikuje fragmenty, które mogą być poddane wektoryzacji. Najczęściej są to pętle, ponieważ operacje wewnątrz pętli mogą być łatwo zrównoleglone. Kompilator dokonując oceny, czy dane fragmenty kodu nadają się do wektoryzacji, sprawdza zależności danych, ale również dostępność odpowiednich instrukcji wektorowych na konkretnej architekturze. Istnieją również półautomatyczne opcje wektoryzacji. Jedną z nich jest wykorzystanie specyfikacji OpenMP, która pozwala na wykorzystanie wektoryzacji w kodzie.

Umożliwia ona jawne zlecenie kompilatorowi wektoryzacji konkretnej pętli poprzez dodanie przed pętlą odpowiedniej dyrektywy. Po przeprowadzeniu procesu wektoryzacji, kompilator generuje nowy kod źródłowy, zawierający wektorowe instrukcje, który należy zrekompilować przed uruchomieniem programu. Wektorowy kod jest przenośny między różnymi architekturami, o ile są one zgodne z używanymi instrukcjami wektorowymi[11].

4.4.2 Wektoryzacja jawna – wykorzystanie wstawek wewnętrznych kompilatora

Wektoryzacja może być również realizowana poprzez implementację bezpośrednio w kodzie źródłowym. Jednym ze sposobów manualnej wektoryzacji jest wykorzystanie rozkazów procesora pod postacią tzw. wstawek wewnętrznych kompilatora (ang. *compiler intrinsics*). W pracy umieszczanie rozkazów procesora zrealizowane zostało w języku C poprzez zastosowanie specyfikacji AVX2 (ang. Advanced Vector Extensions 2), która jest

rozszerzeniem architektury x86-64. Użyte przez to zostały 256-bitowe wektory, przechowujące

cztery 64-bitowe wartości zmiennoprzecinkowe podwójnej precyzji, zapisywane przy użyciu

typu __m256d. W celu użycia wstawek wewnętrznych kompilatora należy do kodu źródłowego dołączyć plik nagłówkowy immintrin.h.

W programie w procesie wektoryzacji (użytej dla formatu SELL) zostały wykorzystane poniższe funkcje[12]:

- Funkcja _mm256_load_pd(double const *a), która pobiera wartości zmiennoprzecinkowe podwójnej precyzji z lokalizacji w pamięci „a” do wektora docelowego,
- Funkcja _mm256_insertf128_pd(__m256d a, __m128d b, int offset), która wstawia 128 bitów spakowanych wartości typu float 64 w wektorze „b” do docelowego wektora „a” z bitowym przesunięciem określonym przez wartość zmiennej „offset”,
- Funkcja _mm256_store_pd(double *a, __m256d b), która zapisuje spakowane wartości z wektora „b” do wyrównanej lokalizacji w pamięci wskazanej przez „a”,
- Funkcja _mm_loadh_pd(__m128d a, *const f64 b) oraz _mm_loadl_pd(__m128d a, *const f64 b), funkcje te ładują wartości podwójnej precyzji z „b” do bitów wyższego (_mm_loadh_pd) lub niższego (_mm_loadl_pd) rzędu 128-bitowego wektora „a”,
- Funkcja _mm256_add_pd(__m256d m1, __m256d m2), która sumuje cztery spakowane elementy zmiennoprzecinkowe podwójnej precyzji pierwszego wektora źródłowego „m1”

z czterema elementami float 64 drugiego wektora źródłowego „m2”,

- Funkcja _mm256_mul_pd(__m256d m1, __m256d m2), która mnoży cztery spakowane elementy pierwszego wektora „m1” z czterema elementami drugiego wektora „m2”.

Efektywne wykorzystanie tej techniki wektoryzacji wymaga dostosowania struktury kodu źródłowego, co z kolei może podnieść poziom trudności procesu optymalizacji. Choć metoda ta umożliwia programiście pełniejszą kontrolę nad procesem obliczeń w porównaniu do wektoryzacji przy użyciu kompilatora, to jednak może być także bardziej czasochłonna w implementacji.

5. Wydajność obliczeń

5.1 Miary wydajności obliczeń

5.1.1 Miary dla obliczeń arytmetycznych (zmiennoprzecinkowych)

Wydajność obliczeń arytmetycznych odnosi się do szybkości, z jaką system komputerowy lub procesor wykonuje lub teoretycznie może wykonywać operacje matematyczne. Można wyznaczyć dwie miary wydajności obliczeń:

- Może ona zostać wyrażona poprzez liczbę operacji zmiennoprzecinkowych na sekundę (FLOPS ang. *floating point operations per second*).
- Jednakże do przedstawienia wydajności najczęściej wykorzystywana jest jednostka GFLOPS (GFLOPS - 10^9 FLOPS). Miara ta wykorzystywana jest w aplikacjach, w których dominują operacje zmiennoprzecinkowe.
    - teoretyczna wydajność szczytowa: GFLOP/s = IPC * liczba rdzeni * liczba wykonanych FLOPów * częstotliwość [GHz], gdzie IPC - maksymalna ilość wykonywanych instrukcji na takt zegara (cykl, ang. *instructions per cycle*), a liczba FLOPów odnosi się do pojedynczej instrukcji (np. 2 dla rozkazów FMA, *ang. Fused Multiply Add,* oraz ponad jeden dla rozkazów wektorowych - w zależności od rozmiaru wektora)
    - rzeczywista wydajność: GFLOP/s = liczba wykonanych FLOPów / czas wykonania

Dla mnożenia macierz-wektor obliczanie miary w jednostce GFLOPS zależy od liczby wartości niezerowych używanej macierzy – w standardowej macierzy używana jest liczba wszystkich N^2 elementów, a w macierzy rzadkiej używana jest jedynie liczba niezerowych elementów, NONZ. Jednostka GFLOPS liczona jest w sposób wynikający bezpośrednio z algorytmu:

$$\frac{2\mathrm{NONZ}}{t}$$

gdzie:

t – czas obliczeń algorytmu mnożenia macierz-wektor w nanosekundach [ns]

Do wyrażania wydajności obliczeń równoległych utworzone zostały również miary względne, w których porównuje się czas wykonania programu na jednym oraz wielu procesorach/rdzeniach.

5.1.2 Miary względne wydajności obliczeń równoległych

Jednym z głównych celów optymalizacji jest maksymalizacja nakładania się czasów obliczeń i komunikacji, a także minimalizacja i ukrywanie narzutu systemu. Jednak w celu znalezienia odpowiedniego sposobu optymalizacji zadania należy przeanalizować uzyskiwane czasy wykonania. Rzeczywisty czas wykonania programu zależy od czasów komponentów oraz stopnia nakładania się czasów komponentów. Jest on jednak mniejszy niż suma czasów obliczeń, komunikacji i obciążenia systemowego.

W kontekście obliczeń równoległych, które obejmują także wymianę komunikatów dla systemów z pamięcią rozproszoną, czas obliczeń (T_{comp}) dotyczy wyłącznie operacji wykonywanych przez procesory i dostępu do pamięci. Całkowity czas obliczeń dla wszystkich procesów można porównać z czasem wykonania sekwencyjnego (tj. pojedynczego wątku). W celu ustandaryzowania miar obliczeń równoległych wprowadzono pojęcia równoległego przyspieszania i efektywności zrównoleglenia:

- Przyspieszenie obliczeń - obliczane jest jako stosunek czasu potrzebnego do rozwiązania zadania przy użyciu pojedynczego procesora do czasu potrzebnego do rozwiązania zadania przy użyciu wielu procesorów. Wzór na przyspieszenie obliczeń może zostać przedstawiony jako: S(p) = Ts / Tp(p) (lub Tp(1) / Tp(p)),

gdzie:

p – liczba procesorów,

Ts – czas wykonania algorytmu sekwencyjnego,

Tp(p) - czas wykonania algorytmu równoległego przy użyciu p procesorów.

- Efektywność zrównoleglenia – obliczana jest jako stosunek uzyskanego przyspieszenia do przyspieszenia idealnego, dla którego zakłada się, że jest równe liczbie użytych procesorów. Mierzy ono jak dobrze wykorzystane są dostępne zasoby obliczeniowe. Wzór na efektywność zrównoleglenia może zostać przedstawiony jako: E(p) = S(p) / p[13],

5.1.3 Skalowalność obliczeń równoległych

Pojęcie skalowalności jest kluczowe w kontekście analizy wydajności systemów, zwłaszcza w obliczeniach i komunikacji. Skalowalność jest kluczową właściwością pozwalającą uzyskać

wysoką wydajność obliczeń. Odnosi się ona do zdolności systemu do efektywnego obsługiwania rosnącego obciążenia poprzez zwiększanie zasobów. Może to dotyczyć sprzętu, środowiska wykonawczego (np. platformy programistycznej) lub samego programu. W kontekście wydajności, skalowalność jest często rozumiana jako zachowanie systemu podczas zwiększania obciążenia i dostarczania oczekiwanej wydajności w miarę wzrostu zasobów. W idealnej sytuacji skalowalny system powinien być w stanie efektywnie wykorzystać dostępne zasoby, a jego wydajność powinna rosnąć proporcjonalnie do zwiększającego się obciążenia. Program dąży do idealnej skalowalności wraz ze wzrostem przyspieszenia obliczeń zmierzającym do nieskończoności.

Wyodrębnić można dwa rodzaje skalowalności: skalowalność w sensie silnym oraz skalowalność w sensie słabym.

- Silna skalowalność mierzy wydajność dla rosnącej liczby zasobów wykorzystywanych w obliczeniach (procesory, węzły obliczeniowe). Program skalowalny w sensie silnym wykazuje liniowe względne przyspieszenie obliczeń dla zadań o stałym rozmiarze. Oznacza to, że całkowity narzut obliczeń równoległych pozostaje stały, a więc narzut na pojedynczy proces maleje. Taka sytuacja w praktyce zdarza się bardzo rzadko. Programy takie charakteryzują się brakiem części sekwencyjnej oraz komunikacji i synchronizacji. Przykładem takiego algorytmu jest algorytm mnożenia wektora przez pojedynczą liczbę, skalar.
- Słaba skalowalność oznacza skalowalność w przypadku, gdy całkowite obciążenie nie jest stałe, jak w przypadku skalowalności w sensie silnym, ale obciążenie przypadające na pojedynczy proces jest stałe. Stałe obciążenie na wątek/proces oznacza natomiast całkowite obciążenie rosnące proporcjonalnie do liczby wątków/procesów.

Słabą skalowalność można przypisać wielu algorytmom, natomiast uzyskanie liniowej silnej skalowalności jest niezwykle rzadkie.

5.2 Wydajność obliczeń a kompilatory optymalizujące

5.2.1 Optymalizacja klasyczna

Optymalizacja klasyczna skupia się głównie na poprawie wydajności pojedynczego wątku. Poprzez minimalizowanie ilości wykonywanych operacji możliwe jest skrócenie czasu wykonania programu, a optymalne wykorzystanie zdolności sprzętu do operacji na wektorach może znacznie przyspieszyć przetwarzanie danych. Należy również pamiętać, że właściwe wykorzystanie

hierarchii pamięci oraz usuwanie zależności między instrukcjami pozwala na zminimalizowanie opóźnienia i umożliwia lepsze wykorzystanie jednostek wykonawczych. Klasyczne techniki optymalizacji mogą być implementowane ręcznie, ale większość z nich jest również wykorzystywana przez kompilatory. Istotne jest jednak unikanie utrudniania pracy kompilatorowi poprzez ręczne modyfikacje kodu źródłowego. Niestety zdarza się, że kod zoptymalizowany ręcznie nie osiąga oczekiwanej wydajności z powodu nieprawidłowej interakcji z mechanizmami optymalizującymi kompilatora.

5.2.2 Kompilatory optymalizujące i opcje optymalizacji

Do uzyskania maksymalnej wydajności kodu na danym złożonym sprzęcie ręczna optymalizacja nie jest wystarczająca i należy zastosować wyrafinowane kompilatory optymalizujące. Optymalizacja jest wykonywana przez kompilatory zwykle po analizie składni i przed wygenerowaniem kodu obiektowego, gdzie niektóre opcje, np. równoległość można zrealizować na etapie przetwarzania wstępnego za pomocą odpowiednich modułów kompilatora. Optymalizacja może zostać dokonana dla czasu kompilacji, wielkości kodu wynikowego, czasu wykonania programu oraz w celu lepszej pracy debuggera. Najczęściej optymalizacja kompilatora jest stosowana za pomocą opcji dla poziomów optymalizacji, gdzie typowe poziomy i przeprowadzane optymalizacje to:

- -O0 - brak optymalizacji (domyślna wartość dla kompilatora gcc).
- -O1 - optymalizacja pod kątem czasu wykonania i rozmiaru kodu (poziom 1).
- -O2 – optymalizacja, gdzie zastosowane zostało więcej opcji optymalizacji pod kątem czasu wykonania i rozmiaru kodu, przejście do opcji, które mogą zmienić wyniki wykonania kodu (poziom 2).
- -O3 - najbardziej agresywna optymalizacja pod kątem czasu wykonania i rozmiaru kodu (poziom 3).
- -Og – optymalizacja używana w celu poprawy pracy debuggera.

Niektóre kompilatory mogą posiadać więcej poziomów optymalizacji np. dla wektoryzacji lub równoległości.

6 Narzędzia wspomagające analizę wydajności obliczeń

6.1 Profilery

Profilery są programami wykorzystywanymi do uzyskania profilu wykonania, lub inaczej wykonania tak zwanego profilowania, poprzez zbieranie danych związanych z wydajnością dla całego programu i przedstawienia ich w zbiorczej postaci. Głównym zastosowaniem profilowania jest uzyskiwanie czasu wykonania dla poszczególnych funkcji kodu lub procedur. Istnieje również możliwość, przy zaawansowanych programach, wnioskowania o czasie wykonania we fragmentach funkcji, blokach kodu lub nawet w poszczególnych liniach kodu[14]. Profilery są popularnymi narzędziami jednak nie zostały one użyte w pracy do analizy ze względu na to, że w kontekście badania wydajności pojedynczej funkcji mnożenia macierz-wektor nie są one używane.

6.2 Liczniki zdarzeń sprzętowych

6.2.1 Idea liczników sprzętowych

Zdarzenia to wystąpienia określonych sygnałów związanych z funkcją procesora. Do obliczeń wydajności w komputerze wykorzystywane są liczniki zdarzeń sprzętowych będące zestawem rejestrów specjalnego przeznaczenia wbudowanych w mikroprocesory. Zliczają zdarzenia, takie jak chybienia w pamięci podręcznej i operacje zmiennoprzecinkowe, podczas gdy program jest wykonywany na procesorze. Pozwalają one również na monitorowanie tych zdarzeń poprzez diagnostykę sprzętu, co ułatwia korelację między strukturą kodu źródłowego, a wydajnością mapowania tego kodu na podstawową architekturę. Każdy procesor ma wiele zdarzeń, które są natywne i częste dla danej architektury. Głównym zadaniem liczników sprzętowych jest przechowywanie zliczeń działań związanych ze sprzętem w systemach komputerowych umożliwiające zebranie danych i zrozumienie, jak procesor oraz inne komponenty sprzętowe reagują na różne zadania[15]. Jednym z zastosowań liczników, które użyte zostało w pracy, jest przeprowadzanie niskopoziomowej analizy wydajności i dostrajania.

6.2.2 Narzędzie perf

Jednym ze sposobów dostępu do rejestrów zliczających konkretne zdarzenia jest wykorzystanie w programach wstawek kodu asemblera zapisujących i odczytujących wartości odpowiednich rejestrów. Rozwiązanie to, poza brakiem przenośności między architekturami

procesorów, może także wymagać wyższych uprawnień przy wykonywaniu kodu, niż dostępne standardowemu użytkownikowi. Z powodu tych wad wprowadzane są narzędzia wyższego poziomu. W systemie Linux istnieje narzędzie perf korzystające z modułu wbudowanego w jądro Linuxa, które podczas śledzenia zapisuje dane na temat wydarzeń, korzystając z punktów śledzenia. Umożliwia to szczegółową analizę zachowania sprzętu oraz związanych z nim zdarzeń[16].

Komenda „perf stat", będąca częścią narzędzia perf, jest narzędziem w systemach Linux, które pozwala na zbieranie statystyk wydajnościowych podczas wykonywania programu oraz na obliczanie liczby zdarzeń w programie. Przykładem informacji z parametrów związanych z wydajnością, które „perf stat" pozwala uzyskać, są informacje na temat liczby cykli zegara oraz instrukcji, a także przełączania kontekstu, błędów strony czy skoków. Dzięki tym informacjom możliwe jest szybkie zidentyfikowanie potencjalnych problemów.

```
Performance counter stats for './2dc_ac':

         1790,90 msec task-clock:u              #    0,999 CPUs utilized
               0      context-switches:u        #    0,000 K/sec
               0      cpu-migrations:u          #    0,000 K/sec
           18198      page-faults:u             #    0,010 M/sec
      4365548964      cycles:u                  #    2,438 GHz
      9116713610      instructions:u            #    2,09  insn per cycle
       753818755      branches:u                #  420,917 M/sec
         3776756      branch-misses:u           #    0,50% of all branches

      1,792252923 seconds time elapsed

      1,743746000 seconds user
      0,047993000 seconds sys
```

Ilustracja 8: Uzyskane wyniki dla komendy „perf stat"

Źródło: opracowanie własne

7. Implementacja mnożenia macierz-wektor dla wybranych formatów przechowywania

W niniejszym rozdziale omówione zostaną fragmenty kodu implementujące algorytm mnożenia macierz-wektor dla macierzy rzadkich, przy wykorzystaniu wybranych formatów przechowywania macierzy wraz z zastosowanymi optymalizacjami. Na podstawie pseudokodów przedstawionych w rozdziale 2.4.1 zrealizowane zostały implementacje funkcji mnożenia macierz-wektor dla formatów CSR oraz SELL. Do utworzenia funkcji realizującej algorytm w sposób równoległy należało funkcję zapisaną w sposób sekwencyjny przekształcić przy pomocy interfejsu OpenMP. W formacie CSR na początku funkcji należało wprowadzić dyrektywę „**#pragma omp parallel**”, a następnie dla zewnętrznej pętli dyrektywę „**#pragma omp for**”. W formacie CSR oraz SELL pętla po wierszach przedstawiona jest jako pierwsza pętla.

Przeprowadzona została również analiza kodu asemblera, dzięki której zauważyć można było za pomocą jakich rozkazów czytane są z pamięci dane do mnożenia i za pomocą jakich rozkazów wykonywane są obliczenia. Przy analizie asemblera najważniejsza była najbardziej wewnętrzna pętla for(), w której obliczenia zawsze (dzięki użytej opcji kompilacji -march=core-avx2) wykonywane były wektorowo za pomocą rozkazu vfmadd.

7.1 Funkcja dla formatu CSR

7.1.1 Kod źródłowy

W funkcji wykorzystywany jest schemat algorytmu mnożenia macierz-wektor, dla którego przedstawiana liczba wierszy określana jest jako WYMIAR, a liczba kolumn obliczana jest na podstawie tablicy **row_ptr**, co pozwala uzyskać liczbę kolumn w danym wierszu. Zmienna **x0**, odczytywana jako numer analizowanej w danym momencie kolumny, w obliczeniach służy jako indeks wartości wektora **x**, która używana jest do obliczania iloczynu. Następnie do kolejnych wartości tablicy **y** dodawany jest obliczony iloczyn.

```
void mat_vec_crs(double* x, double* y,struct CSR csr,long int nn,long int nt){
  register long int i;
  #pragma omp parallel default(none) firstprivate(x,y,csr,nn,i) num_threads(nt)
  {
   register long int n=nn;
   register long int j;
   register long int x0;
   #pragma omp for
   for(i=0;i<n;i++){
      long int edge=csr.row_ptr[i+1]-1;
      for(j=csr.row_ptr[i];j<=edge;j++){
        x0=csr.col_ind[j];
        y[i]+=csr.a_csr[j] * x[x0];
      }
    }
  }
}
```

Ilustracja 9: Równoległa implementacja mnożenia macierz-wektor dla formatu CSR

Źródło: opracowanie własne

7.1.2 Analiza asemblera produkowanego przez kompilator gcc

Poniżej przedstawiony został fragment kodu asemblera dla formatu CSR, w którym znajduje się najważniejsza pętla for() algorytmu:

```
.L4:
        movslq  4(%r10), %rcx
        movslq  (%r10), %rax
        leal    -1(%rcx), %r11d
        cmpl    %eax, %r11d
        jl      .L7
        vmovsd  (%rsi), %xmm0
        .p2align 4,,10
        .p2align 3
.L6:
        movslq  (%r9,%rax,4), %rdx
        vmovsd  (%r8,%rax,8), %xmm1
        incq    %rax
        vfmadd231sd     (%rdi,%rdx,8), %xmm1, %xmm0
        vmovsd  %xmm0, (%rsi)
        cmpq    %rcx, %rax
        jne     .L6
.L7:
        addq    $4, %r10
        addq    $8, %rsi
        cmpq    %r10, %rbx
        jne     .L4
```

Ilustracja 10: Fragment kodu asemblera dla kompilatora gcc (wersja 11.4.0) z opcją -S

Źródło: opracowanie własne

Analizując powyższy fragment można zauważyć, że najważniejsza pętla znajduje się w części .L6. Program pobiera 32-bitową wartość spod (%r9 + rax*4) i przenosi do 64-bitowego rdx, poprzez vmovsd (%r8,%rax,8), %xmm1 ładuje 64-bitową wartość z (%r8 + rax*8) do xmm1 i zwiększa rax o 1. Poprzez instrukcję vfmadd213sd (%rdi,%rdx,8), %xmm1, %xmm0 wykonuje równanie xmm0 = xmm1 * (rdi + rdx*8) + xmm0, które z rejestru xmm0 zapisywane jest do rsi, a następnie program porównuje wartości rejestrów rcx i rax w celu przejścia do kolejnej iteracji pętli.

Niestety kod nie jest zwektoryzowany, używane są wersje skalarne rozkazów jednak kompilator zrealizował kilka optymalizacji:

- Wykorzystanie instrukcji vfmadd213sd łącząc mnożenie i dodawanie w pojedynczej instrukcji skraca czas wykonania i minimalizuje operacje zapisu/odczytu.
- Zmniejszona liczba pobrań z pamięci poprzez ładowanie wartości do rejestrów xmm (jednak w rejestrze 128-bitowym wykorzystane są tylko 64 bity) przed wykonaniem operacji arytmetycznych, minimalizując liczbę operacji odczytu/zapisu z pamięci. W każdym przebiegu pętli głównej są trzy ładowania (z tablicy col_ind, a i x) i jeden zapis do tablicy y - jednak zapis można przenieść poza pętlę (korzystając z rejestru tymczasowego).
- Konwersja wartości całkowitych z 32-bitowych na 64-bitowe jest realizowana przez użycie movslq, co pozwala uniknąć dodatkowych konwersji.

7.2 Funkcja dla formatu SELL

7.2.1 Kod źródłowy

W funkcji tej, podobnie jak w formacie CSR, zewnętrzną pętlą jest pętla po wierszach, a dokładniej pętla po blokach wierszy (plastrach). Pętla po kolumnach w i-tym bloku wierszy swój zakres kończy na wartości odczytanej z tablicy długości plastrów dla i-tego wiersza. W tym formacie występuje również trzecia pętla przechodząca po wartościach do rozmiaru wysokości bloku wierszy C. Następnie wyznaczana jest pomocnicza wartość **pom** w celu wyboru elementu z tablicy wartości macierzy. W tym formacie, tak jak w CSR występuje zmienna **x0**, która odczytywana jest z tablicy **slice_col** jako indeks analizowanej w danym momencie kolumny, a więc także indeks wartości wektora **x**, która używana jest do obliczeń iloczynu macierz-wektor. Tablica **slice_col** w opisie algorytmu przedstawiona została jako tablica **col_ind**. Na koniec do wartości odpowiedniego elementu w tablicy **y**, wyliczanego w oparciu o rozmiar C, dodawana

jest wartość wyliczona na podstawie mnożenia.

Na ilustracji nr 11 przedstawiona została również pętla służąca do przypisania alokowanej w funkcji tablicy y do globalnej tablicy y_global.

```
void mat_vec_sell(double* x, double* y_global, struct Sell sell, long int nn, long int nt) {
  register long int i;
  double* y = malloc(C * sell.nr_slice * sizeof(double));
  for (i = 0; i < C * sell.nr_slice; i++) y[i] = 0.0;

  #pragma omp parallel default(none) firstprivate(x,y,y_global,sell,nn,i) num_threads(nt)
  {
    register long int n = nn;
    register long int j;
    register long int pom;
    register long int x0;
    int k;

    int wyn = sell.nr_slice;

    // pętla po blokach wierszy
    #pragma omp for
    for (i = 0; i < wyn; i++) {

      // pętla po kolumnach w i-tym bloku wierszy
      for (j = 0; j < sell.cl[i]; j++) {
        for(k=0;k<C;k++){
        pom = sell.sliceStart[i] + j * C + k;
        x0 = sell.slice_col[pom];
        y[i * C + k] += sell.val[pom] * x[x0];
        }
      }
    }
  }

  for (i = 0; i < nn; i++)
    y_global[i] = y[i];
```

Ilustracja 11: Równoległa funkcja macierz-wektor dla formatu SELL

Źródło: opracowanie własne

7.2.2 Analiza asemblera produkowanego przez kompilator gcc

- Fragmenty kodu asemblera dla formatu SELL

```
.L3:
        movslq  (%rbx,%r9,4), %rsi
        testl   %esi, %esi
        jle     .L6
        movslq  (%r12,%r9,4), %r8
        leaq    (%r8,%rsi,4), %rsi
        leaq    (%r10,%r8,4), %rdx
        leaq    0(%rbp,%r8,8), %rcx
        leaq    (%r10,%rsi,4), %r8
        .p2align 4,,10
        .p2align 3
.L4:
        movslq  (%rdx), %rsi
        vmovsd  (%rcx), %xmm0
        addq    $16, %rdx
        addq    $32, %rcx
        vmovsd  (%rax), %xmm1
        vmovsd  8(%rax), %xmm2
        vfmadd132sd     (%rdi,%rsi,8), %xmm1, %xmm0
        movslq  -12(%rdx), %rsi
        vmovsd  16(%rax), %xmm3
        vmovsd  24(%rax), %xmm4
        vmovsd  %xmm0, (%rax)
        vmovsd  -24(%rcx), %xmm0
        vfmadd132sd     (%rdi,%rsi,8), %xmm2, %xmm0
        movslq  -8(%rdx), %rsi
        vmovsd  %xmm0, 8(%rax)
        vmovsd  -16(%rcx), %xmm0
        vfmadd132sd     (%rdi,%rsi,8), %xmm3, %xmm0
        movslq  -4(%rdx), %rsi
        vmovsd  %xmm0, 16(%rax)
        vmovsd  -8(%rcx), %xmm0
        vfmadd132sd     (%rdi,%rsi,8), %xmm4, %xmm0
        vmovsd  %xmm0, 24(%rax)
        cmpq    %rdx, %r8
        jne     .L4
.L6:
        incq    %r9
        addq    $32, %rax
        cmpq    %r11, %r9
        jne     .L3
```

Ilustracja 12: Fragment kodu asemblera dla kompilatora gcc (wersja 11.4.0) z opcją -S

Źródło: opracowanie własne

Program w głównej pętli (blok podstawowy zaczyna się od etykiety .L4) pobiera wartość 32-bitową spod (%rdx) i przenosi do 64-bitowego rsi. Poprzez vmovsd (%rcx), %xmm0 ładuje 64-bitową wartość zmiennoprzecinkową z adresu (%rcx) do xmm0. Następnie zwiększany jest wskaźnik rdx o 16 bajtów oraz wskaźnik rcx o 32 bajty. Poprzez komendy vmovsd (%rax), %xmm1 oraz vmovsd 8(%rax), %xmm2 ładowane są 64-bitowe wartości zmiennoprzecinkowe spod adresu (%rax) do xmm1 oraz spod adresu 8(%rax) do xmm2. Poprzez funkcję vfmadd132sd (%rdi,%rsi,8), %xmm1, %xmm0 wykonywana jest operacja xmm0 = xmm1 * (rdi + rsi*8) + xmm0. Ze względu na występujący rozmiar C=4 w kolejnych krokach wykonywane jest:

- Pobranie wartości 32-bitowej z adresu -12(%rdx) do rsi, poprzez movslq, załadowanie 64-bitowej wartości spod adresu 16(%rax) do xmm3 oraz 64-bitowej wartości spod adresu 24(%rax) do xmm4 i wykonanie operacji xmm0 = xmm2 * [rdi + rsi*8] + xmm0.
- Pobranie wartości 32-bitowej z adresu -8(%rdx) do rsi poprzez movslq i wykonanie operacji xmm0 = xmm3 * [rdi + rsi*8] + xmm0.
- Pobranie wartości 32-bitowej z adresu -4(%rdx) do rsi poprzez movslq i wykonanie operacji xmm0 = xmm4 * [rdi + rsi*8] + xmm0.

Na koniec zapisywany jest wynik xmm0 do pamięci pod adresem 24(%rax), a następnie program porównuje wartości rejestrów rdx i r8 w celu przejścia do kolejnej iteracji pętli.

Ten kod również nie został zwektoryzowany przez co wykorzystywane są rozkazy skalarne.

7.3 Funkcja wektorowa dla formatu SELL

7.3.1 Kod źródłowy

W celu zwiększenia wydajności przetwarzania wykorzystana została wektoryzacja obliczeń, gdzie uzyskiwane są lepsze efekty, gdy elementy macierzy występujące w kolejnych operacjach algorytmu przechowywane są w następujących po sobie lokalizacjach w pamięci. Sposób ten realizowany jest poprzez format SELL i jego algorytm mnożenia, dlatego też w programie wykorzystano wektoryzację dla tego formatu. W zastosowanej funkcji zewnętrzną pętlą jest pętla przechodząca po plastrach, a wewnętrzną pętla przechodząca po kolumnach plastra. W pętlach do zmiennych pomocniczych pakowane są cztery wartości macierzy rzadkiej (zmienna **val**) i dwie wartości wektora (zmienna **rhstmp**). Następnie 128 bitowa zmienna **rhstmp** wstawiana jest do 256 bitowej zmiennej **rhs**. Proces ten jest powtarzany również dla dwóch kolejnych elementów wektora **x**. Na koniec wykonywane jest mnożenie zmiennych **val** i **rhs**, a wynik dodawany do zmiennej **tmp**, która zapisywana jest do wektora **y**.

```
int mat_vec_vector(double *x,double* y,struct Sell sell,long int nt){

  int c;
  #pragma omp parallel for default(none) firstprivate(y,x,sell)
  for (c=0; c < sell.nr_slice; c ++){ //pętla po plastrach
    int j , offs ;
    __m256d tmp , val , rhs ;
    __m128d rhstmp ;
    tmp = _mm256_load_pd(&y[c<<2]); //wpakuj 4 wartości LHS
    offs = sell.sliceStart[c]; // poczatkowy offset jest początkiem plastra
    for (j =0; j < sell.cl[c]; j ++){ //pętla w plastrze po wartościach do długści plastra
      val = _mm256_load_pd(&sell.val[offs]); //załaduj 4 wartości macierzy rzadkiej
      rhstmp = _mm_loadl_pd(rhstmp ,&x[sell.slice_col[offs ++]]); //załaduj pierwszą wartość RHS (wektor)
      rhstmp = _mm_loadh_pd(rhstmp ,&x[sell.slice_col[offs ++]]); //załaduj drugą wartość RHS (wektor)
      rhs = _mm256_insertf128_pd(rhs , rhstmp ,0); //wstaw spakowane wartości
      rhstmp = _mm_loadl_pd(rhstmp ,&x[sell.slice_col[offs ++]]); //załaduj trzecią wartość RHS (wektor)
      rhstmp = _mm_loadh_pd(rhstmp ,&x[sell.slice_col[offs ++]]); //załaduj czwartą wartość RHS (wektor)
      rhs = _mm256_insertf128_pd(rhs , rhstmp ,1); //wstaw spakowane wartości
      tmp = _mm256_add_pd(tmp , _mm256_mul_pd(val , rhs )); //gromadzenie wartości
    }
    _mm256_store_pd(&y[c<<2] , tmp ); //przechowaj 4 wartości LHS (y)
  }
}
```

Ilustracja 13: Funkcja macierz-wektor realizowana w sposób równoległy wektorowy dla formatu SELL

Źródło: opracowanie własne na podstawie [17]

7.3.2 Analiza asemblera produkowanego przez kompilator gcc

- Fragmenty kodu asemblera dla formatu SELL zrealizowanego w sposób wektorowy

```
.L5:
        movslq  %r9d, %rax
        movl    (%r8), %ecx
        leaq    (%rbx,%rax,8), %r10
        movslq  (%r12,%r9), %rax
        vmovapd (%r10), %ymm2
        testl   %ecx, %ecx
        jle     .L3
        movq    24(%rbp), %rcx
        leaq    (%rcx,%rax,8), %rsi
        leaq    0(%r13,%rax,4), %rax
        xorl    %ecx, %ecx
        .p2align 4,,10
        .p2align 3
.L4:
        movslq  (%rax), %rdi
        incl    %ecx
        addq    $32, %rsi
        addq    $16, %rax
        vmovlpd (%rdx,%rdi,8), %xmm0, %xmm0
        movslq  -12(%rax), %rdi
        vmovhpd (%rdx,%rdi,8), %xmm0, %xmm1
        movslq  -8(%rax), %rdi
        vmovlpd (%rdx,%rdi,8), %xmm1, %xmm0
        movslq  -4(%rax), %rdi
        vmovhpd (%rdx,%rdi,8), %xmm0, %xmm0
        vinsertf128     $0x1, %xmm0, %ymm1, %ymm1
        vfmadd231pd     -32(%rsi), %ymm1, %ymm2
        cmpl    %ecx, (%r8)
        jg      .L4
.L3:
        addq    $4, %r9
        vmovapd %ymm2, (%r10)
        addq    $4, %r8
        cmpq    %r9, %r11
        jne     .L5
```

Ilustracja 14: Fragment kodu asemblera dla kompilatora gcc (wersja 11.4.0) z opcją -S

Źródło: opracowanie własne

W głównej pętli (fragment .L4) program pobiera wartość 32-bitową spod adresu (%rax), zapisuje ją do rdi i zwiększa wartość rejestru ecx o 1. Poprzez komendę addq przesuwa wskaźnik rsi o 32 bajty, przygotowując do odczytu kolejnego bloku pamięci oraz przesuwa wskaźnik rax o 16 bajtów. Funkcja vmovlpd ładuje 128-bitowy (2-elementowy) wektor z pamięci spod adresu (%rdx + %rdi * 8) do dolnej części xmm0. Następnie pobierana jest 32-bitowa wartość z adresu -12(%rax) i zapisywana do rejestru rdi. Ze względu na występujący rozmiar C=4 w kolejnych krokach wykonywane jest:

- Załadowanie górnej połowy (64 bity) z (%rdx + %rdi * 8) do rejestru xmm0, pobranie 32-bitowej wartości spod -8(%rax) i zapisanie jej do rdi.
- Załadowanie dolnej połowy (128-bit) z (%rdx + %rdi * 8) do xmm1, pobranie 32-bitowej wartości spod -4(%rax) i zapisanie jej do rejestru rdi.
- Załadowanie górnej połowy (64 bity) do xmm0.

Następnie poprzez komendę vinsertf128 $0x1, %xmm0, %ymm1, %ymm1 rejestr xmm0 wstawiany jest jako górne 128 bity do ymm1. Poprzez funkcję vfmadd132pd wykonywana jest operacja FMA, czyli mnożenie ymm1 z wartościami z pamięci (%rsi - 32) i dodanie wyniku do ymm2. Na zakończenie porównywane są wartości ecx z wartością przechowywaną w rejestrze(%r8)w celu przejścia do kolejnej iteracji pętli.

W powyższym kodzie kompilator zrealizował kilka optymalizacji:

- Użycie rejestrów ymm, które pozwala na operacje na czterech 64-bitowych elementach zmiennoprzecinkowych jednocześnie.
- Użycie instrukcji vfmadd132sd dla zredukowania liczby operacji.
- Zredukowana liczba ładowań do rejestrów poprzez użycie rejestrów xmm/ymm, co minimalizuje opóźnienia związane z dostępem do pamięci.
- Użycie vinsertf128 pozwalające na efektywne tworzenie 256-bitowych wektorów z dwóch 128-bitowych fragmentów, co usprawnia przetwarzanie wektorowe.

7.4 Algorytm mnożenia macierz-wektor dla MPI

W celu przeprowadzenia obliczeń na programie ze specyfikacją MPI do implementacji kodu użyta została wersja niezwektoryzowana formatu ELLPACK. Poprzez dodanie podstawowych procedur: MPI_Init(), MPI_Comm_rank(), MPI_Comm_size() na początku kodu oraz procedury komunikacji grupowej, które poprzez rozdzielenie tablic na lokalne umożliwiają wspólną interakcję między procesami możliwe było przeprowadzenie obliczeń. W programie wykonano dekompozycje macierzy oraz wektora na lokalne części, tak aby każdy proces mógł przeprowadzać obliczenia na własnych zasobach. W kodzie występują funkcje MPI_Bcast(), dla zmiennych nr_slice_local, oznaczającą lokalną wysokość plastra, cl_max zliczającą maksymalną długość plastrów oraz tablicy x_new będącej rozszerzeniem tablicy x. Wysyłanie i odbieranie wartości cl, val i slice_col zrealizowane zostało poprzez funkcje MPI_Send() oraz MPI_Recv(), przy czym MPI_Send() realizowane jest jedynie przez proces 0, a MPI_Recv() realizowane jest przez pozostałe procesy. Następnie funkcja MPI_Allgather() zbiera wartości tablicy x_new, dzięki czemu każdy proces dla

swoich tablic uruchamia algorytm mnożenia macierz-wektor. Po obliczeniach funkcja MPI_Gather() zbiera wyniki z okalnych tablic y_local do globalnej tablicy y.

Na ilustracji nr 15 przedstawiony został pseudokod algorytmu mnożenia macierz-wektor dla formatu SELL przy użyciu specyfikacji MPI:

```
for i = 0, i < nr_slice_local
  for j = 0, j < cl_max
    for k=0, k<C
      pom = sliceStart_local[i] + j * C + k
      x0 = slice_col_local
      y_local[i * C + k] += val_local[pom] * x_new[x0]

    end;
  end;
end;
```

Ilustracja 15: Mnożenie macierz-wektor przedstawione w sposób równoległy przy użyciu specyfikacji MPI dla formatu SELL

Źródło: Opracowanie własne

8. Eksperymenty obliczeniowe i analiza wydajności

8.1 Sprzęt obliczeniowy

Do przedstawienia działania funkcji mnożenia macierz-wektor użyta została macierz przygotowana w programie obliczeń metodą elementów skończonych ModFEM[18] w formacie CRS o wymiarach 476 912 x 476 912, która posiada 9 788 784 niezerowych elementów. Testy implementacji mnożenia macierz-wektor przeprowadzone zostały na dwóch systemach: Honorata oraz Ares.

8.1.1 Serwer Honorata

Serwer Honorata jest dwuprocesorowym serwerem z procesorami Intel Xeon E5-2630 v4, pracującym pod kontrolą systemu operacyjnego Ubuntu. Jako kompilator wykorzystany został gcc w wersji 11.4.0. W celu wyznaczenia różnic uzyskanych między czasami mnożenia macierz-wektor dla użytych formatów wykorzystana została funkcja zliczania czasu ze środowiska OpenMP, omp_get_wtime() oraz ze środowiska MPI funkcja MPI_Wtime(). Dodatkowo z faktu, że rejestry w AVX2 są 256 bitowe, a liczba double ma 64 bity, testowanie macierzy w formacie SELL przeprowadzone zostało tylko dla wartości C=4. Program został skompilowany i uruchomiony kilkukrotnie w celu znalezienia najkrótszych czasów wykonania.

8.1.2 Superkomputer Ares

Ares jest to superkomputer zbudowany z serwerów obliczeniowych z procesorami firmy Intel (modele Xeon Platinum i Xeon Gold), podzielonych na trzy grupy:

- 532 serwery, każdy wyposażony w 192 GB pamięci RAM,
- 256 serwerów, każdy posiadający 384 GB pamięci RAM,
- 9 serwerów, każdy posiadający 8 kart NVIDIA Tesla V100.

Sumaryczna, teoretyczna wydajność części CPU to ponad 3,5 PFlops, a części GPU to ponad 500 TFlops. Aresa wspomaga również system dyskowy o pojemności ponad 11 PB. Do przesyłania danych używana jest sieć typu InfiniBand EDR. Superkomputer posiada 37 824 rdzeni obliczeniowych. Został też wyposażony w system chłodzenia cieczą[19].

8.2 Analiza jednowątkowego wykonania programu

8.2.1 Uzyskane czasy

Po uruchomieniu kodu otrzymano wyniki, które w sposób tabelaryczny przedstawione zostały w tabelach nr 1 i 2 dla serwera Honorata oraz w tabelach nr 3 i 4 dla superkomputera Ares:

Sekwencyjne			
Czas [ns]	**Format**		
	CSR	**SELL**	**SELL wektor**
	0.0151	0.0129	0.0087

Tabela 1: Uzyskane wyniki czasu dla programu na serwerze Honorata

Źródło: opracowanie własne

Sekwencyjne			
GFLOPS	**Format**		
	CSR	**SELL**	**SELL wektor**
	1.2965	1.5176	2.2503

Tabela 2: Uzyskane wyniki jednostki GFLOPS dla programu na serwerze Honorata

Źródło: opracowanie własne

Sekwencyjne			
Czas [ns]	**Format**		
	CSR	**SELL**	**SELL wektor**
	0.0125	0.0124	0.0138

Tabela 3: Uzyskane wyniki czasu dla programu na serwerze Ares

Źródło: opracowanie własne

Sekwencyjne			
GFLOPS	**Format**		
	CSR	**SELL**	**SELL wektor**
	1.5662	1.5788	1.4187

Tabela 4: Uzyskane wyniki jednostki GFLOPS dla programu na serwerze Ares

Źródło: opracowanie własne

8.2.2 Analiza wyników liczników zdarzeń sprzętowych

Analizując wyniki po uruchomieniu komendy „perf stat” można zauważyć, że program charakteryzuje się wysoką wartością IPC, wynoszącą 1.32, co wskazuje na dobrą optymalizację kodu i efektywne wykorzystanie procesora. Wskaźnik błędnych przewidywań skoków przedstawia skuteczne przewidywanie przez mechanizm skoków. Uzyskany czas systemowy sugeruje, że operacje systemowe nie stanowiły dużego obciążenia, a program skupiał się na rzeczywistych obliczeniach.

```
Performance counter stats for './mat_vec_test_driver':

          3 398,36 msec task-clock                #    1,017 CPUs utilized
                78      context-switches          #   22,952 /sec
                 0      cpu-migrations            #    0,000 /sec
            10 112      page-faults               #    2,976 K/sec
     9 497 057 144      cycles                    #    2,795 GHz
    12 529 317 187      instructions              #    1,32  insn per cycle
     2 867 602 046      branches                  #  843,819 M/sec
         1 893 855      branch-misses             #    0,07% of all branches

       3,342711556 seconds time elapsed

       3,333041000 seconds user
       0,068349000 seconds sys
```

Ilustracja 16: Wyniki uzyskane na serwerze Honorata

Źródło: opracowanie własne

8.3 Analiza wielowątkowego wykonania programu przy użyciu OpenMP

8.3.1 Uzyskane czasy

Po uruchomieniu kodu otrzymano wyniki, które w sposób tabelaryczny przedstawione zostały w tabelach nr 5 i 6 dla serwera Honorata oraz w tabelach nr 7 i 8 dla superkomputera Ares:

Równolegle			
Czas [ns]	**Format**		
	CSR	**SELL**	**SELL wektor**
2 wątki	0.0093	0.0092	0.0063
4 wątki	0.0058	0.0052	0.0062
8 wątków	0.0037	0.0041	0.0051

Tabela 5: Uzyskane wyniki czasu dla programu na serwerze Honorata

Źródło: opracowanie własne

Równolegle			
GFLOPS	**Format**		
	CSR	**SELL**	**SELL wektor**
2 wątki	2.1051	2.1280	3.1076
4 wątki	3.3754	3.7649	3.1577
8 wątków	5.2912	4.7750	3.8387

Tabela 6: Uzyskane wyniki wydajnościowe implementacji mnożenia macierz-wektor dla różnych formatów przechowywania macierzy w GFLOPS dla programu na serwerze Honorata

Źródło: opracowanie własne

Równolegle			
Czas [ns]	**Format**		
	CSR	**SELL**	**SELL wektor**
2 wątki	0.0063	0.0062	0.0063
4 wątki	0.0034	0.0032	0.0057
8 wątków	0.0030	0.0024	0.0022

Tabela 7: Uzyskane wyniki czasu dla programu na serwerze Ares

Źródło: opracowanie własne

Równolegle			
GFLOPS	**Format**		
	CSR	**SELL**	**SELL wektor**
2 wątki	3.1076	3.1577	3.1076
4 wątki	5.7581	6.1180	3.4347
8 wątków	6.5259	8.1573	8.8989

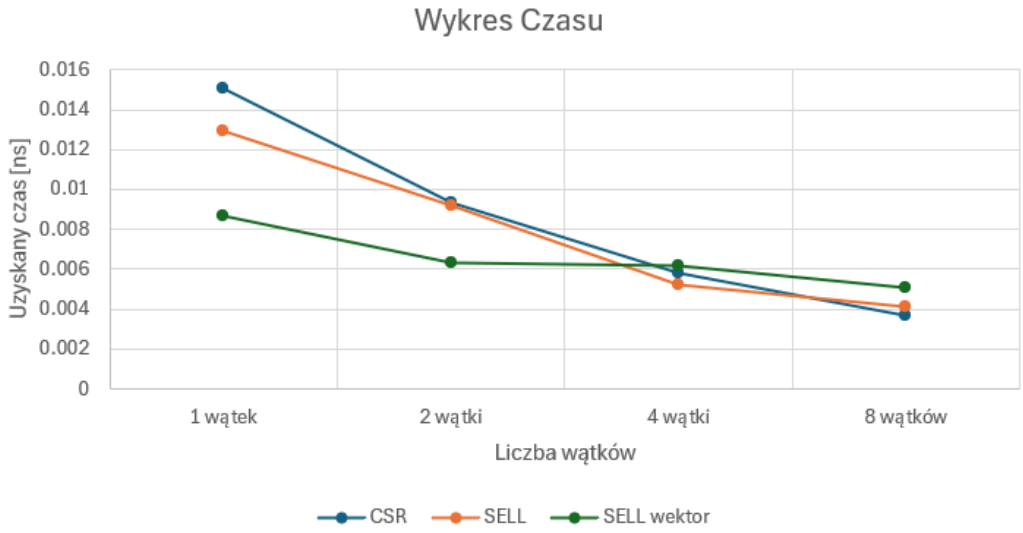
Tabela 8: Uzyskane wyniki wydajnościowe implementacji mnożenia macierz-wektor dla różnych formatów przechowywania macierzy w GFLOPS dla programu na serwerze Ares

Źródło: opracowanie własne

8.3.2 Wyniki wydajnościowe

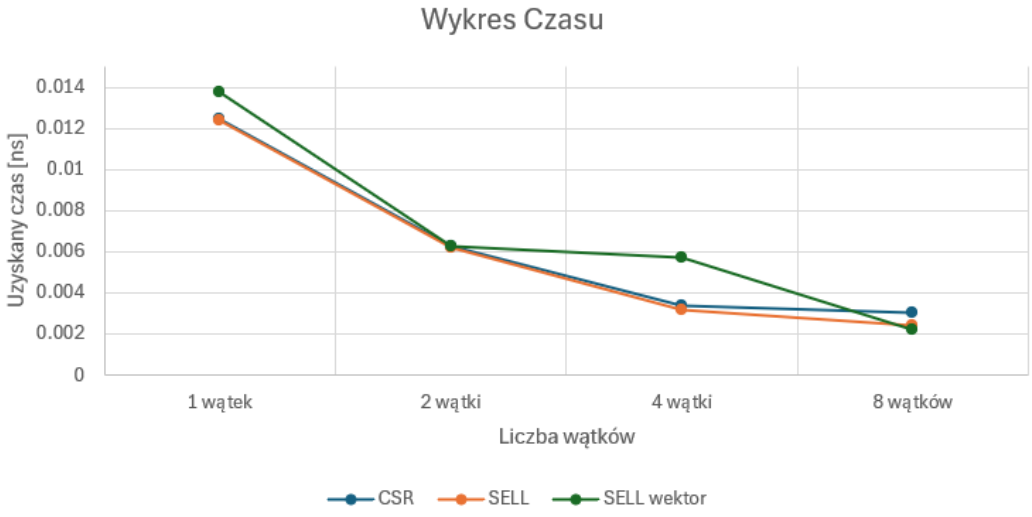
Na podstawie wyników uzyskanych w tabelach 1, 2 i 5, i 6 dla serwera Honorata oraz w tabelach 3, 4 i 7, i 8 dla superkomputera Ares stworzone zostały wykresy:

• wykres zależności czasu od liczby wątków (ilustracja nr 17 oraz nr 18),



Ilustracja 17: Wykres czasu dla programu na serwerze Honorata

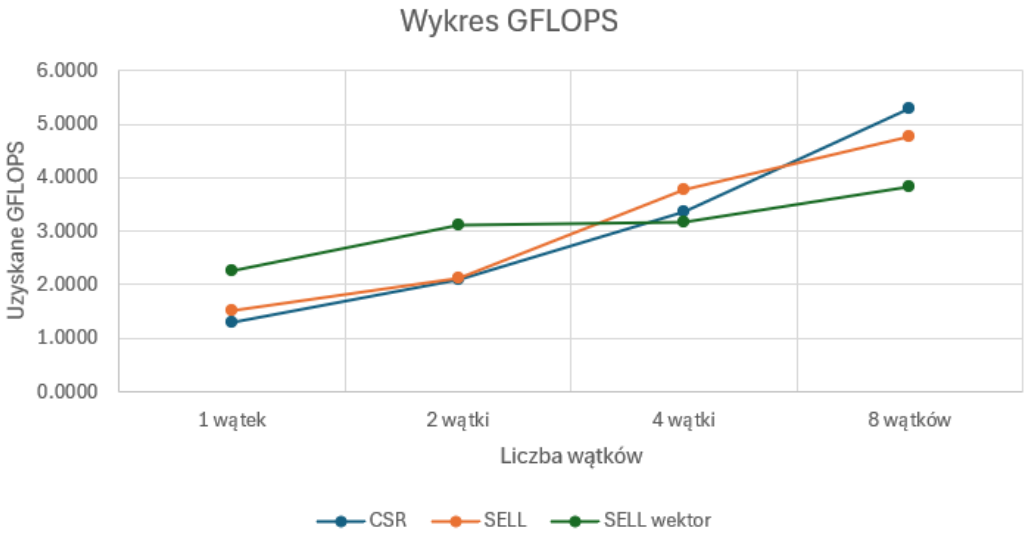
Źródło: opracowanie własne



Ilustracja 18: Wykres czasu dla programu na serwerze Ares

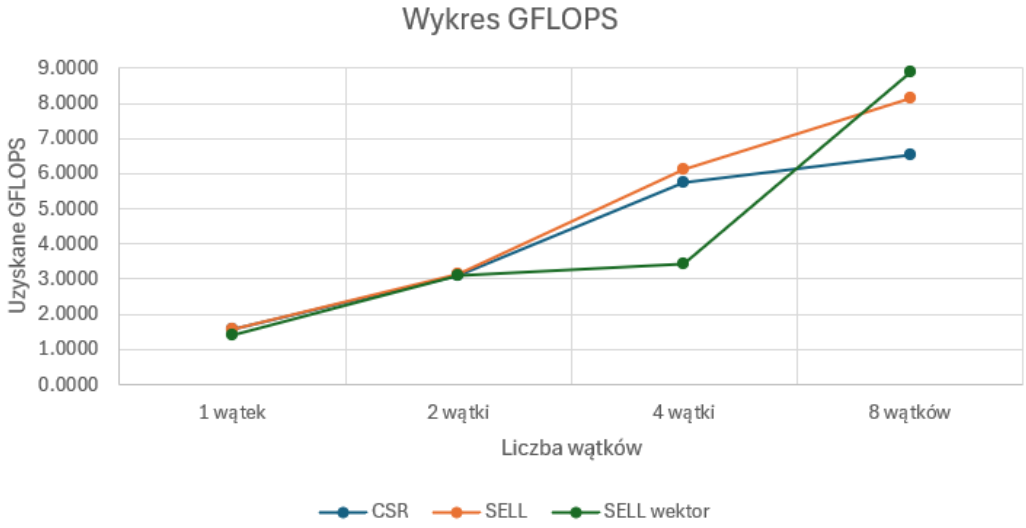
Źródło: opracowanie własne

• wykres zależności uzyskanych GFLOPS od liczby wątków (rys. 19 oraz 20),



Ilustracja 19: Wykres uzyskanych GFLOPS na serwerze Honorata

Źródło: opracowanie własne

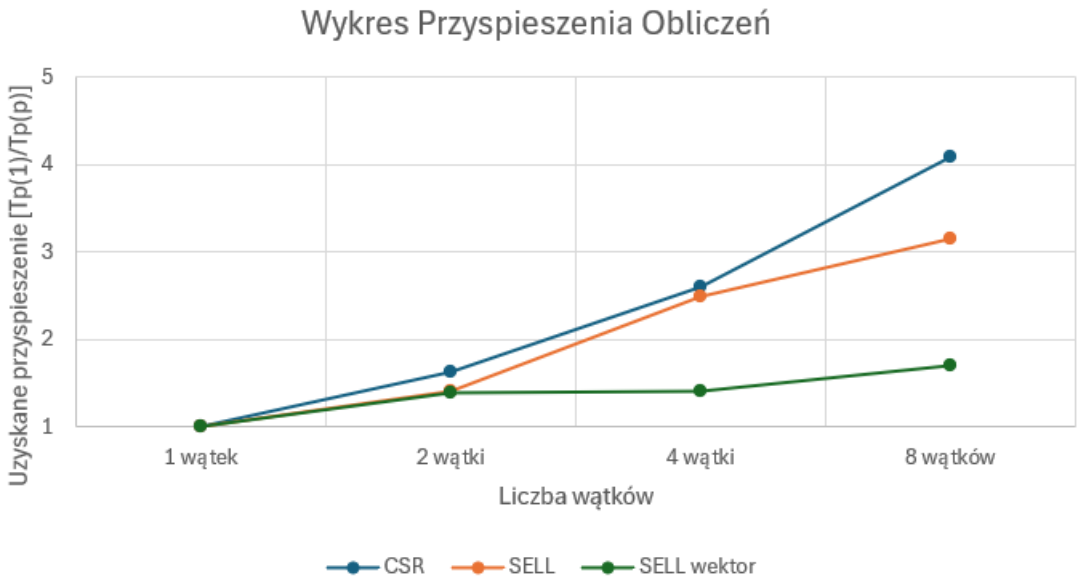


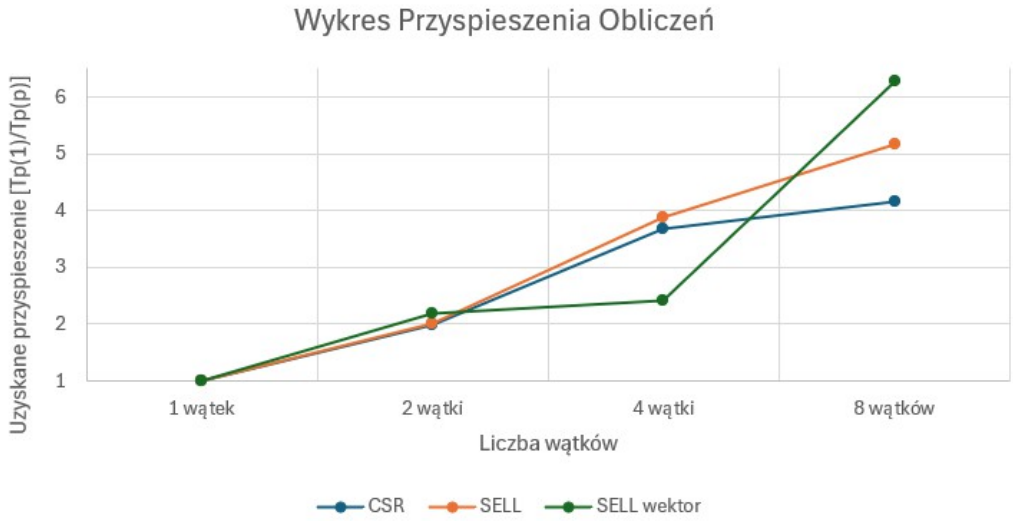
Ilustracja 20: Wykres uzyskanych GFLOPS na serwerze Ares

Źródło: opracowanie własne

• wykres zależności uzyskanego przyspieszenia od liczby wątków (rys. 21 oraz 22).

Wartości przyspieszenia widoczne na ilustracji nr 21 oraz nr 22 zostały obliczone na podstawie wzoru Tp(1)/Tp(p).



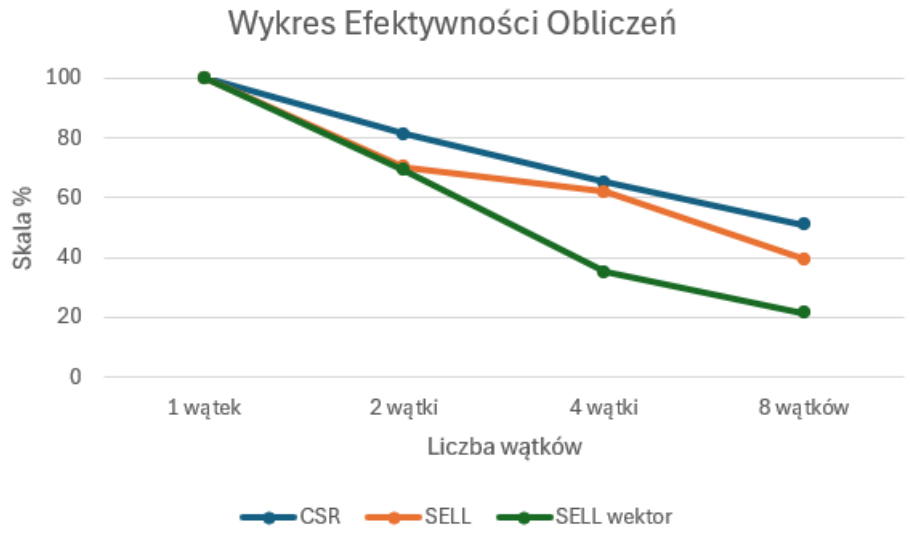


Ilustracja 22: Wykres przyspieszenia uzyskany na serwerze Ares

Źródło: opracowanie własne

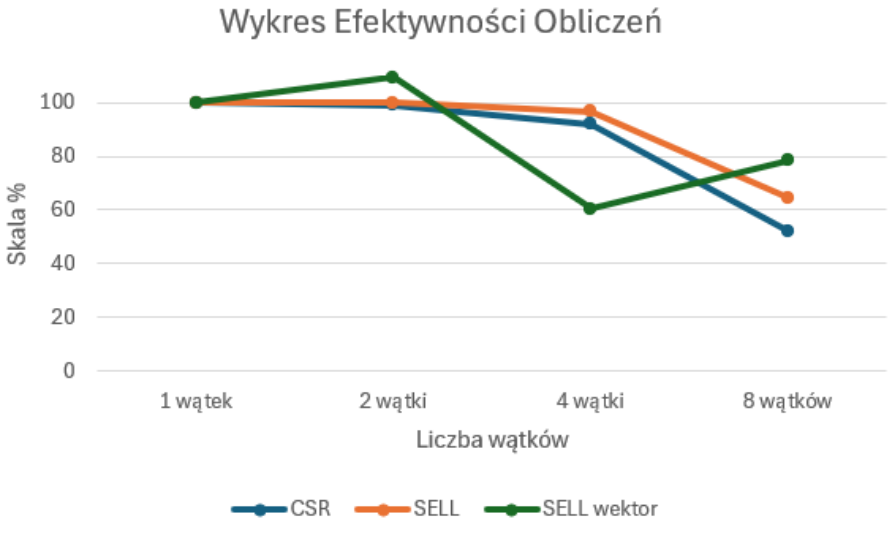
• wykres efektywności zrównoleglenia w zależności od liczby wątków (rys. 23 oraz 24).

Wartości efektywności widoczne na ilustracji nr 23 oraz nr 24 zostały obliczone na podstawie wzoru E(p) = S(p) / p.



Ilustracja 23: Wykres efektywności uzyskany na serwerze Honorata

Źródło: opracowanie własne



Ilustracja 24: Wykres efektywności uzyskany na serwerze Ares

Źródło: opracowanie własne

8.3.3 Analiza i wnioski

Analizując uzyskane w tej metodzie wyniki można zauważyć, że uzyskiwane czasy dla formatów zmniejszają się w miarę zwiększania liczby wątków, co sugeruje poprawę wydajności przy większej liczbie wątków. Jednakże największe zmiany można zauważyć dla formatów CSR oraz SELL. Użycie formatu SELL realizowanego wektorowo również przedstawia zmniejszanie czasu wraz ze wzrostem liczby wątków, ale nie jest to tak wyraźne jak w pozostałych formatach. Na superkomputerze Ares wyniki dla poszczególnych formatów są do siebie bardziej zbliżone jednak największe zmiany można zauważyć dla formatu SELL realizowanego w sposób wektorowy. Na obu serwerach użycie formatu SELL realizowanego w sposób wektorowy uzyskuje niskie czasy od samego początku, jednak wyniki dla 8 wątków na superkomputerze Ares są około dwukrotnie niższe niż uzyskane na serwerze Honorata.

W analizie wykresów GFLOPS wzrost liczby wątków powoduje stałe zwiększanie uzyskiwanych GFLOPS. Wydajność formatu CSR, w miarę zwiększania liczby wątków, na serwerze Honorata rośnie bardziej dynamicznie niż na Aresie, a przy 8 wątkach metoda ta osiąga najwyższe wartości GFLOPS, co wskazuje na jej dobrą skalowalność i optymalność przy dużej liczbie wątków. Format SELL realizowany w sposób wektorowy na serwerze Honorata uzyskuje stabilne wyniki, osiągając bardzo wysoką wydajność przy mniejszej liczbie wątków. Również w tym formacie najwyższe wyniki uzyskiwane są dla 8 wątków, jednak na superkomputerze Ares wyniki te są prawie dwukrotnie wyższe niż na serwerze Honorata. Wyniki GFLOPS dla poszczególnych formatów są do siebie bardziej zbliżone na serwerze Honorata.

Analizując wykresy przyspieszeń obliczeń można zauważyć, że format CSR jak i format SELL wykazują stale rosnące przyspieszenie. Format SELL realizowany wektorowo na serwerze Honorata posiada stabilne przyspieszenie, które wzrasta w miarę zwiększania liczby wątków. Na superkomputerze Ares wyniki te uzyskują bardziej widoczny wzrost.

Można jednak zauważyć różnicę w uzyskanych wynikach efektywności. Wraz ze wzrostem liczby procesów efektywność maleje jednak większy spadek widoczny jest na serwerze Honorata. Na superkomputerze Ares dla 2 wątków wyniki wszystkich badanych formatów oscylują na poziomie 100%. Dla 8 wątków format SELL realizowany wektorowo uzyskał efektywność na poziomie około 80%.

8.4 Uruchomienie programu przy użyciu MPI

8.4.1 Uruchomienie programu MPI na serwerze Honorata

Jako implementacji standardu MPI na serwerze Honorata użyto OpenMPI. Na serwerze program uruchomiony został poprzez użycie narzędzi **mpicc** i **mpiexec**. Narzędzie mpicc służy do skompilowania i połączenia programów MPI napisanych w języku C. Dostarcza ono opcji i specjalnych bibliotek, które przyczyniają się do poprawnego zbudowania plików programu wykorzystującego MPI np. pliku wykonywalnego programu. Mpiexec to program zastępczy dla skryptu mpirun, który jest częścią pakietu MPICH i został zdefiniowany przez standard MPI. Służy do inicjowania zadania równoległego oraz pozwala określić liczbę procesów i ich rozmieszczenie na węzłach w klastrze. Komenda ta powinna być uruchamiana jako **mpiexec -n <numprocs> <program>**, gdzie <**program>** uruchamiany jest na **<numprocs>** procesach równoległych. W pracy uruchamiana była komenda:

/usr/bin/mpiexec -display-map --bind-to core --map-by socket -np <numprocs> ./<program>, gdzie parametry **-display-map --bind-to core --map-by socket** przedstawiają powiązanie procesów z określonymi procesorami, które można zrealizować poprzez określenie tych opcji podczas wykonywania programu.

8.4.2 Uruchomienie programu MPI na superkomputerze Ares

Uruchomienie programu MPI na klastrze obliczeniowym Aresa możliwe było poprzez użycie SLURM (*ang. Simple Linux Utility for Resource Management*) otwartego oprogramowania do zarządzania klastrami i harmonogramowania zadań wykorzystywanego w środowiskach obliczeniowych. Narzędzie wspiera uruchamianie aplikacji MPI, co umożliwia efektywne przetwarzanie równoległe w dużych klastrach. System składa się z centralnego demona zarządzającego (slurmctld) oraz demonów na węzłach obliczeniowych (slurmd), które zapewniają odporną na błędy komunikacji. Slurm umożliwia przydzielanie zasobów (np. węzłów), uruchamianie i monitorowanie zadań, a także zarządzanie kolejkami zadań według priorytetów. Obsługuje szeroką gamę poleceń, takich jak **srun** do uruchamiania zadań w czasie rzeczywistym, **sbatch** do wysyłania skryptów wsadowych czy **squeue** do monitorowania zadań[20].

W pracy na superkomputerze Ares program uruchamiany był za pomocą komendy: **srun -p plgrid -A plgmodfem4-cpu -N <liczba_węzłów> --ntasks-per-node=<liczba wątków> --pty /bin/bash -l**, gdzie **-p plgrid** oznacza specyfikację partycji, **-A plgmodfem4** oznacza nazwę grantu do rozliczenia zużycia zasobów. Granty na Aresie wymagają podania przyrostka do nazwy

grantu: **-cpu**. Parametr **-N** oznacza liczbę alokowanych węzłów, **--ntasks-per-node=** oznacza liczbę wątków przypadającą na każdy węzeł. Samo polecenie `srun` odpowiada za uruchomienie komendy w ramach zaalokowanych zasobów. Jednak w przypadku, gdy zasoby nie zostały wcześniej zaalokowane, komenda ta uprzednio dokonuje ich rezerwacji tuż przed uruchomieniem obliczeń. Podczas uruchamiania aplikacji nie należało podawać explicite parametru **-n** i **-np**, ponieważ system sam dobierał wartości na podstawie parametrów alokacji (umieszczonych w skrypcie startowym. Slurm tworzy przydział zasobów dla zadania, a następnie mpirun uruchamia zadania przy użyciu infrastruktury Slurm.

Program urachamiany był dla parametrów **-N** równego 1, 2, 4 i 8 oraz –**ntasks-per-node** równego 1.

8.5 Analiza wykonania programu przy użyciu MPI

8.5.1 Uzyskane czasy

Po uruchomieniu kodu otrzymano wyniki, które w sposób tabelaryczny przedstawione zostały w tabelach nr 9 i 11 dla serwera Honorata oraz w tabelach nr 10 i 12 dla superkomputera Ares:

Równolegle	
Czas [ns]	**Format - SELL**
2 procesy	0.0105
4 procesy	0.0083
8 procesów	0.0057

Tabela 9: Uzyskane wyniki czasu dla programu na serwerze Honorata

Źródło: opracowanie własne

Równolegle	
GFLOPS	**Format - SELL**
2 procesy	1.8638
4 procesy	2.3539
8 procesów	3.4125

Tabela 10: Uzyskane wyniki wydajnościowe implementacji mnożenia macierz-wektor dla różnych formatów przechowywania macierzy w GFLOPS dla programu na serwerze Honorata

Źródło: opracowanie własne

Równolegle	
Czas [ns]	**Format - SELL**
2 procesy	0.0097
4 procesy	0.0081
8 procesów	0.0054

Tabela 11: Uzyskane wyniki czasu dla programu na serwerze Ares

Źródło: opracowanie własne

Równolegle	
GFLOPS	**Format - SELL**
2 procesy	2.0158
4 procesy	2.4245
8 procesów	3.6255

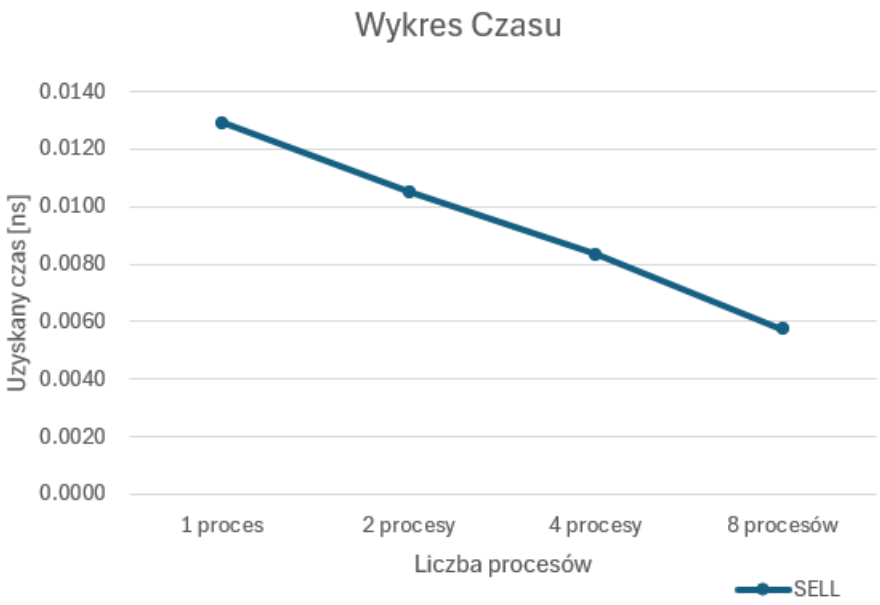
Tabela 12: Uzyskane wyniki wydajnościowe implementacji mnożenia macierz-wektor dla różnych formatów przechowywania macierzy w GFLOPS dla programu na serwerze Ares

Źródło: opracowanie własne

8.5.2 Wyniki wydajnościowe

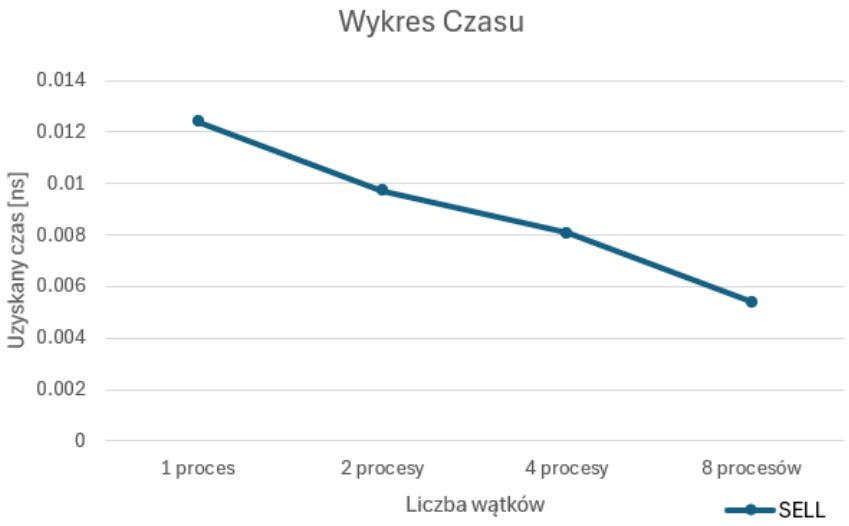
Na podstawie wyników uzyskanych w tabelach 1, 2 i 9, i 10 dla serwera Honorata oraz w tabelach 3, 4 i 11 i 12 dla superkomputera Ares stworzone zostały wykresy:

• wykres zależności czasu od liczby wątków (ilustracja nr 25 oraz nr 26),



Ilustracja 25: Wykres czasu dla programu na serwerze Honorata

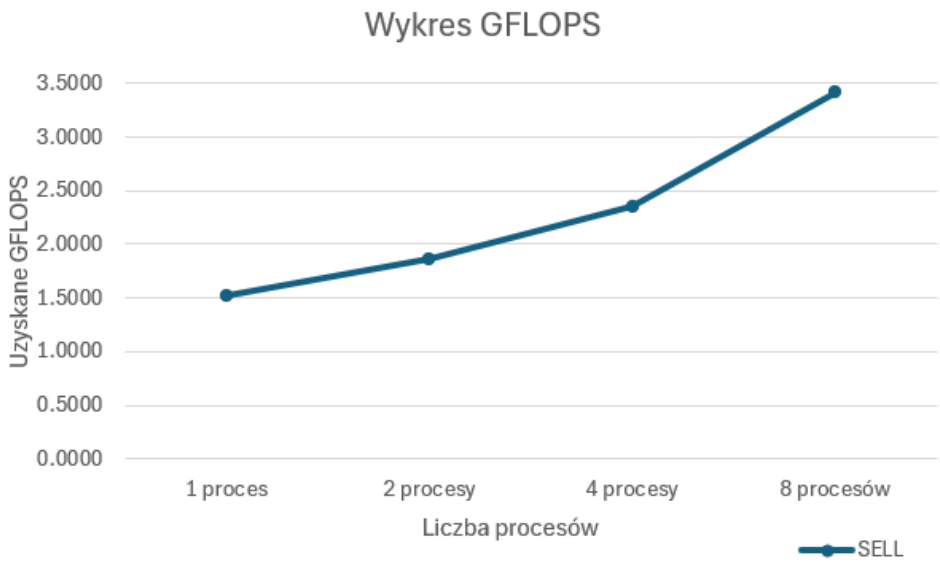
Źródło: opracowanie własne



Ilustracja 26: Wykres czasu dla programu na serwerze Ares

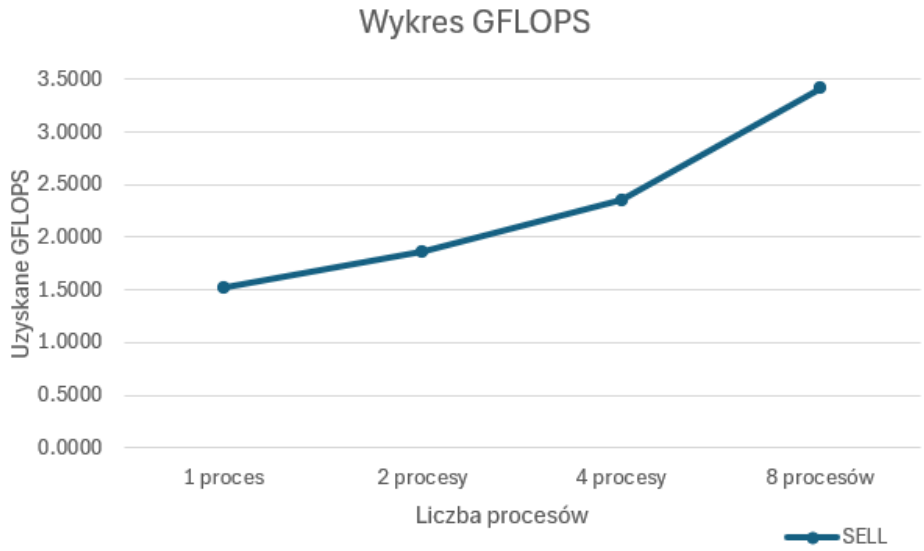
Źródło: opracowanie własne

• wykres zależności uzyskanych GFLOPS od liczby wątków (rys. 27 oraz 28),



Ilustracja 27: Wykres uzyskanych GFLOPS na serwerze Honorata

Źródło: opracowanie własne

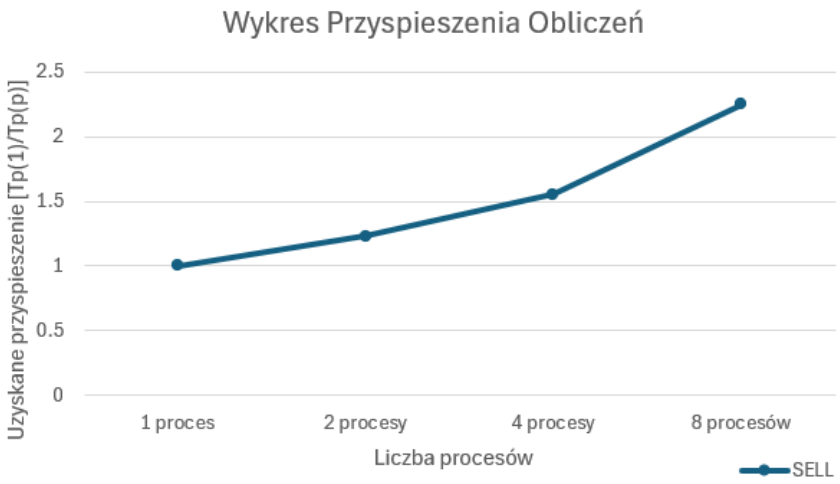


Ilustracja 28: Wykres uzyskanych GFLOPS na serwerze Ares

Źródło: opracowanie własne

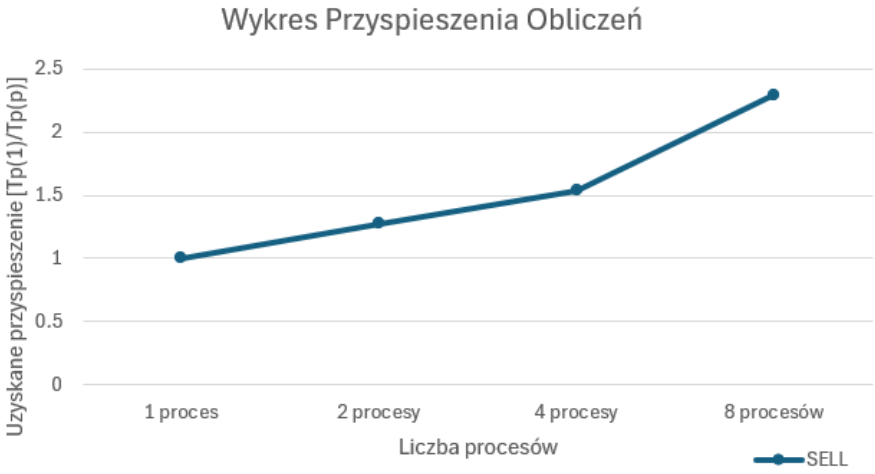
• wykres zależności uzyskanego przyspieszenia od liczby wątków (rys. 29 oraz 30).

Wartości przyspieszenia widoczne na ilustracji nr 29 oraz nr 30 zostały obliczone na podstawie wzoru S(p) = Tp(1)/Tp(p).



Ilustracja 29: Wykres przyspieszenia uzyskany na serwerze Honorata

Źródło: opracowanie własne

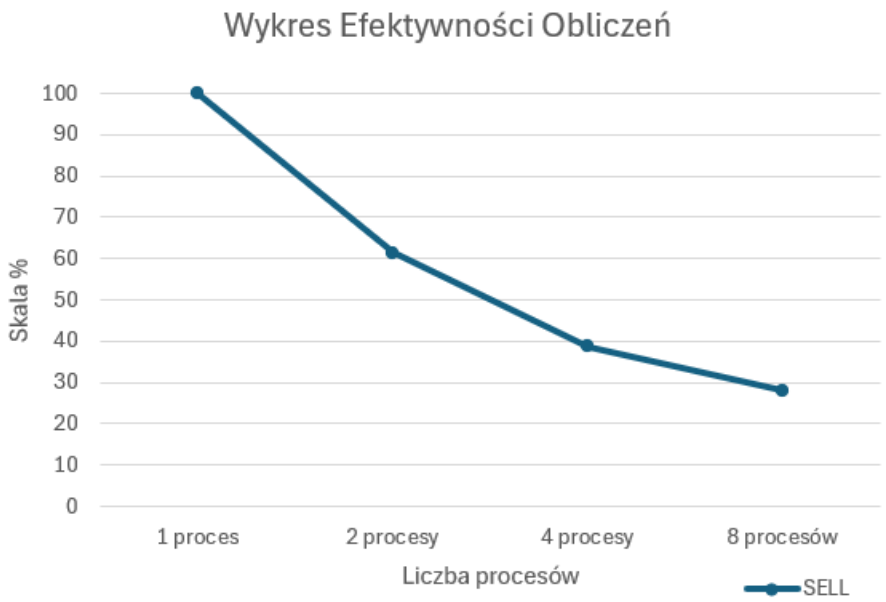


Ilustracja 30: Wykres przyspieszenia uzyskany na serwerze Ares

Źródło: opracowanie własne

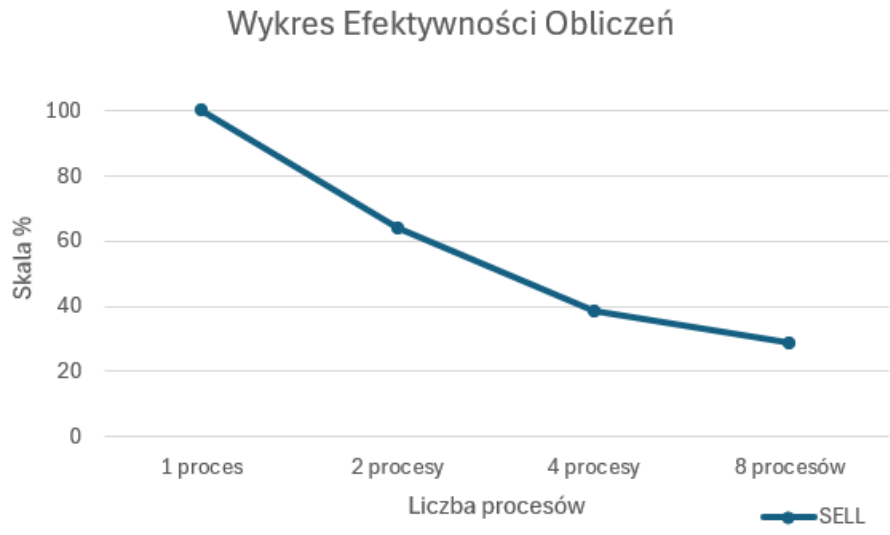
• wykres efektywności zrównoleglenia w zależności od liczby wątków (rys. 31 oraz 32).

Wartości efektywności widoczne na ilustracji nr 31 oraz nr 32 zostały obliczone na podstawie wzoru E(p) = S(p) / p.



Ilustracja 31: Wykres efektywności uzyskany na serwerze Honorata

Źródło: opracowanie własne



Ilustracja 32: Wykres efektywności uzyskany na serwerze Ares

Źródło: opracowanie własne

8.5.3 Analiza i wnioski

Analizując wyniki można zauważyć, że uzyskiwane czasy dla formatu zmniejszają się w miarę zwiększania liczby procesów, co sugeruje poprawę wydajności przy większej liczbie używanych procesów. Wykres uzyskany na serwerze Honorata jest bardziej liniowy, wyniki czasu dla superkomputera Ares mają szybszą tendencję spadkową uzyskując niższy wyniki dla 8 procesów.

W analizie wykresów GFLOPS wzrost liczby procesów powoduje stałe zwiększanie uzyskiwanych GFLOPS. Wydajność formatu SELL, w miarę zwiększania liczby wątków, na Aresie rośnie porównywalnie jak na serwerze Honorata, jednak przy 8 procesach metoda ta osiąga najwyższe wartości GFLOPS, co wskazuje na jej dobrą skalowalność i optymalność przy dużej liczbie wątków.

Analizując wykresy przyspieszeń obliczeń można zauważyć, że wyniki uzyskane na superkomputerze Ares również w tym przypadku są porównywalne co uzyskane wyniki na serwerze Honorata.

Również wykresy efektywności przedstawiają porównywalne wyniki na obu serwerach. Jednak dla 8 procesów wyniki na superkomputerze Ares są nieco wyższe.

9. Podsumowanie

Celem niniejszej pracy była analiza interakcji sprzętu i oprogramowania oraz narzędzi wykorzystywanych do badania wydajności programu realizującego algorytm mnożenia macierz-wektor dla wybranych formatów przechowywania macierzy rzadkiej. W analizie wykonano porównanie czasów i jednostki GFLOPS. Na ich podstawie wyliczone zostały miary wydajności, dzięki czemu późniejszej analizie poddane zostały uzyskane przyspieszenia i efektywności dla wszystkich badanych formatów macierzy rzadkiej. Testowanie programu przeprowadzone zostało na jednej dużej macierzy, w której liczba niezerowych elementów była większa niż 9 milionów. Dodatkowo poprzez zrealizowane w pracy testy można było przeanalizować czasy i wydajność zaimplementowanych funkcji, a tym samym stwierdzić, czy wykorzystywane specyfikacje przyniosły korzyści w usprawnieniu obliczeń.

Uzyskane wyniki dla wszystkich formatów macierzy przy użyciu specyfikacji OpenMP oraz MPI można uznać za prawidłowe ze względu na widoczny wzrost przyspieszenia dla każdego formatu. Programy nie są idealnie skalowalne, ponieważ ich przyspieszenie odbiega od przyspieszenia idealnego, $S(p) = p$. Widoczna różnica w uzyskanych czasach, wydajności i obliczonych, na podstawie czasu, przyspieszeniach i efektywności obliczeń pozwala na stwierdzenie, że rekomendowanym formatem dla obliczeń mnożenia macierz-wektor realizowanego w sposób równoległy jest format SELL realizowany wektorowo. Jednakże w celu uzyskania jak najlepszych wyników należy stosować największą badaną w pracy liczbę wątków i procesów.

W przeprowadzonych analizach fragmentów kodu asemblera wykorzystywana instrukcja FMA (`vfmadd231sd`) w głównej pętli algorytmu mnożenia macierz-wektor pozwoliła na jednoczesne wykonanie mnożenia i dodawania, co było bardziej efektywne niż wykonywanie tych operacji oddzielnie. Kompilator zdołał zoptymalizować kod minimalizując operacje pamięciowe oraz efektywnie zarządzając pętlami i indeksami, co znacząco przyspieszyło wykonanie operacji. Również w tej analizie zauważyć można było, że algorytm używający zwektoryzowanego formatu SELL przechowywania macierzy rzadkiej miał najlepiej przeprowadzoną optymalizację kodu.

Uruchomienie programu na superkomputerze oraz porównanie otrzymanych wyników wraz z wynikami z serwera Honorata pozwoliło na głębszą analizę wyników. Superkomputer Ares posiada lepsze specyfikacje sprzętowe i lepsze wskaźniki wydajności jednak uzyskane wyniki programu na superkomputerze w niektórych sytuacjach były bardziej narażone na wahania. Wybór między serwerami, do przyszłych obliczeń, może zależeć od konkretnych wymagań obciążenia,

takich jak potrzeba wyższej wydajności jednowątkowej lub większej pamięci i przepustowości sieci. Należy również wziąć pod uwagę ilość posiadanych procesów oraz wątków, których poprawne użycie może przyczynić się do uzyskania wyższych wyników.

Praca ta nie wyczerpuje w pełni realizowanego tematu. Tworzony program ma perspektywy rozwoju w wielu kierunkach, z których najważniejszymi mogą być wykorzystanie procesu wektoryzacji dla wszystkich opisywanych w pracy formatów oraz realizacja obliczeń równoległych przy użyciu większej liczby wątków i procesów. W przyszłości można również wykonać analizę wydajności programu porównując wyniki uzyskane na innych superkomputerach należących do Akademickiego Centrum Komputerowego Cyfronet AGH: Helios i Athena.

Bibliografia

[1] Adaptive Optimization of Sparse Matrix-Vector Multiplication on Emerging Many-Core Architectures, https://jianbinfang.github.io/files/2018-06-28-aspmv.pdf, [dostęp: 20.12.2021]

[2] Implementing a Sparse Matrix Vector Product for the SELL-C / SELL-C-σ formats on NVIDIA GPUs, https://icl.utk.edu/files/publications/2014/icl-utk-772-2014.pdf, [dostęp: 14.11.2024]

[3] Analiza i modelowanie wydajności obliczeniowej, http://ww1.metal.agh.edu.pl/~banas/AMWO_WWW.pdf, [dostęp: 06.01.2024]

[4] Malwina Cieśla, Mnożenie macierz-wektor dla macierzy rzadkich na procesorach

z jednostkami wektorowymi, Kraków 2022, [dostęp: 14.11.2024]

[5] Strona internetowa specyfikacji OpenMP, https://www.openmp.org/, [dostęp: 21.12.2021]

[6] Materiały dydaktyczne do przedmiotu Przetwarzanie Równoległe i Rozproszone,

http://ww1.metal.agh.edu.pl/~banas/PR/PR_W07_OpenMP_1.pdf, [dostęp: 05.01.2022]

[7] Materiały dydaktyczne do przedmiotu Przetwarzanie Równoległe i Rozproszone, http://ww1.metal.agh.edu.pl/~banas/PR/PR_W03_Wspolbieznosc.pdf, [dostęp: 27.12.2021]

[8] Internetowy podręcznik interfejsu MPI, https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, [dostęp: 28.11.2023]

[9] Materiały dydaktyczne do przedmiotu Przetwarzanie Równoległe i Rozproszone, http://ww1.metal.agh.edu.pl/~banas/PR/PR_W10_MPI_wstep.pdf, [dostęp: 08.09.2024]

[10] A Guide to Vectorization with Intel® C++ Compilers,

https://www.intel.com/content/dam/www/public/us/en/documents/guides/compiler-auto-

vectorization-guide.pdf, [dostęp: 26.12.2021]

[11] Wykład „Programowanie równoległe z wykorzystaniem współczesnych architektur komputerowych z pamięcią współdzieloną” w ramach programu Regionalna Inicjatywa Doskonałości, https://icis.pcz.pl/~lszustak/OpenMP/Temat5/Temat_nr5.pdf, [dostęp: 15.01.2024]

[12] Intel® C++ Compiler Classic Developer Guide and Reference Development Reference Guides,

https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-

and-reference/top/compiler-reference/intrinsics/intrinsics-for-intel-advanced-vector-extensions.html, [dostęp: 02.01.2022]

[13] Materiały dydaktyczne do przedmiotu Przetwarzanie Równoległe i Rozproszone, http://ww1.metal.agh.edu.pl/~banas/PR/PR_W13_Wydajnosc.pdf, [dostęp: 03.01.2024]

[14] Materiały dydaktyczne do przedmiotu Wydajność Oprogramowania, http://ww1.metal.agh.edu.pl/~banas/WO/WO_W01_Introduction.pdf, [dostęp: 03.01.2024]

[15] Materiały dydaktyczne do przedmiotu Wydajność Oprogramowania, http://ww1.metal.agh.edu.pl/~banas/WO/WO_W08_Classical_optimization.pdf, [dostęp: 16.08.2023]

[16] Internetowe materiały dotyczące perf, https://pl.linux-console.net/?p=1055#gsc.tab=0, [dostęp: 03.01.2024]

[17] Sparse Matrix-Vector Multiplication with Wide SIMD Units: Performance Models and a Unified Storage Format, adres: https://blogs.fau.de/essex/files/2012/11/SELL-C-sigma.pdf, [dostęp: 04.01.2022]

[18] Modular FEM framework ”ModFEM” for generic scientific parallel simulations / Kazimierz MICHALIK, Krzysztof BANAŚ, Przemysław PŁASZEWSKI, Paweł CYBUŁKA // Computer Science ; 2013 vol. 14 no. 3, s. 513–528. , https://www.researchgate.net/publication/307699978_Modular_Fem_Framework_Modfem_For_Generic_Scientific_Parallel_Simulations, [dostęp: 24.11.2024]

[19] Akademickie Centrum Komputerowe Cyfronet AGH, opis superkomputera Ares, https://www.cyfronet.pl/komputery/18486,artykul,superkomputer_ares.html, [dostęp: 30.07.2024]

[20] Quick Start User Guide, https://slurm.schedmd.com/quickstart.html, [dostęp: 20.11.2024]