POLITECHNIKA SZCZECIŃSKA

WYDZIAŁ INFORMATYKI

KATEDRA TECHNIK PROGRAMOWANIA

---

mgr inż. Krzysztof Siedlecki

Algorytmy wyszukiwania drobno- i gruboziarnistej równoległości w pętlach programowych z zależnościami afinicznymi

Rozprawa doktorska

PROMOTOR:

prof. dr hab. inż. Włodzimierz Bielecki

SZCZECIN 2008

Podziękowania

Składam serdeczne podziękowania wszystkim, którzy służąc mi swoją pomocą przyczynili się do powstania niniejszej pracy, a w szczególności:

- prof dr. hab. inż. Włodzimierzowi Bieleckiemu za opiekę naukową w trakcie moich studiów doktoranckich oraz cenną pomoc merytoryczną podczas realizacji pracy naukowej,
- mojej żonie Edycie za wszelką pomoc w trakcie pisania pracy, cierpliwość i wsparcie w wielu trudnych chwilach.

Spis treści

Spis rysunków

Spis tabel

Spis algorytmów

Spis symboli

$\prec$	symbol oznacza porządek leksykograficzny między dwoma instancjami instrukcji $S_1(I)$ i $S_2(J)$, taki, że wykonanie instancji instrukcji $S_1(I)$ poprzedza wykonanie instrukcji $S_2(J)$
δ	symbol oznacza wystąpienie zależności prostej (ang. *data-flow dependence*) między dwoma instancjami instrukcjami
δ^{-1}	symbol oznacza wystąpienie antyzależności (ang. *antidependence*) między dwoma instancjami instrukcjami
δ^{0}	symbol oznacza wystąpienie zależności 'po wyjściu' (ang. *output dependence*) między dwoma instancjami instrukcjami
$\exists$	ang. *exist*, kwantyfikator szczegółowy oznacza, że istnieje takie podstawienie zmiennej, że dane twierdzenie zachodzi
$\cap$	ang. *intersection*, część wspólna dwóch zbiorów lub relacji
$-$	ang. *difference*, różnica dwóch zbiorów lub relacji
$\cup$	ang. *union*, unia (suma) dwóch zbiorów lub relacji
$\neg$	ang. *inverse*, zaprzeczenie relacji, nieprawda że ...
$\forall$	ang. a*ll*, kwantyfikator ogólny oznacza, że dane twierdzenie jest prawdziwe przy dowolnej wartości zmiennej

Słownik skrótów

ATF	ang. *Affine Transformation Framework*, zbiór przekształceń afinicznych pozwalających na wykonanie wielu transformacji w sposób jenolity
UMA	ang. *uniform memory access*, architektura o jednolitym dostępie do pamięci gloalnej
NUMA	ang. *nonuniform memory Access*, architektura o niejednolitym dostępie do pamięci globalnej
COMA	ang. *cache-only memory architecture*, architektura w której pamięć globalna składa się z pamięci podręcznych procesorów
SMP	ang. *symetric multiprocessor*, system wieloprocesorowy o architekturze typu UMA
OpenMP	ang. *Open Multi-Processing*, standard programowania równoległego oparty na dyrektywach preprocesora oraz funkcjach bibliotecznych
RDG	ang. *reduced dependence graph*, zredukowany graf zależności
CRDG	ang. *connected reduced dependence graphs*, połączony zredukowany graf zależności
SCC	ang. *strongly connected component*, ściśle połączony graf
NPB	ang. *NAS Parallel Benchmarks*, zbiór programów pozwalający na zbadanie wydajności systemów jedno- i wieloprocesorowych
SPMD	ang. *Single Program Multiple Data*, pojedynczy program wiele danych
PCAM	ang. *partitioning, communication, agglomeration and mapping*, proces projektowania algorytmów równoległych złożony z czterech etapów: podział, komunikacja, aglomeracja i mapowanie

1. WSTĘP

Niniejszy rozdział stanowi wprowadzenie do problematyki poruszanej w rozprawie doktorskiej. W podrozdziale 1.1 zaprezentowano skrócony opis stanu wiedzy w zakresie poruszanego tematu oraz prowadzonych badań. Przedstawiono uzasadnienie podjętego zagadnienia, lukę poznawczą, w której zawiera się praca oraz dziedzinę nauk, do jakiej praca została zakwalifikowana. W kolejnym punkcie zdefiniowano cel pracy wraz z tezą badawczą pracy. Przedstawiono przesłanki przemawiające za wybranym kierunkiem badań oraz korzyści wynikające z tego wyboru. Dodatkowo opisano dane wejściowe i wyjściowe, zakres stosowalności, wykorzystane narzędzia oraz metodykę badań i sposób weryfikacji zaproponowanych algorytmów. Podrozdział 1.3 obejmuje strukturę pracy wraz z krótką charakterystyką poszczególnych rozdziałów.

1.1. Stan problemu

Nieustanny postęp technologiczny w dziedzinie sprzętu komputerowego oraz wysiłki ze strony twórców oprogramowania w zakresie przetwarzania równoległego skutkują popularyzacją tej dziedziny w środowiskach akademickich i biznesowych. Zapotrzebowanie na maszyny o dużej mocy obliczeniowej nieustannie wzrasta. Tendencja ta ma charakter stały i w najbliższej przyszłości będzie ulegać nasileniu. Programy dedykowane na maszyny wieloprocesorowe znacząco różnią się od programów sekwencyjnych. Zagadnienia takie jak: zarządzanie wątkami czy wyszukiwanie fragmentów kodu do zrównoleglenia nie należą do łatwych, dlatego też użytkownicy nieposiadający wiedzy z dziedziny przetwarzania równoległego i rozproszonego najczęściej nie wykorzystują w pełni mocy tkwiących w maszynach wieloprocesorowych. Tworzone przez nich programy niejednokrotnie zawierają błędy

i są nieefektywne. W związku z powyższym dla przeciętnego użytkownika bardziej naturalne i mniej skomplikowane wydaje się tworzenie rozwiązań sekwencyjnych w przeciwieństwie do rozwiązań równoległych. Przekształcanie programów sekwencyjnych na ich odpowiedniki równoległe powinno być w jak największym stopniu zautomatyzowane np. przy pomocy dedykowanych kompilatorów wykorzystujących odpowiednie techniki analizy i transformacji kodu sekwencyjnego na jego równoległy odpowiednik.

Statystycznie najwięcej czasu, jaki procesor potrzebuje do realizacji zadania zajmuje wykonanie instrukcji zawartych w pętlach programowych, w szczególności w pętlach składających się z wielu iteracji. Ze względu na czas wykonania, transformacja tych obszarów programu może przynieść największe korzyści. Przekształcenie pętli programowych do ich równoległej postaci należy poprzedzić analizą zależności, której zadaniem jest określenie, w jaki sposób iteracje bądź instrukcje pętli zależą od siebie w trakcie wykonania. Informacje uzyskane w wyniku przeprowadzonej analizy pozwalają wyeliminować zależności poprzez wykonanie jednej lub sekwencji odpowiednich transformacji. Kolejnym krokiem jest podział przestrzeni iteracji pętli na wiele procesorów (dla maszyn równoległych) lub węzłów (dla systemów rozproszonych). W związku ze złożonością procesu analizy zależności występujących w pętlach jak i generacji kodu, wydaje się niezbędnym stworzenie algorytmów umożliwiających automatyzację procesu transformacji pętli sekwencyjnych na pętle równoległe. Zaproponowano wiele rozwiązań umożliwiających wykonanie automatycznej transformacji pętli [7], [89], [90], [35], [27], [31], [32], [68], [67], [69], jednak żadne z nich nie pozwala na wyszukanie pełnej równoległości pozbawionej synchronizacji.

Poniższa praca prezentuje autorskie algorytmy pozwalające na uzyskanie niezależnych fragmentów kodu dla pętli programowych w sytuacji, kiedy inne techniki zawodzą. Algorytmy te mogą być również zastosowane dla pętli dowolnie zagnieżdżonych w przypadku, kiedy górna granica pętli podana jest jako parametr.

Wyszukiwanie niezależnych fragmentów kodu pozwala na uzyskanie skalowanego rozwiązania dla maszyn równoległych o różnej liczbie jednostek przetwarzających. Dodatkowo umożliwia osiągnięcie wzrostu wydajności - dzięki zwiększeniu lokalności danych (dla komputerów jednoprocesorowych) oraz redukcję wymaganej pamięci, co w systemach mobilnych i osadzonych wiąże się ze zmniejszeniem zapotrzebowania na pobór prądu). Wyznaczenie niezależnych fragmentów kodu pozwala na zastosowanie dowolnej techniki zwiększającej lokalność kodu np. blokowanie [1]. W takim przypadku wynik uzyskany w jednej iteracji musi być przechowywany w buforze do

czasu ostatniego użycia przez instrukcje z innej iteracji. Należy pamiętać jednocześnie, iż rozmiar pamięci niezbędnej do przechowania wyników uzyskanych i używanych w różnych iteracjach określony jest ilością iteracji, przez które musi być przechowywany.

Główna idea proponowanego rozwiązania bazuje na podziale przestrzeni iteracji pętli [80]. Początkowo przestrzeń pętli dzielona jest na zbiór iteracji zależnych oraz niezależnych. Następnie zbiór zależnych iteracji przeszukiwany jest w celu uzyskania niezależnych fragmentów kodu. Uzyskane fragmenty przestrzeni iteracji mogą być wykonane niezależnie bez konieczności synchronizacji pomiędzy nimi. Prezentowane podejście bazuje na dokładnej analizie zależności opisanych przy pomocy relacji i zbiorów krotek [58].

Zgodnie z powyższym opisem, proces tworzenia wydajnego oprogramowania jest czynnością czasochłonną i skomplikowaną, wymagającą od twórcy gruntownej wiedzy z dziedziny przetwarzania równoległego. Główne czynniki, na jakie należy zwrócić uwagę podczas tworzenia/projektowania programu równoległego to:

- architektura systemu,
- podział problemu na mniejsze zadania (dekompozycja / partycjonowanie) - dekompozycja domeny problemu, dekompozycja funkcjonalna,
- komunikacja – synchroniczna oraz asynchroniczna, koszt komunikacji, opóźnienia oraz przepustowość,
- synchronizacja – bariera, semafor, flaga,
- zależności danych – zależności między danymi, relacje zależności,
- zarządzanie obciążeniem obliczeń,
- ziarnistość podziału – drobnoziarnistość, gruboziarnistość.

W niniejszej pracy główny nacisk położony został na podział przestrzeni iteracji pętli umożliwiający wyszukanie jak największej liczby niezależnych fragmentów kodu. Podział taki bezpośrednio związany jest z ziarnistością kodu, która określa liczbę oraz wielkość zadań, na jakie problem został podzielony. Pozwala to na uzyskanie zarówno drobnoziarnistości jak i gruboziarnistości poprzez zastosowanie aglomeracji (łączenie niezależnych ziaren w większe). Wiele istniejących transformacji dedykowanych jest dla określonego typu pętli programowych np. transformacje unimodularne [7], [89], [90] - pętle idealnie zagnieżdżone, czy też transformacje oparte na rekurencji Hamiltona (ang. *Hamiltonian Recurrence*) [35] - pętle o stałym wektorze dystansu. Przekształcenia oparte na mapowaniu przestrzeni iteracji przy pomocy transformacji afinicznych (ang. A*ffine Transformation Framework*) uważane są aktualnie za transformacje

najmocniejsze pozwalające na wyznaczenie równoległości pozbawionej synchronizacji [27], [31], [32], [68], [67], [69]. Jednakże dla pętli niejednorodnych transformacje te nie umożliwiają wyznaczenia harmonogramowania pozwalającego na uzyskanie maksymalnej liczby niezależnych fragmentów kodu [27]. Celem prowadzonych badań było wypełnienie luki poznawczej, poprzez opracowanie algorytmów umożliwiających wyszukanie maksymalnej liczby niezależnych fragmentów kodu dla pętli dowolnie zagnieżdżonych.

Niniejsza praca leży w dziedzinie nauk technicznych i dotyczy dyscypliny informatyka. Problematyka w niej poruszana mieści się w zakresie przetwarzania równoległego, analizy zależności, transformacji pętli programowych oraz technik kompilacji.

Wyniki uzyskane w ramach prowadzonych badań dotyczą przekształceń pętli programowych z wykorzystaniem analizy zależności. Wyniki zostały przedstawione w autorskich publikacjach w czasopismach i na konferencjach polskich [19], [17], [11], [18] oraz światowych [15], [14], [13], [10], [16], [12]. Stworzone algorytmy stanowią rozwinięcie zagadnień związanych z automatycznym zrównolegleniem pętli sekwencyjnych dla kompilatorów ogólnego przeznaczenia.

1.2. Cel i teza badawcza pracy

Aktualnie dostępne algorytmy umożliwiające transformacje pętli programowych, pozwalają na wyznaczenie równoległości w ograniczonym stopniu i zakresie. Najczęściej ograniczenia te dotyczą struktury pętli, typu występujących zależności jak również poziomu, na jakim rozpatrywana jest równoległość. W odniesieniu do struktury pętli dziedzina problemu może zostać zawężona do: pętli idealnie zagnieżdżonych, pętli nie zawierających instrukcji sterujących lub pętli o stałym kroku. W drugim przypadku zawężenie dziedziny problemu dotyczy typu występowania zależności, np. akceptowane są tylko zależności jednorodne (stały wektor dystansu) lub też zależności określonego typu (proste, odwrotne lub po-wyjściu). Ostatni typ ograniczeń dotyczy składowej pętli, traktowanej jako część niepodzielna (atomowa). Wiele rozwiązań ogranicza się do transformacji pętli na poziomie pojedynczej iteracji - instrukcje (instancje instrukcji) nie są rozpatrywane osobno.

Nałożenie jednego lub kombinacji powyższych ograniczeń umożliwia zawężenie problemu do wybranej klasy pętli, a tym samym upraszcza proces analizy i transformacji pętli. Według autora niniejszej pracy prawdziwym wyzwaniem jest opracowanie rozwiązania pozwalającego na transformację pętli dowolnie

zagnieżdżonych zarówno na poziomie iteracji jak i instrukcji przy jednoczesnej minimalizacji dodatkowych ograniczeń. Takie podejście będzie pierwszym krokiem do opracowania efektywnego kompilatora ogólnego przeznaczenia, umożliwiającego w automatyczny sposób transformację kodu sekwencyjnego na jego równoległy odpowiednik.

Celem niniejszej pracy jest opracowanie oraz weryfikacja algorytmów wyszukiwania drobno- i gruboziarnistej równoległości w pętlach programowych z zależnościami afinicznymi.

Badania prowadzone w ramach pracy służą do wykazania prawdziwości następującej tezy badawczej.

Możliwe jest opracowanie algorytmów do automatycznego wyszukiwania drobno- i gruboziarnistej równoległości w pętlach programowych o złożoności pozwalającej na ich zastosowanie w akademickich oraz przemysłowych kompilatorach optymalizujących.

W zakresie przetwarzania równoległego ziarnistość obliczeń (ang. *granularity*) [86] wyznacza stosunek czasu obliczeń do czasu potrzebnego na synchronizację i komunikację. W przypadku, gdy czas obliczeń jest mniejszy od czasu zarządzania wątkami (stosunek czasów jest mniejszy lub zbliżony do 1) mówimy o drobnoziarnistej równoległości (ang. *fine-grained parallelism*). Drobnoziarnistość, przy prawidłowym mapowaniu zadań do poszczególnych procesorów, umożliwia maksymalną redukcję bezczynności jednostek przetwarzających oraz efektywne zarządzanie obciążeniem systemu wieloprocesorowego. Jednak koszt zarządzania wieloma wątkami może być zbyt duży i wpłynąć na obniżenie wydajności przetwarzania. W związku z tym podział kodu na drobne ziarna preferowany jest w systemach, w których koszt zarządzania wątkami jest niewielki, a balansowanie obciążeniem ma zasadnicze znaczenie dla wydajności całego systemu. Jeżeli natomiast czas obliczeń jest większy od czasu potrzebnego na zarządzanie wątkami to mamy do czynienia z gruboziarnistą równoległością (ang. *coarse-grained parallelism*). Zaletą takiego podziału jest możliwość zwiększenia wydajności algorytmu poprzez: redukcję czasu potrzebnego na synchronizację i komunikację, zwiększenie lokalności kodu oraz zmniejszenie zapotrzebowania na pamięć. Ze względu na duże czasy przestojów i różną wielkość ziaren główną niedogodnością jest trudność w balansowaniu obciążeniem na poszczególnych jednostkach przetwarzających. Gruboziarnisty podział preferowany jest w systemach, w których koszt komunikacji między jednostkami przetwarzającymi jest znaczny (np. systemy rozproszone).

Do głównych korzyści wynikających z opracowania algorytmów automatycznego wyznaczania drobno- i gruboziarnistej równoległości można zaliczyć: zwiększenie skalowalności kodu, minimalizacja kosztów związanych z zarządzaniem wątkami, zmniejszenie zapotrzebowania na pamięć niezbędną do rozwiązania określonego problemu, redukcja kosztów i błędów związanych z ręczną transformacją kodu sekwencyjnego na równoległy odpowiednik.

Za podjęciem zagadnienia będącego tematem niniejszej pracy przemawiają przesłanki naukowe i ekonomiczne. Biorąc pod uwagę przesłanki naukowe ważnym wydaje się udzielenie odpowiedzi na pytanie, jaka jest maksymalna liczba niezależnych fragmentów kodu dostępna w zadanej pętli programowej. Niezależny fragment kodu jest to zbiór instrukcji powiązanych ze sobą poprzez odwołania do wspólnych danych, a jednocześnie nieposiadający zależności miedzy innymi fragmentami kodu. Każdy z takich fragmentów (zwany niezależnym wątkiem) może zostać wykonany niezależnie od innych fragmentów na oddzielnych jednostkach przetwarzających. Równocześnie zależność poszczególnych instrukcji w pojedynczym fragmencie, pozwala na zwiększenie lokalności kodu. Każda kolejna instrukcja bazuje na wyniku instrukcji poprzedzającej. Zachowując kolejność wykonywania instrukcji możliwe jest zapisanie bieżącego wyniku w szybkiej pamięci podręcznej procesora, na potrzeby kolejnych obliczeń. Wpływa to na redukcję czasu wykonania programu. Uzupełniając przesłanki naukowe o aspekt ekonomiczny daje się zauważyć fakt, iż automatyzacja procesu transformacji kodu sekwencyjnego na odpowiednik równoległy pozwala na zmniejszenie kosztów oraz redukcję błędów ludzkich związanych z ręcznym przekształcaniem kodu. Minimalizując udział użytkownika w procesie zrównoleglenia kodu przyczyniamy się do wzrostu popularności i wykorzystana systemów wieloprocesorowych w rozwiązaniach komercyjnych, umożliwiając tym, samym skorzystanie z maszyn i środowisk wieloprocesorowych szerszemu gronu użytkowników.

Ze względu na możliwość transformacji pętli dowolnie zagnieżdżonych, opracowane algorytmy mają szeroki zakres stosowalności. Zaproponowane przekształcenia pętli znajdują swoje zastosowanie w procesie transformacji dowolnych programów naukowych jak również komercyjnych. W zależności od uzyskanej ziarnistości, wygenerowany kod może zostać z powodzeniem wykonany na maszynach wieloprocesorowych różnego typu (systemy rozproszone lub systemy ze wspólna pamięcią wykorzystujące w swojej architekturze procesory jedno i wielordzeniowe). Algorytmy dedykowane dla pętli idealnie zagnieżdżonych pozwalają na wyznaczenie kodu o większym ziarnie w stosunku do algorytmów dla pętli dowolnie

zagnieżdżonych. Różnica ziarnistości uzyskiwanego kodu w zależności od zastosowanego algorytmu wynika głównie z tego, iż algorytmy 3.1 – 3.5 analizują pętlę na poziomie iteracji, natomiast w algorytmach 3.6 – 3.9 analiza prowadzona jest na poziomie pojedynczych instrukcji.

Dane wejściowe algorytmów wyszukiwania drobno i gruboziarnistej równoległości stanowi kod źródłowy zapisany w języku Petit. W celu umożliwienie konwersji pętli programowych zapisanych w języku Fortran wykorzystany został translator Fortran-to-Petit. Wyjściem algorytmów, po uprzednim wykryciu i redukcji zależności oraz wyznaczeniu zbiorów zawierających niezależne iteracji/instancji instrukcji, jest kod źródłowy reprezentujący równoległość. Po wybraniu docelowego środowiska (OpenMP, Posix Threads, MPI, PVM) może zostać przekształcony do odpowiedniej postaci.

Do wyznaczenia zależności wykorzystano narzędzie Petit [57]. Modyfikacje na relacjach (wyjście programu Petit) oraz generację kodu przeprowadzono przy pomocy pakietu Omega Calculator [59]. Powyższe narzędzie stanowią integralną część projektu Omega opracowanego na Uniwersytecie Maryland (University of Maryland). Badania przeprowadzone były na maszynie wieloprocesorowej z pamięcią dzieloną SGI Altix 3700 (Poznańskie Centrum Superkomputerowo-Sieciowe) [46], przy wykorzystaniu kompilatora C++ firmy Intel w wersji 9.0 [52], [53]. Kod równoległy zapisany został zgodnie ze standardem programowania OpenMP [75].

Zaproponowane algorytmy zweryfikowano pod względem poprawności i jakości uzyskiwanych wyników. Uzyskane wyniki dla wersji wielowątkowych (ilość wątków: 2, 4, 8, 16; górne granice pętli: 64, 128, 256, 512) porównano z wersja sekwencyjną, co pozwoliło na ustalenie poprawności oraz określenie jakości uzyskanych wyników (przyspieszenie, efektywność). Ilość operacji wykonanych w pojedynczym teście wahała się od 64 iteracji (dla niezagnieżdżonej pętli) do 512^4 iteracji (dla pętli poczwórnie zagnieżdżonej) dla każdej instrukcji pętli.

1.3. Struktura pracy

Praca podzielona została na sześć rozdziałów. Zawartość merytoryczna każdego z nich opisana została poniżej.

Pierwszy rozdział stanowi wprowadzenie do tematyki pracy. Przedstawiono w nim uzasadnienie wyboru tematu, tło omawianych zagadnień, cel, tezę badawczą oraz strukturę pracy.

W rozdziale 2 przedstawiono zagadnienia z dziedziny przetwarzania równoległego i rozproszonego. Omówione zostały najbardziej popularne architektury systemów równoległych wraz ze środowiskami programistycznymi umożliwiającymi tworzenie na nich oprogramowania równoległego. Wyjaśniono podstawowe pojęcia związane z analizą zależności, przedstawiono wskaźniki wykorzystywane do określania jakości kodu jak i samej architektury systemów równoległych.

Kolejny rozdział zawiera opis zaproponowanych algorytmów wraz z przykładami obrazującymi ich działanie. Algorytmy te podzielono na dwie kategorie w zależności od typu pętli. Dodatkowo każdy zbiór algorytmów poprzedzono niezbędną wiedzą umożliwiającą zrozumienie zaproponowanego rozwiązania.

W rozdziale 4 zaprezentowano badania eksperymentalne przeprowadzone przy pomocy autorskiego narzędzia, opracowanego na potrzeby badań zgodnie z zaproponowanymi algorytmami opisanymi w poprzednim rozdziale. Rozdział kończy podsumowanie wraz z wnioskami wynikającymi z pracy.

Rozdział 5 zawiera prezentację pokrewnych prac dotyczących tematyki poruszanej w niniejszej dysertacji. Dodatkowo w rozdziale tym przedstawiono przykład pętli, dla której transformacje afiniczne (jedne z mocniejszych transformacji) zawodzą, natomiast zaproponowane algorytmy powalają na uzyskanie niezależnych fragmentów kodu, gotowych do zrównoleglenia.

Ostatni rozdział zawiera wnioski końcowe dotyczące realizacji podjętych w pracy zagadnień.

2. PRZETWARZANIE RÓWNOLEGŁE – PRZEGLĄD ZAGADNIENIA

Nieustanny wzrost ilości danych koniecznych do przetworzenia wiąże się nierozerwalnie z ciągłym wzrostem zapotrzebowania na moc obliczeniową systemów informatycznych. Moce obliczeniowe pojedynczej jednostki przetwarzającej w wielu dziedzinach nauki, przemysłu, jak również w codziennym życiu, stały się niewystarczające. Jednym z bardziej popularnych rozwiązań problemu braku mocy obliczeniowej jest udostępnienie użytkownikowi systemów posiadających więcej niż jedną jednostkę przetwarzającą. Architektury takich systemów mogą różnić się od siebie w sposób zasadniczy, co skutkuje odmiennym podejściem do modelu programowania. Podstawową różnicę pomiędzy programami dedykowanymi na maszyny wieloprocesorowe, a programami sekwencyjnymi stanowi liczba wątków odpowiedzialna za rozwiązanie określonego problemu. W programach sekwencyjnych obliczenia realizowane są przez pojedynczy wątek, natomiast w programach dedykowanych na maszyny równoległe problem rozwiązywany jest przez wiele wątków mapowanych na więcej niż jedną jednostkę przetwarzającą. Podział problemu na mniejsze zadania, przydzielane do niezależnych wątków, wiąże się z koniecznością uwzględnienia wielu dodatkowych czynników takich jak:

- docelowa architektura,
- ziarnistości kodu,
- zależności między danymi,
- mapowanie danych.

W niniejszym rozdziale przedstawiona została architektura maszyn równoległych oraz modele programowania dedykowane dla poszczególnych architektur. Dodatkowo omówione zostały podstawowe pojęcia dotyczące reprezentacji i analizy zależności,

opisano przykładowe transformacje oraz przedstawiono wskaźniki wykorzystywane w systemach wieloprocesorowych (przyśpieszenie, efektywność, itp.).

2.1. Architektura maszyn równoległych

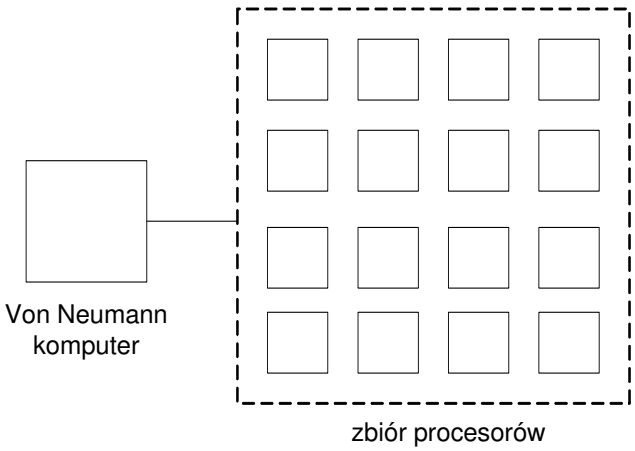
Zgodnie z klasyfikacja zaproponowaną przez Flynna w 1966 r., różne architektury komputerów możemy sklasyfikować ze względu na rodzaj strumienia informacji. Strumienie informacji wpływające do procesora to: rozkazy oraz dane. Strumień rozkazów zdefiniowany jest, jako sekwencja instrukcji wykonywanych przez jednostkę przetwarzającą. Strumień danych zdefiniowany jest, jako przepływ danych występujący między jednostką przetwarzającą a pamięcią. Zgodnie z klasyfikacją Flynna obydwa strumienie mogą być pojedyncze lub wielorakie. Na podstawie powyższego wyróżnia się cztery rozdzielne kategorie architektur komputerów [29]:

- pojedyncze rozkazy pojedyncze dane (ang. *single-instruction single-date streams* - SISD)
- pojedyncze rozkazy wiele danych (ang. *single-instruction multiple-date* - SIMD)
- wiele rozkazów pojedyncze dane (ang. *multiple-instruction single-data* - MISD)
- wiele rozkazów wiele danych (ang. *multiple-instruction multiple-data* - MIMD)

Konwencjonalne jednoprocesorowe komputery von Neumanna sklasyfikowane są jako systemy SISD. Komputery równoległe sklasyfikowane są jako maszyny typu SIMD lub MIMD. W przypadku, gdy istnieje jedna jednostka zarządzająca i wszystkie procesory wykonują ten sam rozkaz w sposób zsynchronizowany, maszyna sklasyfikowana jest jako SIMD. W przypadku równoległej maszyny typu MIMD, każdy z procesorów zawiera własną jednostkę sterującą i może wykonywać różne instrukcje na różnych danych. W architekturze MISD, ten sam strumień danych przepływa przez procesory wykonujące różne strumienie rozkazów. W praktyce nie istnieje maszyna spełniająca założenie kategorii MISD.

W modelu SIMD komputer równoległy składa się z dwóch głównych części: z komputera zarządzającego (najczęściej komputer w rozumieniu maszyny von Neumanna) oraz ze zbioru synchronizowanych jednostek przetwarzających, mogących wykonywać te same operacje na różnych danych (rys. 2.1). Każdy z procesorów posiada niewielką ilość pamięci, w której rozproszone dane są przechowywane w trakcie przetwarzania równoległego. Procesory połączone są do szyny pamięci komputera zarządzającego. W związku z tym komputer zarządzający ma dostęp do pamięci lokalnej każdego procesora. Aplikacje wykonywane są sekwencyjnie przez komputer zarządzający, przy jednoczesnym wysyłaniu komend do zbioru procesorów w

celu wykonania instrukcji SIMD w sposób równoległy. W architekturze SIMD, równoległość uzyskiwana jest poprzez wykonywanie identycznych instrukcji na dużym zbiorze danych.

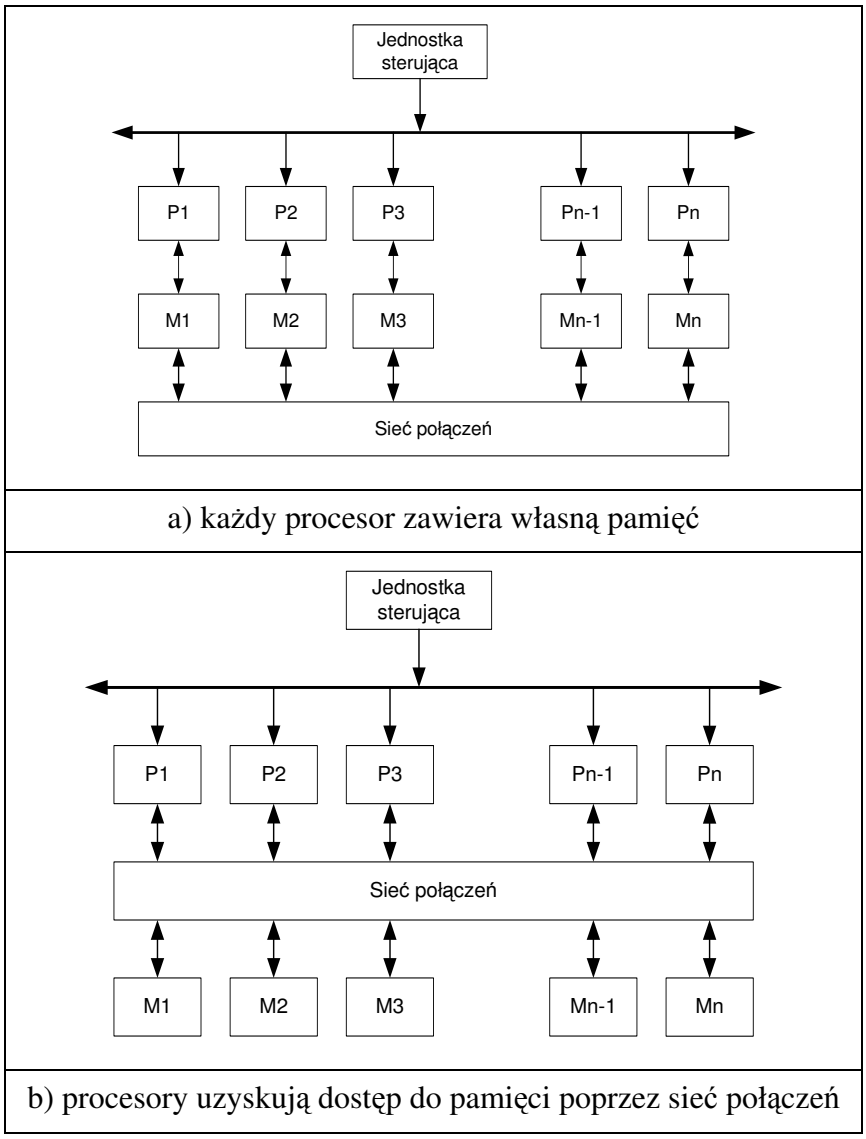


Rysunek 2.1. Architektura komputera równoległego typu SIMD [29]

Wyróżnia się dwie główne konfiguracje maszyn równoległych typu SIMD. Na rys. 2.2a każdy procesor posiada własną pamięć lokalną. Procesory mogą komunikować się ze sobą poprzez sieć połączeń. Jeśli sieć połączeń uniemożliwia uzyskanie bezpośredniego połączenia między dwoma procesorami, para procesorów może wymienić dane poprzez procesor pośredniczący. Na rys. 2.2b procesory i moduły pamięci komunikują się między sobą poprzez sieć połączeń. Dwa procesory przesyłają między sobą dane poprzez moduły pamięci.

W architekturze typu MIMD system równoległy zbudowany jest z wielu procesorów i wielu modułów połączonych ze sobą poprzez sieć połączeń. W modelu tym wyróżnia się dwie podstawowe architektury: z pamięcią dzieloną lub z pamięcią rozproszoną (przesyłaniem komunikatów). W architekturze z pamięcią dzieloną procesory wymieniają się informacjami poprzez centralną pamięć dzieloną. W systemach z pamięcią rozproszoną procesory komunikują się między sobą za pośrednictwem sieci połączeń poprzez system przesyłania komunikatów.

Preferowany model ziarnistości zależy głównie od sposobu komunikacji między poszczególnymi jednostki przetwarzającymi. Dla systemów z pamięcią rozproszoną, zarówno typu SIMD jak i MIMD, ze względu na koszt komunikacji między procesorami, preferowana jest gruboziarnista równoległość. W przypadku architektury z pamięcią dzieloną, gdzie koszt dostępu do dowolnej komórki pamięci jest stały dla dowolnego procesora, opłacalne jest zastosowanie zarówno grubo- jak i drobnoziarnistej równoległości.



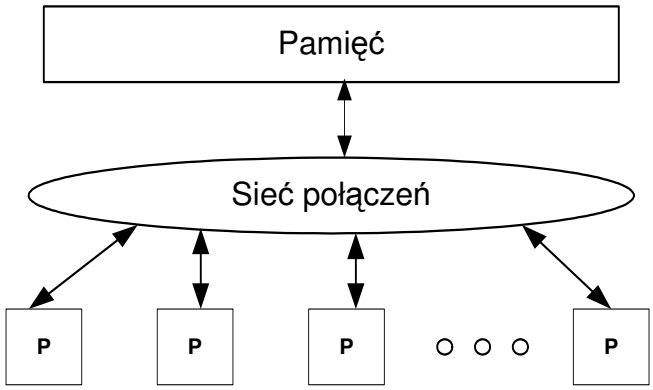
Rysunek 2.2. Architektura maszyn równoległych typu SIMD [29]

2.1.1. Systemy z pamięcią współdzieloną

W systemach równoległych z pamięcią dzieloną komunikacja między poszczególnymi zadaniami, wykonywanymi na różnych procesorach, odbywa się poprzez operację odczytu i zapisu do wspólnej globalnej pamięci [28], [62]. Systemy z pamięcią dzieloną zbudowane są ze zbioru niezależnych procesorów, zbioru modułów pamięci oraz sieci połączeń (patrz rys. 2.3).

Ze względu na sposób dostępu procesorów do globalnej pamięci, systemy z pamięcią dzieloną dzielimy na trzy główne kategorie:

- UMA (ang. *uniform memory access*),
- NUMA (ang. *nonuniform memory access*),
- COMA (ang. *cache-only memory architecture*).



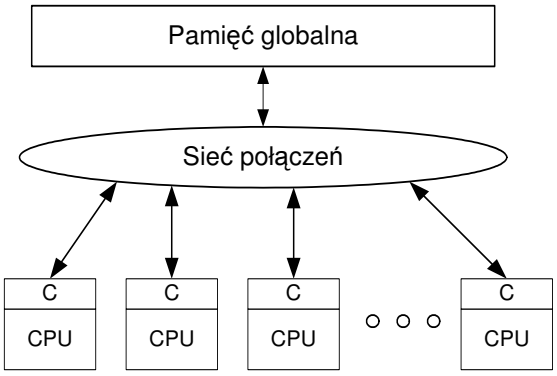
Rysunek 2.3. Schemat budowy systemów wieloprocesorowych z pamięcią dzieloną[29]

W architekturze UMA dostęp do pamięci dzielonej przez każdy z procesorów odbywa się poprzez sieć połączeń w taki sam sposób jak dostęp do pamięci poprzez pojedynczy procesor. Ze względu na to, iż czas dostępu do każdej komórki pamięci jest taki sam dla wszystkich procesorów, system taki nazywany jest również symetrycznym multiprocesorem SMP (ang. *symetric multiprocessor*). Budowa typowego systemu o architekturze SMP przedstawiony został na rys. 2.4. Załadowanie instrukcji oraz danych do pamięci podręcznej cache (C) umożliwia zredukowanie obciążenia szyny łączącej pamięć z procesorami. W sprzyjających warunkach wykorzystanie sieci połączeń będzie bliskie zeru, jeśli możliwe jest przechowanie wszystkich instrukcji oraz danych niezbędnych do wykonania obliczeń w pamięci podręcznej procesorów.

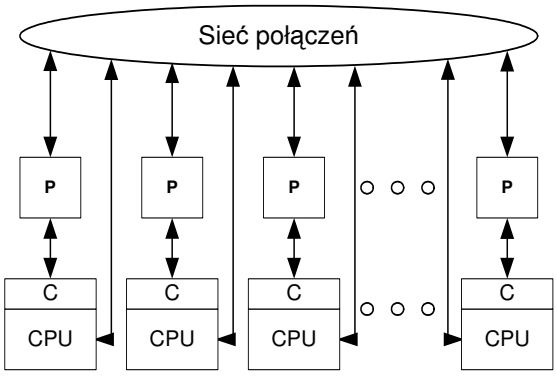
Nieco odmienne podejście do pamięci globalnej zastosowano w systemach NUMA (patrz rys. 2.5), każdy z procesorów posiada część pamięci globalnej. Podobnie jak w architekturze UMA, adresacja pamięci odbywa się w tej same przestrzeni adresowej [63]. Każdy z procesorów może odwołać się do dowolnej komórki pamięci przy użyciu rzeczywistego adresu. Jednakże czas dostępu, w przeciwieństwie do architektury UMA, zależny jest od odległości między pamięcią a procesorem. W związku z powyższym czas odwołania się do dowolnej komórki pamięci nie jest stały pomimo zachowania jednolitej przestrzeni adresowej.

W architekturze COMA, podobnie do architektury NUMA, każdy procesor posiada pewną część pamięci dzielonej (patrz rys. 2.6). Jednakże w tym przypadku pamięć dzielona składa się jedynie z pamięci podręcznej każdego procesora. Pamięć systemu nie jest zorganizowana w sposób hierarchiczny, przestrzeń adresowa obejmuje wszystkie dostępne moduły pamięci cache w systemie. W architekturze COMA występuje również słownik (na rys. 2.6 oznaczony jako D - ang. *cache directory*)

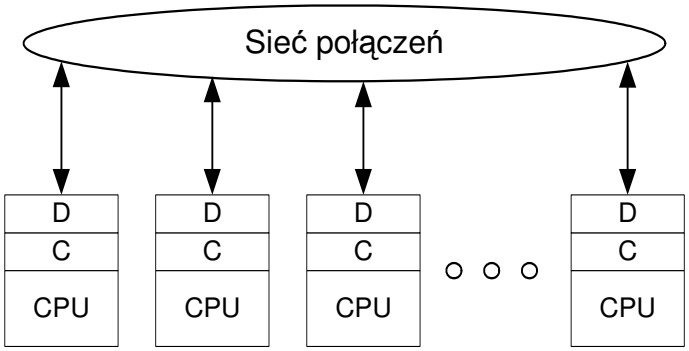
pośredniczący między odwołaniami do fragmentów pamięci znajdujących się przy innych jednostkach obliczeniowych [29].



Rysunek 2.4. Architektura UMA [29]



Rysunek 2.5. Architektura NUMA [29]

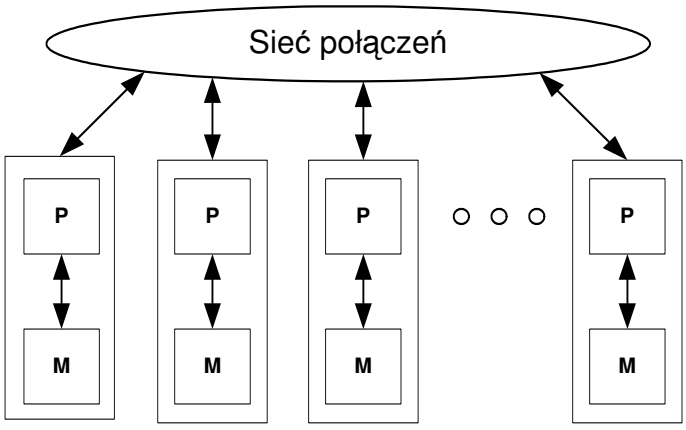


Rysunek 2.6. Architektura COMA [29]

W systemach z pamięcią dzieloną, ze względu na sposób dostępu do dowolnej komórki pamięci przez jednostki przetwarzające, możliwe jest zastosowanie zarówno drobno- jak i gruboziarnistej równoległości. Jednakże w architekturze UMA, gdzie czas dostępu do pamięci dla dowolnego procesora jest stały, zastosowanie drobnoziarnistej równoległości będzie bardziej efektywne niż w architekturach NUMA lub COMA. Z drugiej strony, jeśli wszystkie potrzebne instrukcje oraz dane nie przekroczą wielkości pamięci P dla systemu NUMA (rys. 2.5) to zastosowanie gruboziarnistej równoległości przyniesie większe korzyści niż dla architektury UMA. W takim przypadku możliwe jest zredukowanie wykorzystania sieci połączeń do niezbędnego minimum, przy jednoczesnej maksymalizacji wykorzystania pamięci P. W architekturze NUMA zwiększenie wykorzystania pamięci P przez procesor bezpośrednio z nią połączony (rys. 2.5) pozwala w znacznym stopniu zredukować czas oczekiwania procesora na niezbędne dane.

2.1.2. Systemy z pamięcią rozproszoną

Systemy z pamięcią rozproszoną, w odróżnieniu od systemów z pamięcią dzieloną, nie posiadają pamięci globalnej. Systemy te, zwane inaczej systemami z przekazywaniem komunikatów, tworzą klasę multiprocesorów, gdzie poszczególne węzły obliczeniowe mają bezpośredni dostęp jedynie do pamięci umiejscowionej na danej jednostce przetwarzającej [29], [62]. W związku z tym niezbędne jest przenoszenie danych między pamięcią jednostki X, a pamięcią jednostki Y. Komunikacja jak i przesyłanie danych w systemie odbywa się zazwyczaj przez parę instrukcji wyślij (ang. *send*) i odbierz (ang. *receive*), które muszą być umieszczone wewnątrz aplikacji przez programistę. Rysunek 2.7 przedstawia schemat systemu z przesyłaniem komunikatów.



Rysunek 2.7. Budowa systemu z pamięcią rozproszoną [29]

System rozproszony może składać się z dowolnej liczby węzłów, gdzie każdy z nich zawiera przynajmniej jeden procesor (P) oraz pamięć (M) z dostępem w obrębie lokalnej przestrzeni adresowej. Procesory komunikują się między sobą poprzez sieć połączeń – zazwyczaj jest to sieć statyczna. W systemach z pamięcią rozproszoną konieczne jest uwzględnienie dwóch podstawowych wskaźników: przepustowość łączy (ang. *link bandwidth*) oraz opóźnienie sieci (ang. *network latency*). Przepustowość łączy zdefiniowana jest jako liczba bitów jaką można wysłać w jednostce czasu (np. bit/sek), natomiast opóźnienie sieci jest to czas potrzebny na przesłanie pełnego komunikatu.

W systemach z przesyłaniem komunikatów, ze względu na stosunkowo duży koszt komunikacji między poszczególnymi węzłami, preferowana jest gruboziarnista równoległość. Jeśli węzły obliczeniowe posiadają więcej niż jedną jednostkę przetwarzającą np. budowa zgodna z architekturą UMA lub NUMA, możliwe jest zastosowanie drobnoziarnistej równoległości w obrębie pojedynczego węzła. Systemy

z przesyłaniem komunikatów, w których pojedyncze węzły zbudowane są z systemów z pamięcią dzieloną nazywane są systemami równoległymi hybrydowymi.

Opracowane algorytmy pozwalają na wyznaczenie drobno- lub gruboziarnistej równoległości (ziarnistość opisana została dokładniej w rozdziale 2.3.1) w zależności od typu pętli oraz rodzaju zależności w niej występujących. W związku z tym, możliwe jest wykonanie uzyskanego kodu na każdej z przytoczonych wcześniej architektur, w zależności od wielkości uzyskanego ziarna. W trakcie wykonania kodu drobnoziarnistego stosunkowo często konieczna jest synchronizacja pomiędzy wątkami pracującymi nad rozwiązaniem jednego problemu. Duża częstotliwość komunikacji między zadaniami powoduje, iż efektywne wykonanie kodu możliwe jest w architekturze o niskich kosztach komunikacji między procesorami np. systemy z pamięcią dzieloną. Gruboziarnistość, w odróżnieniu od drobnoziarnistości, pozwala na zminimalizowanie komunikacji między procesorami, gdyż komunikacja między stosunkowo dużymi ziarnami (fragmentami) kodu odbywa się rzadziej. Kod reprezentujący gruboziarnistość dedykowany jest w szczególności dla systemów rozproszonych, gdzie koszt komunikacji między jednostkami przetwarzającymi jest stosunkowo duży. Możliwe jest również efektywne wykonanie kodu o dużym ziarnie na systemach z pamięcią dzieloną, jednak w tym przypadku skalowalność takich systemów może zostać niewykorzystana. Należy również podkreślić, iż ziarnistość kodu jest pojęciem względnym zależnym w dużym stopniu od architektury (szersze wyjaśnienie tego zagadnienia w rozdziale 2.3.1).

2.2. Zależności: reprezentacja i wykrywanie

Ustalenie, w jaki sposób jedna instrukcja zależy od drugiej, jest jednym z ważniejszych zadań podczas wyznaczania poziomu równoległości kodu jak i jego sposobu zrównoleglenia. W celu wyznaczenia równoległości na poziomie instrukcji niezbędne jest określenie, które instrukcje mogą zostać wykonane w sposób równoległy. Jeżeli dwie instrukcje są zależne od siebie to wykonanie ich musi przebiegać w sposób sekwencyjny, zgodnie z porządkiem leksykograficznym. W przypadku braku zależności mogą one zostać wykonane równolegle niezależnie od siebie. Kluczowym zadaniem w obu sytuacjach jest ustalenie czy istnieją oraz jakiego rodzaju są to zależności [41].

W poniższym rozdziale przedstawione zostały rodzaje zależności, wraz z omówieniem warunków, w jakich dana zależność zachodzi. W podrozdziale 2.2.2

przedstawiono rozszerzenie definicji zależności między pojedynczymi instrukcjami na definicję zależności między instrukcjami występującymi w pętli. W podrozdziale przedstawiono sposób reprezentacji zależności wykorzystany w niniejszej dysertacji do opisu algorytmów i przeprowadzonych badań eksperymentalnych.

2.2.1. Rodzaje zależności

Bezpośrednie zrównoleglenie wybranego fragmentu kodu, możliwe jest w przypadku braku zależności między poszczególnymi instrukcjami. W przypadku występowania zależności konieczne jest zastosowanie odpowiednich transformacji (opisanych w dalszej części pracy, podrozdziały 2.3.1 oraz 4.1) pozwalających na eliminację zależności lub przekształcenie kodu do postaci równoległej nienaruszającej wykrytych zależności. Ze względu na rodzaj występujących zależności wyróżniamy dwa typy ograniczeń uniemożliwiających bezpośrednie zrównoleglenie fragmentu kodu.

Pierwsze ograniczenie odnosi się do przepływu sterowania i określane jest zależnością sterowania (ang. *control dependence*). Poniższy pseudokod przedstawia przykład zależności sterowania:

```
S1:   IF (T ≠ 0.0) GOTO S3
S2:       A = A / T
S3:   CONTINUE
```

Instrukcja S_2 nie może zostać wykona przed instrukcją S_1, gdyż S_2 jest warunkowo zależna od S_1. Wykonanie instrukcji S_2, jako pierwszej może spowodować błąd dzielenia przez zero, niemożliwy do wystąpienia w oryginalnej wersji programu [1].

Zależność sterowania [71] występuje między instrukcją S_i opisującą warunkowy skok do instrukcji S_j ($S_i \neq S_j$) jeśli skok w instrukcji S_i kontroluje wykonanie instrukcji S_j.

Drugie ograniczenie odnosi się do prawidłowej kolejności pobierania i modyfikowania danych. Zależność występująca zgodnie z tym ograniczeniem nosi nazwę zależności danych (ang. *data dependence*) [1]. Poniższy pseudokod przedstawia przykład zależności danych:

```
S1: PI = 3.14
S2: R = 5.0
S3: AREA = PI * R^2
```

Instrukcja S_3 musi zostać wykonana jako ostatnia. Wykonanie instrukcji S_3 przed instrukcjami S_1 lub S_2 może prowadzić do uzyskania nieprawidłowych wyników, gdyż instrukcja S_3 wykorzystuje wartości zmiennych PI i R modyfikowanych w instrukcjach

S_1 i S_2, odpowiednio. Instrukcje S1 i S2 mogą zostać wykonane współbieżnie względem siebie, gdyż nie występują między nimi zależności danych.

Między instrukcjami S1 i S2 występuje **zależność danych** (instrukcja S2 zależy od instrukcji S1) wtedy i tylko wtedy, gdy [1]:

obydwie instrukcje odwołują się do tej samej komórki pamięci i przynajmniej jedno z odwołań jest zapisem do pamięci,

istnieje ścieżka odwołań do tej samej komórki pamięci od instrukcji S1 do instrukcji S2 w trakcie wykonania.

W obrębie zależności danych istnieją trzy podstawowe typy zależności [1], [71]:

- Prosta - zwana również zależnością przepływu danych (ang. *Data – Flow Dependence*), występuje między instrukcjami S1 i S2 ($S1 \prec S2$), gdy instrukcja S2 odczytuje dane zapisane w instrukcji S1.

  S1 : X = ...

  S2 : ... = X

  Zależność przepływu danych wskazuje tzw. kolejność zapis-przed-odczytem *(*ang. *write-before-read)*, która musi być spełniona [71]. Sposób oznaczenia zależności prostej przyjęto zgodnie za R.Allen, K.Kennedy [1]: S1 δ S2 (odczyt w S2 zależy od zapisu w S1).

- Odwrotna - zwana również antyzależnością (ang. *antidependence*), występuje między instrukcjami S1 i S2 ($S1 \prec S2$), gdzie instrukcja S1 odczytuje dane z miejsca, które zapisywane jest w instrukcji S2.

  S1 : ... = X

  S2 : X = ...

  Antyzależność wyznacza tzw. kolejność odczyt-przed-zapisem (ang. *read-before-write)*, która nie powinna być naruszona w trakcie wykonywania równoległych obliczeń [71]. Sposób oznaczenia zależności odwrotnej przyjęto zgodnie za R.Allen, K.Kennedy [1]: S1 δ^{-1} S2.

- Po wyjściu (ang. *output dependence*) – występuje miedzy instrukcjami S1 i S2 ($S1 \prec S2$), gdy instrukcje S1 i S2 zapisują do tego samego miejsca.

  S1 : X = ...

  S2 : X = ...

  Zależność „po wyjściu" wskazuje tzw. kolejność zapis-przed-zapisem *(*ang. *write-before-write)*, która nie powinna być naruszona w trakcie wykonywania

obliczeń równoległych [71]. Sposób oznaczenia zależności po wyjściu przyjęto za R.Allen, K.Kennedy [1]: S1 δ^0 S2.

Ze względu na występowanie powyższych zależności, w wielu przypadkach niemożliwe jest wykonanie kodu w sposób równoległy bez uprzedniej analizy, a następnie redukcji zależności. Możliwe jest wyeliminowanie zależności odwrotnych oraz po wyjściu przy zastosowaniu odpowiednich technik opisanych w wielu istniejących już pracach np. [1], [71].

2.2.2. Zależności w pętlach

Wykrywanie zależności w pętlach programowych wymaga rozszerzenia koncepcji zależności przedstawionych w podrozdziale 2.2.1. Do wyrażeń, dla których wyznaczane są zależności konieczne jest dodanie dokładnych informacji o iteracji, w której dane wyrażenie występuje. Informacje o konkretnej iteracji zawarte są w wektorze iteracji (definicja w dalszej części rozdziału).

Pętla idealnie zagnieżdżona *(*ang. *perfectly nested loops*) jest to pętla n–krotnie zagnieżdżona (n > 0), której wszystkie instrukcje znajdują się w ciele pętli najbardziej zagnieżdżonej. Jeśli nie wszystkie instrukcje tworzą ciało najbardziej wewnętrznej pętli to pętlę nazywamy nieidealnie zagnieżdżoną (ang. *non-perfectly nested loops*).

Poniższy pseudokod przedstawia strukturę dwóch opisywanych typów pętli.

DO i = 1, M
  DO j = 1, N
  Instrukcja 1
  Instrukcja 2
  ENDDO
ENDDO
	DO i = 1, M
	  Instrukcja 1
	  DO j = 1, N
	  Instrukcja 2
	  ENDDO
	ENDDO

Pętla idealnie zagnieżdżona	Pętla nieidealnie zagnieżdżona

Dla dowolnej pętli, w której indeks pętli I przechodzi z indeksu L do U zgodnie z krokiem S, **numer iteracji i**, wybranej iteracji, jest równa wartości (I-L+S) / S, gdzie I jest wartością indeksu dla tej iteracji [1].

Dla gniazda n pętli, **wektor iteracji I** dla najbardziej wewnętrznej pętli jest wektorem liczb całkowitych zawierającym numery iteracji dla każdej pętli zgodnie z kolejnością zagnieżdżenia pętli. Innymi słowy, wektor zależności postaci I = {i_1, i_2, ..., i_n}, gdzie i_k, $1 \leq k \leq n$, reprezentuje numer iteracji pętli dla poziomu zagnieżdżenia k [1].

Zgodnie z powyższym, sparametryzowane wyrażenie określone poprzez wektor iteracji, oznacza instancję wyrażenia wykonanego przy zmiennych indeksowych przyjmujących wartości wektora iteracji. Przykładowo dla poniższej pętli:

```
DO I = 1, 2
   DO J = 1, 2
      S
   ENDDO
ENDDO
```

wyrażenie S[(2,1)] oznacza instancję wyrażenia S wykonywaną dla zmiennej indeksowej I = 2 (druga iteracja pierwszej pętli) oraz J = 1 (pierwsza iteracja drugiej pętli). Zbiór wszystkich wektorów iteracji dla wyrażeń pętli określa **przestrzeń iteracji**. Przestrzeń iteracji dla wyrażenia S w powyższej pętli przedstawiona została jako następujący zbiór kolejnych iteracji: [(1,1), (1,2), (2,1), (2,2)].

Na podstawie występujących zależności opisanych w rozdziale 2.2.1 oraz powyższych definicji możliwe jest określenie zależności występujących wewnątrz pętli.

Miedzy wyrażeniami S_1 a S_2 dla wspólnego gniazda pętli występuje zależność wtedy i tylko wtedy, gdy istnieją dwa wektory iteracji I oraz J dla danego gniazda takie, że [1]:

- $i < j$ lub $i = j$ oraz istnieje ścieżka od instrukcji S_1 do S_2 w ciele pętli,
- wyrażenia S_1, S_2 odwołują się do lokalizacji M w iteracji i oraz j,
- przynajmniej jedno z tych odwołań jest zapisem.

Zakładając istnienie zależności między instrukcją S_1 oraz S_2 dla iteracji i oraz j, **wektor zależności** zdefiniowany jest jako wektor o długości n, taki że $d(i,j)_k = j_k - i_k$.

Instrukcje S_1 i S_2 mogą odwoływać się do tej samej lokalizacji w obrębie pojedynczej iteracji pętli, jak również w obrębie dwóch różnych iteracjach pętli. W pierwszym przypadku mamy do czynienia z zależnością niezależną od pętli, w drugim przypadku z zależnością przenoszoną pętlą.

Zależność przenoszona pętlą (ang. *loop-carried dependence*) występuje między wyrażeniami pętli S_2 i S_1 wtedy i tylko wtedy gdy wyrażenie S_1 odwołuje się do lokalizacji M dla iteracji i oraz wyrażenie S_2 odwołuje się do tej samej lokalizacji dla iteracji j oraz wektor dystansu d(i,j) jest leksykograficznie większy od zera (>0).

Zależność niezależna od pętli (ang. *loop-independent dependence*) występuje między wyrażeniami pętli S_2 i S_1 wtedy i tylko wtedy, gdy wyrażenie S_1 odwołuje się do lokalizacji M w iteracji i, wyrażenie S2 odwołuje się do lokalizacji M w iteracji j, iteracja i jest równa iteracji j (i = j) oraz istnieje ścieżka od instrukcji S_1 do instrukcji S_2 wewnątrz pojedynczej iteracji.

Na poniższym przykładzie przedstawione zostały dwa rodzaje zależności

```
      DO I = 1, N
S1:       A( i  + 1 ) = ...
S2:       B( i ) = A( i )
S3 :      ... = B( i )
      ENDDO
```

W każdej iteracji występuje zależność prosta między instrukcjami S2 i S3 (S2 δ S3). Instrukcja S2 zapisuje wartości do tablicy B, natomiast instrukcja S3 odczytuje uprzednio zapisane wartości z tablicy B. Pętla zawiera również zależności przenoszone pętlą między instrukcjami S1 i S2. W instrukcji S1 następuje zapis do tablicy A w iteracji i, natomiast w instrukcji S2 odczyt z tej samej tablicy w iteracji i+1, w stosunku do zapisu w instrukcji S1. Druga zależność jest również zależnością prostą, lecz w odróżnieniu od pierwszej, zależność jest przenoszona pętlą.

2.2.3. Reprezentacja i wykrywanie zależności

Do opisu zależności oraz implementacji algorytmów wybrana została analiza zależności zaproponowana przez Pugha oraz Wonncotta [81], gdzie zależności reprezentowane są przez relacje zależności zbudowane z formuł Presburgera. Formuły Presburgera zbudowane są z ograniczeń liniowych nad zmiennymi całkowitymi, logicznych połączeń, kwantyfikatora ogólnego (ang. *universal quantifier*) oraz kwantyfikatora szczegółowego (ang. *existantial quantifier*) [81].

Na potrzeby niniejszej pracy wybrane zostały narzędzia Petit i Omega Calculator. Petit, narzędzie akademickie [57], pozwala na wyznaczenie relacji zależności zgodnie z analizą zależności zaproponowana przez Pugha oraz Wonnacotta [81]. Narzędzie Omega Calculator [59] umożliwia manipulacje na zbiorach i relacjach krotek.

W celu zrozumienia dalszej części pracy niezbędne jest wyjaśnienie podstawowych pojęć związanych z formułami Presburgera jak i wybranymi narzędziami [58]:

- **k-Krotka** (ang. *Tuple*) jest to punkt w przestrzeni Z^k o wartościach całkowitych o wymiarze k.
- **Zbiór** (ang. *Set*) zbudowany jest z k-ktorek, gdzie k jest liczbą całkowitą.
- **Relacja** (ang. *Relation*) jest to mapowanie n wymiarowych krotek na m wymiarowe krotki.

W relacji mapowane krotki nazywane są krotkami wejściowymi, natomiast krotki na które następuje mapowanie to krotki wyjściowe. Wymiar krotek wejściowych oraz

wyjściowych określa odpowiednio wymiar wejściowy oraz wyjściowy relacji. Wymiary na wejściu i wyjściu relacji mogą być różne od siebie.

Ze względu na to, że zbiór lub relacja może być nieskończona lub zależeć od wartości innych zmiennych, w ogólnym przypadku niemożliwe jest opisane zbioru lub relacji poprzez proste zapisanie wszystkich krotek lub par krotek. Do opisu zbiorów oraz relacji użyto zmiennych odpowiadających pozycjom w krotkach wejściowych i wyjściowych. Dla relacji zmienne te określane są jako zmienne wejściowe lub wyjściowe. Dla zbioru są to zmienne opisujące zbiór.

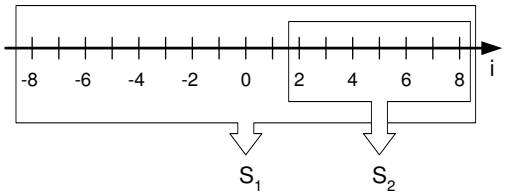
Przykładowy zbiór reprezentujący wszystkie liczby parzyste przedstawiony jest poniżej.

$$S_1 = \{[i]: \exists \alpha \; s.t. \; i = 2\alpha\}.$$

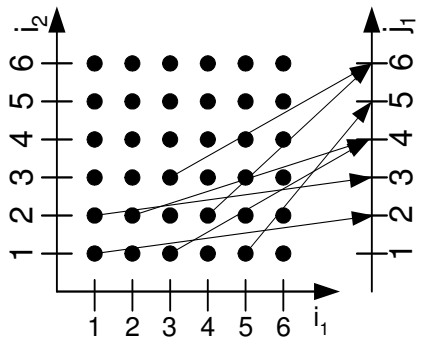
Zbiór S_1 jest zbiorem jednowymiarowym opisywanym przez jedną zmienną „i". Na zmienną „i" nałożone jest ograniczenie $\exists \alpha \; s.t. \; i = 2\alpha$ oznaczające, że ze zbioru liczb całkowitych wybierane są wartości dla zmiennej „i" dla których istnieje podstawienie i = 2α, gdzie α jest dowolną liczbą całkowitą. Rozszerzając ograniczenia dla zbioru S_1 do postaci:

$$S_2 = \{[i]: \exists \alpha \; s.t. \; i = 2\alpha \wedge 0 < i < n\},$$

uzyskujemy zbiór reprezentujący wszystkie liczby parzyste z określonego zakresu, w tym przypadku liczby parzyste są jednocześnie większe od zera i mniejsze od stałej symbolicznej n. Graficzna reprezentacja zbiorów S1 i S2 przedstawiona został na rys. 2.8.



Rysunek 2.8. Graficzna reprezentacja zbiorów S_1 i S_2 [op. własne]



Rysunek 2.9. Graficzna reprezentacja mapowania zgodnie z relacją R [op. własne]

Przykładowa relacja odpowiadająca operacji dodawania dwóch liczb całkowitych przedstawiona została poniżej,

$$R=\{[i_1,i_2]\rightarrow[j_1]:i_1+i_2=j_1\wedge 0<i_1,i_2<m\}.$$

Wejściem relacji R są krotki dwuwymiarowe $[i_1,i_2]$ natomiast wyjściem krotki jednowymiarowe $[j_1]$. W tym przypadku mamy do czynienia z mapowaniem przestrzeni dwuwymiarowej na jednowymiarową. Mapowanie odbywa się zgodnie ze zdefiniowanym ograniczeniem $i_1+i_2=j_1$, gdzie zmienna wyjściowa j_1 relacji R przyjmuje wartości równe sumie wartości zmiennych wejściowych $i_1 + i_2$. Dodatkowo zakres wartości przyjmowanych zmiennych wejściowych i_1 i i_2 został ograniczony do liczb większych od zera i mniejszych od stałej symbolicznej m. Graficzna reprezentacja mapowania zgodnie z relacją R przedstawiona została na rys. 2.9.

Tabela 2.1. Podstawowe operacje na relacjach i zbiorach w arytmetyce Pressburgera

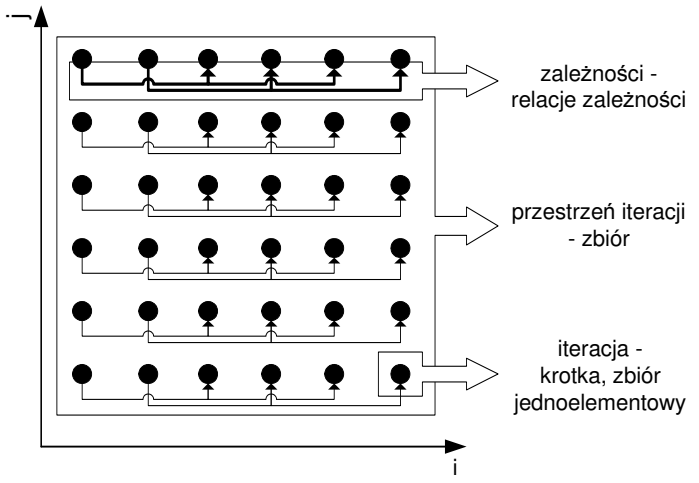
Relacja, Operacja	**Zapis w Omega Calculator**	**Znaczenie**
$R_1 \cap R_2$ $S_1 \cap S_2$	Intersection	Część wspólna dwóch relacji lub zbiorów
$R_1 - R_2$ $S_1 - S_2$	−, Difference	Różnica dwóch relacji lub zbiorów
$R_1 \cup R_2$ $S_1 \cup S_2$	Union	Suma dwóch relacji lub zbiorów
Inverse R	¬, Inverse	Odwrotność relacji
Transitive Closure of R	+, Transitive_Closure	Tranzytywne Domknięcie relacji

Ograniczenia w zbiorach i relacjach zdefiniowane są za pomocą formuł Presburgera [81]. Podstawowe operacje na zbiorach i relacjach w arytmetyce Presburgera przedstawione zostały w tab. 2.1. Formuły Presburgera zbudowane są z ograniczeń afinicznych na zmiennych całkowitych. Ograniczenia mogą przyjąć formę równań lub nierówności. Dodatkowo między poszczególnymi ograniczeniami dopuszczalne jest stosowanie operacji logicznych: zaprzeczenie (¬), alternatywa (∨), koniunkcja (∧). Dopuszczalne jest również stosowanie kwantyfikatorów szczegółowego (∃) oraz ogólnego (∀) wewnątrz ograniczeń. Ograniczenia na zbiór bądź relacje mogą przyjąć formę nieokreśloną (ang. *UNKNOWN*). Ograniczenie to oznacza, że istnieje jedno lub więcej innych nieznanych ograniczeń w formułach Presburgera. Ograniczenie to może być dodane do formuł przez użytkownika, jak również może wystąpić w przypadku zastosowania nieznanych symboli w funkcjach lub w przypadku użycia tranzytywnego domknięcia. Relacja zawierająca ograniczenia

typu UNKNOWN nazywana jest relacją niedokładną (ang. *inexact*). Niedokładność może zostać usunięta poprzez zastosowanie górnej lub dolnej granicy na danej relacji. Zastosowanie górnej granicy powoduje, że wszystkie ograniczenia typu UNKNOWN będą interpretowane jako logiczna prawda. W przypadku dolnej granicy ograniczenia UNKNOWN interpretowane są jako ograniczenia typu fałsz.

Krotka jak i przestrzeń iteracji pętli zbudowane są z punktów w przestrzeni Z^k (gdzie k jest wymiarem przestrzeni). W pierwszym przypadku wymiar określa liczbę zmiennych opisujących krotkę, natomiast w przypadku pętli wymiar określa gniazdo pętli. W związku z powyższym pojedyncza krotka może reprezentować pojedynczą iterację dowolnej pętli. Przy pomocy zbioru zbudowanego z k-krotek możliwe jest reprezentowanie całej bądź części przestrzeni iteracji pętli. Zależności pomiędzy poszczególnymi iteracjami pętli reprezentowane są przez relacje, w dalszej części pracy zwane relacjami zależności. Graficzna reprezentacja powyższych założeń przedstawiona została na rys. 2.10 dla przykładu 2.1:

```
DO I = 1, M
   DO J = 1, N
S1:    A(i + 2, j) = ...
S2:    ... = A(i, j)
   ENDDO
ENDDO
```



Przykład 2.1. Pętla z zależnościami o stałym wektorze

Rysunek 2.10. Mapowanie pojęć z dziedziny zależności danych na przyjęty sposób reprezentacji zależności [op. własne]

Dla powyższego przykładu przestrzeń iteracji pętli reprezentowana jest przez następujący zbiór:

$$S = \{i, j\} : 1 \le i \le M \ \wedge 1 \le \ j \le N \ ,$$

natomiast relacje zależności opisane są przez relację:

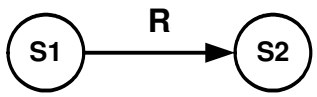
$$R = \{i, j\} \rightarrow \{i', j'\} : i' = i + 2 \wedge \ j' = \ j \wedge 1 \le i, i' \le M \ \wedge 1 \le \ j, j' \le N \ .$$

Możliwe jest również wyznaczenie wektora zależności między dowolną parą zależnych iteracji należących do zbioru S (czyli do przestrzeni iteracji pętli) opisanych relacją zależności R (czyli dla których zachodzi zależność).

$$I = \{1,1\},$$
$$J = R(I) = R(\{1,1\}) = \{3,1\},$$
$$K = J - I = \{3,1\} - \{1,1\} = \{2,0\}.$$

W tym przypadku relacja zależności R opisuje zależność prostą między S1 a S2 przenoszoną pętlę o wektorze dystansu K={2,0}. Dla bardziej skomplikowanych pętli, pojedyncza relacja może opisywać zależności o różnych wektorach dystansu, w związku z czym zbiór K może zawierać więcej niż jeden wektor dystansu.

W niniejszej pracy do reprezentacji zależności między instrukcjami wykorzystane zostały również grafy. Instrukcje pętli reprezentowane są przez wierzchołki grafu, natomiast zależności między instrukcjami odpowiadają krawędziom grafu. Dla przykładu 2.1 graf opisujący zależności przyjmuje następującą postać jak na rys. 2.11.



Rysunek 2.11. Graf zależności [op. własne]

2.3. Przetwarzanie równoległe – podstawowe pojęcia

Przetwarzanie równoległe nierozerwalnie wiąże się z podziałem większych problemów na mniejsze składowe - zadania lub obliczenia możliwe do wykonania w sposób równoległy. Tworzenie rozwiązań dla maszyn równoległych wymaga od programisty znajomości docelowej architektury. W zależności od architektury systemu preferowane są różne modele równoległości. Dla maszyn ze wspólna pamięcią stosuje się model z równoległością danych (ang. *data parallelism*), natomiast dla maszyn z pamięcią rozproszoną stosuje się model programowania z przesyłaniem komunikatów (ang. *message passsing*). Przykładowe środowiska realizujące powyższe modele programowania zostały omówione w kolejnych podrozdziałach.

Architektura systemu oraz przyjęty model programowania określa sposób podziału problemu na mniejsze podzadania. Jeśli podział problemu zostanie przeprowadzony bez uwzględnienia docelowej architektury, to uzyskane rozwiązanie może być mało efektywne, a nierzadko wolniejsze od rozwiązania problemu w sposób sekwencyjny. Biorąc pod uwagę jedynie koszty związane z komunikacją między jednostkami przetwarzającymi, dla systemów z pamięcią rozproszoną preferowany jest

gruboziarnisty podział zadań, w którym ilość niezbędnej komunikacji jest w naturalny sposób minimalizowana. W systemach z pamięcią dzieloną, gdzie koszt komunikacji między jednostkami przetwarzającymi jest stosunkowo mały, możliwe jest zastosowanie drobnoziarnistego podziału problemu. Jednakże nie tylko komunikacja wpływa na zwiększenie opóźnień w systemach wieloprocesorowych. Duże znaczenie mają również następujące wskaźniki:

- czas tworzenia / unicestwiania wątków
- liczba punktów synchronizacji wykorzystana w programie (np. bariery)
- operacje wejścia-wyjścia (np. operacje wejścia-wyjścia na plikach)
- aktualne obciążenie systemu przez inne procesy

Głównym celem stawianym systemom równoległym jest skrócenie czasu rozwiązywania problemu w stosunku do czasu uzyskanego na maszynie sekwencyjnej. Porównanie powyższych czasów umożliwia dokonanie wstępnej oceny jakości programu równoległego, a w szczególności pozwala na określenie uzyskanego przyśpieszenia. Programista tworzący rozwiązanie dla systemów równoległych, już na etapie podziału zadania na mniejsze podzadania, powinien uwzględnić wiele wskaźników opisujących jakość uzyskanego kodu np. lokalność danych, ziarnistość, przyśpieszenie, efektywność, determinizm uzyskiwanych wyników.

Poruszone zagadnienia zostaną omówione w poniższym rozdziale.

2.3.1. Ziarnistość kodu

Równoległe zadania mogą być zdefiniowane na różnych poziomach ziarnistości. Z jednej strony pojedynczy program, w zbiorze programów, może być traktowany jako pojedyncze zadanie. Z drugiej strony, pojedyncze instrukcje w programie mogą być postrzegane jako zadania równoległe. Między tymi dwoma ekstremami znajduje się zbiór modeli określających struktury kontrolujące wykonanie programu oraz wspierające je architektury [37].

Liczba oraz wielkość zadań, na jakie problem został podzielony określają **ziarnistość dekompozycji**. Podział na wiele zadań o niewielkich rozmiarach określany jest jako **podział drobnoziarnisty** (ang. *fine-grained*). Z drugiej strony podział problemu na niewielką liczbę dużych zadań określany jest jako **podział gruboziarnisty** (ang. *coarse-grained*) [37].

Inni badacze **ziarnistość obliczeń** określają jako liczbę wykonanych operacji między wystąpieniem zdarzenia komunikacji lub synchronizacji. Ziarnistość może być również zdefiniowana, jako liczba wykonanych operacji pomiędzy zmianą kontekstu

obliczeń lub między odwołaniami do zewnętrznej pamięci w systemie, w którym hierarchiczna struktura pamięci jest widoczna dla programisty. Podział programu na stosunkowo duże ziarna nazywany jest podziałem gruboziarnistym, zaś podział na drobne ziarna nazywamy podziałem drobnoziarnistym [9].

Ziarnistość obliczeń jest ważnym wskaźnikiem w przetwarzaniu równoległym, mającym zasadnicze znaczenie w określaniu, w jakim stopniu obciążenie programu może być zbalansowane na maszynie równoległej. Im mniejsza ziarnistość kodu tym większe możliwości do równomiernego rozłożenia obciążenia na dostępne procesory. Jeśli ilość obliczeń wykonanych przez każdy z procesorów jest porównywalna, w takim przypadku mówimy, że program jest dobrze zbalansowany (ang. *well-balanced*) [9]. Jednakże podział programu na zbyt drobne ziarna, może wpłynąć negatywnie na czas wykonania całego programu. W przypadku drobnoziarnistej równoległości narzuty czasowe związane z zarządzaniem wieloma zadaniami (np. czas tworzenia i unicestwiania, komunikacji i synchronizacji wątków) mogą stać się zbyt kosztowne i nieopłacalne w stosunku do wykonanych obliczeń w ramach pojedynczego ziarna oraz całego programu. W związku z tym podział kodu na drobne ziarna preferowany jest w systemach, gdzie koszty operacji niezaliczanych do obliczeń są relatywnie niskie oraz prawidłowe balansowanie obciążeniem ma duże znaczenie. Gruboziarnista równoległość pozwala na redukcję kosztów zarządzania wieloma zadaniami oraz zmniejszenie zapotrzebowania na pamięć poprzez zwiększenie lokalności kodu. Jednakże gruboziarnisty podział bardzo często prowadzi do nierównomiernego rozłożenia obliczeń między ziarnami, co z kolei uniemożliwia prawidłowe balansowanie obciążeniem. Gruboziarnistość preferowana jest w systemach gdzie koszt komunikacji między jednostkami przetwarzającymi jest stosunkowo duży (np. dla systemów rozproszonych).

W zależności od typu użytej transformacji pętli możliwe jest uzyskanie pożądanej ziarnistości kodu dla wybranej architektury. Tabela 2.2 przedstawia podział transformacji pętli programowych w zależności od możliwej do uzyskania ziarnistości [1], [5], [56].

W ogólnym przypadku ziarnistość kodu jest pojęciem względnym, ściśle związanym z określona architekturą systemu wieloprocesorowego [66]. Kod wykonany na systemie z pamięcią dzieloną sklasyfikowany jako gruboziarnisty, może zostać uznany za drobnoziarnisty w systemie z pamięcią rozproszoną. Zgodnie z tym, co zostało przedstawione, w zależności od architektury koszt zarządzania wielozadaniowością (np. zarządzanie wątkami) może znacznie różnić się od siebie.

Tabela 2.2. Zestawienie transformacji do uzyskania określonej ziarnistości kodu [1], [5]

Transformacja	**Ziarnistość**[1]	
	drobno-	**grubo-**
Zmiana kolejności wykonani pętli (ang. *loop interchange*)	●	●
Rozszerzenie skalaru (ang. *scalar expansion*)	●	○
Zmiana nazw zmienny skalarnych (ang. *scalar renaming*)	●	○
Zmiana nazw zmiennych tablicowych (ang. *array renaming*)	●	○
Podział węzłów (ang. *node splitting*)	●	○
Redukcja (ang. *reduction*)	●	○
Podział zmiennych indeksowych (ang. *index-set splitting*)	●	○
Przekoszenie pętli (ang. *loop skewing*)	●	●
Prywatyzacja (ang. *privatization*)	○	●
Podział pętli (ang. *loop distribution*)	○	●
Wyrównanie (ang. *alignment*)	○	●
Replikacja kodu (ang. *code replication*)	○	●
Łączenie pętli (ang. *loop fusion*)	○	●
Odwrócenie wykonania pętli (ang. *loop reversal*)	○	●
Wielowymiarowe łączenie pętli (ang. *multilevel loop fusion*)	○	●

Akceptowany poziom ziarnistości powinien być ustalany odrębnie dla każdej architektury, w ekstremalnym przypadku dla każdego systemu o tej samej architekturze. Możliwy jest scenariusz, w którym użycie różnych komponentów dla tej samej architektury wpłynie znacząco, na co najmniej jeden z ważnych czynników związanych z wydajnością systemu wieloprocesorowego np. koszt komunikacji między poszczególnymi jednostkami obliczeniowymi. W systemach z pamięcią dzieloną efekt taki możliwy jest do uzyskania poprzez zastosowanie pamięci dzielonej o różnych prędkościach. W przypadku szybkiej pamięci akceptowalna będzie mniejsza ziarnistość kodu w porównaniu do systemu, w którym użyta została stosunkowo wolna pamięć. W systemach rozproszonych, gdzie pamięć globalna rozproszona jest między poszczególnymi węzłami obliczeniowymi połączonymi między sobą siecią, opóźnienia

[1] ● – umożliwia uzyskanie ziarnistości, ○ – nie umożliwia uzyskanie ziarnistości

w komunikacji są jeszcze bardziej oczywiste. W przypadku, gdy komunikacja między węzłami obliczeniowymi odbywa się poprzez globalną sieć (np. WAN, potocznie zwana Internetem) akceptowana ziarnistość kodu (pozwalająca na uzyskanie pozytywnego przyśpieszenia, większego od 1) może zmieniać się w jednostce czasu w zależności od chwilowego obciążenia sieci połączeń.

2.3.1. Mierniki w przetwarzaniu równoległym

Jednym z podstawowych mierników, na jakie należy zwrócić uwagę podczas tworzenia kodu dla maszyn wieloprocesorowych jest **determinizm programu** (ang. *determinism*). Algorytm lub program jest deterministyczny, jeśli po wykonaniu dla identycznych danych wejściowych uzyskamy zawsze to samo wyjście. Jeżeli dla tych samych danych wejściowych możliwe jest uzyskanie różnych wyników oznacza to, że algorytm lub program jest niedeterministyczny. Równoległe modele programowania muszą zapewniać wsparcie powyższym wariantom, jednakże model programowania równoległego pozwalający na pisanie deterministycznych programów w jak najprostszy sposób jest wielce pożądany [34]. Niedeterministyczne algorytmy znajdują również swoje zastosowanie np. w przetwarzaniu obrazu gdzie niewielkie przekłamania często nie dyskwalifikują uzyskanych wyników. Najczęstszą przyczyną akceptowania niedeterministycznych rozwiązań jest uzyskanie dobrych wyników pod względem przyśpieszenia jak i efektywności kodu (pojęcia omówione w dalszej części rozdziału).

Szybkość obliczeń nie jest zależna jedynie od prędkości procesora. Jednym z głównych czynników wpływających na prędkość obliczeń jest zdolność do szybkiego dostarczenia danych z pamięci do procesorów, w celu zapewnienia stałego obciążenie procesorów. W ostatniej dekadzie częstotliwość taktowania procesorów wzrastała o około 40% każdego roku, podczas gdy czas dostępu do pamięci ulegał poprawie o około 10% w skali roku. Dodatkowo zwiększenie liczby instrukcji możliwych do wykonania podczas jednego cyklu zegara, stworzyło dużą przepaść między prędkością procesorów, a prędkością pamięci – na styku procesor-pamięć mamy do czynienia z tzw. „wąskim gardłem". Ze względów ekonomicznych nieopłacalne jest zastosowanie bardzo szybkiej pamięci w całym systemie. Przepaść między prędkością procesora a pamięcią niwelowana jest poprzez ułożenie pamięci w sposób hierarchiczny, gdzie pamięć leżąca najbliżej procesora jest pamięcią najszybszą, lecz ze względów ekonomicznych jest jednocześnie najmniejszą. Przykładem takiej pamięci jest pamięć

podręczna procesora (ang. *cache*), gdzie dostęp do niej jest dużo szybszy niż do pamięci na wyższym poziomie (np. ang. *RAM – random Access memory*) [37].

W związku z powyższym, w celu zapewnienia ciągłości przepływu danych z pamięci do procesora, optymalnym rozwiązaniem byłoby zapewnienie, iż wszystkie dane potrzebne do obliczeń znajdują się w pamięci położonej jak najbliżej procesora. Ze względu na hierarchiczne ułożenie pamięci (na każdym kolejnym poziomie pamięć jest większa, lecz jednocześnie wolniejsza) niezbędne jest, aby w trakcie wykonywania programu procesory odwoływały się do tych samych danych w jak najmniejszym odstępie czasu. Jeśli powyższy warunek jest spełniony mówimy, że program ma dobrą lokalność danych (ang. *data locality*). Lokalność danych ma zasadniczy wpływ na zwiększenie wydajności programu w systemach z pojedynczą jednostką przetwarzającą, a w szczególności dla maszyn wieloprocesorowych, gdzie wpływ lokalności dla całego systemu jest zwielokrotniony przez liczbę jednostek przetwarzających oraz ilość pamięci typu cache.

Wyróżnia się dwa rodzaje lokalności: tymczasową i przestrzenną. Z **tymczasową lokalnością** (ang. *temporal locality*) mamy do czynienia, gdy te same dane wykorzystywane są kilkukrotnie w krótkim odstępie czasu. **Przestrzenna lokalność** (ang. *spational locality*) występuje, gdy w krótkim okresie czasu następuje odwołanie do różnych danych rozmieszczonych niedaleko od siebie w pamięci [41]. Kod umożliwiający uzyskanie dwóch rodzajów lokalności bardzo często pozwala na uzyskanie zadowalającego przyśpieszenia, a nierzadko prowadzi do przyśpieszenia superliniowego (ang. *superlinear speedup*).

Przyśpieszenie (ang. *speedup*) jest to współczynnik przyśpieszenia obliczeń równoległych (S_p) wykonanych na p procesorach określony poniższym wzorem [72], [83]:

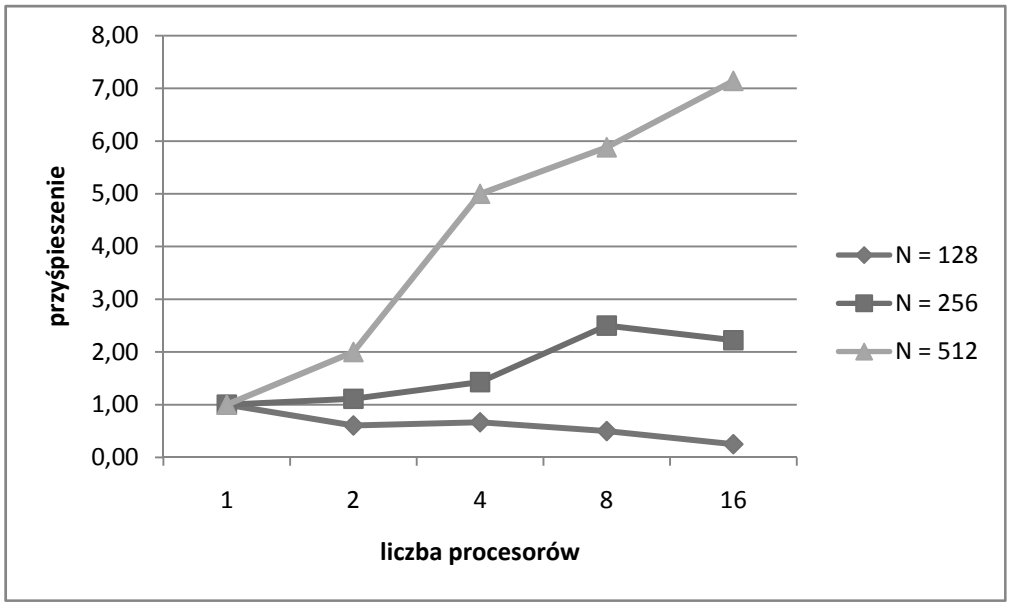
$$S_p = \frac{T_1}{T_p},$$

gdzie T_1 oznacza czas wykonania obliczeń na pojedynczym procesorze, natomiast T_p to czas potrzebny do wykonania tych samych obliczeń na p procesorach. Ze względu na czas niezbędny do tworzenia, synchronizacji, komunikacji zachodzącej między wątkami jak i innymi czynnikami opóźniającymi, **przyśpieszenie rzeczywiste** jest zazwyczaj mniejsze niż liczba procesorów ($1 \leq S_p \leq p$). W przypadku gdy współczynnik przyśpieszenia jest równy liczbie procesorów ($S_p = p$) mówimy o **przyspieszeniu liniowym**. W przypadku, gdy przyspieszenie jest większe od liczby użytych procesorów

do wykonania obliczeń (Sp > p) mamy do czynienia z tzw. **przyśpieszeniem superliniowym** (ang. *superlinear speedup*) [72].

Zgodnie z rys. 2.12 przyśpieszenie (S_p) przyjmuje wartości z trzech charakterystycznych zakresów:

- $0 \leq S_p \leq 1$ – wartości z tego zakresu oznaczają brak przyśpieszenia w stosunku do wersji sekwencyjnej, w takim przypadku wykonanie zadania na liczbie procesorów większej od 1 (P > 1) jest nieopłacalne – przykładem jest wielkość problemu dla N = 128;
- $1 < S_p \leq P$ – wartości z tego zakresu oznaczają dodatnie przyśpieszenie w stosunku do wersji sekwencyjnej, przykład przyśpieszenia przedstawiono dla problemu N = 256 oraz N = 512 z wyłączeniem wyników uzyskanych dla P = 4; jeśli przyspieszenie przyjmujące wartość P ($S_p = P$) to określane jest jako przyspieszenie liniowe
- $P < S_p$ – przyśpieszenie z tego zakresu określane jest jako superliniowe, głównym czynnikiem wpływającym na uzyskanie takiego przyśpieszenia jest bardzo dobre wykorzystanie lokalności kodu, jest to najbardziej porządne przyśpieszenie lecz jednocześnie najtrudniejsze do osiągnięcia; przykładem takiego przyspieszenia jest pomiar czasu dla 4 procesorów i N = 512.



Rysunek 2.12. Przyśpieszenie dla wielkości teoretycznego problemu (N) oraz liczby procesorów (P) [op. własne]

Wskaźnikiem ściśle związanym z przyśpieszeniem jest efektywność. **Efektywność** (ang. *efficiency*) obliczeń równoległych zdefiniowana jest jako stosunek współczynnika przyśpieszenia (S_p) do liczby procesorów p [72]:

$$E_p = \frac{S_p}{p} = \frac{T_1}{p * T_p}.$$

Efektywność można również zdefiniować w nieco odmienny sposób. Zakładając, że obliczenia na maszynie równoległej składają się z dwóch części. Pierwsza część (T_c) określa czas wykonania głównych obliczeń, natomiast część T||(P) określa dodatkowy czas wymagany w trakcie wykonania programu równoległego (np. czas wymagany na komunikację, czas oczekiwania na nadejście wiadomości, czas przestojów procesorów). W związku z powyższym równanie efektywności przyjmuje następującą postać [9]:

$$E_p = \frac{T_c}{T_c + T\,\|\,(P)} = \frac{1}{1 + \frac{T\,\|\,(P)}{T_c}}$$

Zgodnie z powyższym wzorem, aby efektywność E(P) zmierzała do wartości 1, T||(P) musi maleć w stosunku do T_c wraz ze wzrostem liczby procesorów p. Szybkość, z jaką to następuje może posłużyć do określenia, z jaką efektywnością równoległość może zostać wykorzystana do rozwiązania określonego problemu [9].

Teoretycznie, zgodnie z podanymi wzorami, dla sztucznie stworzonego środowiska (systemu) możliwe jest uzyskanie dowolnego przyśpieszenia jak i dowolnej efektywności. Jednakże w rzeczywistości w większości przypadków program składa się z dwóch części: sekwencyjnej i równoległej. Wiadomym jest, iż część sekwencyjna będzie wpływała niekorzystnie na czas wykonania całego programu. W związku z tym, dla problemu którego rozwiązanie wymaga wykonania części sekwencyjnej oraz części równoległej stosuje się **prawo Amdahla**. Maksymalne przyśpieszenie obliczeń równoległych, zgodnie z tym prawem, ograniczone jest przez rozmiar części sekwencyjnej wchodzącej w skład całego problemu, bez względu na liczbę użytych procesorów. Zakładając, że obliczenia sekwencyjne (f_s) stanowią pewną część z przedziału $0 < f_s < 1$ wszystkich obliczeń, a część poddana zrównolegleniu jest równa $1\text{-}f_s$ oraz przyjmując, że dla części zrównoleglonej osiągane przyśpieszenie równe jest przyśpieszeniu liniowemu, to przyśpieszenie na p procesorach dla wybranego problemu wynosi [2]:

$$S(P) = \frac{T(1)}{T(P)} = \frac{T(1)}{f_s T(1) + (1 - f_s)\frac{T(1)}{P}} = \frac{1}{f_s + \frac{1 - f_s}{P}}$$

gdzie:

T(1) – czas wykonania programu na pojedynczym procesorze

P – liczba procesorów użytych dla części równoległej

f_s – część sekwencyjna rozpatrywanego problemu.

Z prawa Amdahla wynika, iż bez względu na liczbę użytych procesorów, uzyskane przyśpieszenie będzie ograniczone do wartości $\frac{1}{f_s}$. Efektywność rozwiązania określona jest wzorem:

$$E(P) = \frac{1}{P * f_s + (1 - f_s)},$$

będzie dążyć do zera dla dużej liczby procesorów [9].

Rozwinięciem prawa Amdahla jest prawo Gustafsona (ang. *Gustafsons Law*), gdzie na efektywność systemu ma wpływ zarówno liczba procesorów jak również wielkość problemu do rozwiązania. Zgodnie z **prawem Gustafsona** jeśli zrównoleglona część problemu jest odpowiednio skalowalna, wtedy dowolna efektywność może zostać osiągnięta dla dowolnej liczby procesorów. Zgodnie z prawem Gustafsona przyspieszenie określone jest następującym wzorem [40]:

$$S = \frac{T_s + T \,||\, (N,1)}{T_s + T \,||\, (N,P)},$$

gdzie:

T_s – stały czas potrzebny do wykonania części sekwencyjnej problemu

T||(N,P) – czas potrzebny na wykonanie równoległej części obliczeń o rozmiarze N na P procesorach

Jeśli przyjmiemy, że T|| zwiększa się wraz ze wzrostem N, a maleje ze wzrostem P, wtedy dowolna wymagana efektywność (E) może zostać osiągnięta poprzez zwiększenie wielkości problemu. Przykładowo jeśli $T||(N,P)=N^2/P$ wówczas:

$$\lim_{N \to \infty} S = \frac{Ts + N^2}{Ts + N^2 / P} = \frac{PT_s + PN^2}{PT_s + N^2} = P,$$

w związku z czym efektywność dąży do P dla odpowiednio dużego problemu [9].

Prawo Gustafsona w lepszym stopniu odzwierciedla sposób wykorzystania systemów równoległych (rozproszonych), jednakże podobnie do prawa Amdahla jest tylko pewną idealizacją [9]. Do określenia związku między wielkością problemu

a efektywnością jego rozwiązania służy funkcja **izoefektywności** (ang. *isoefficiency function*), określona następującym wzorem [38]:

$$E(P) = \frac{T_c}{T_c + T\,\|\,(P)} = \frac{1}{1 + \frac{T\,\|\,(P)}{T_c}},$$

gdzie:

Tc – czas potrzebny na wykonanie obliczeń potrzebnych do rozwiązania problemu

T||(P) – narzut czasowy związany z obliczeniami równoległymi

Funkcja izoefektywości pozwala ustalić jak wielkość problemu musi rosnąć, aby w stosunku do liczby użytych procesorów do rozwiązania problemu efektywność rozwiązania nie uległa zmianie (pozostała stała) [9].

Funkcja izoefektywności przyjmuje wartości z przedziału <0,1>. Dla mocno skalowanych systemów równoległych wartości te będę małe, oznacza to, że niewielki wzrost problemu jest wystarczający do utrzymania utylizacji zwiększającej się liczby procesorów. System jest tym mniej skalowany im wartość funkcji zbliża się do górnej granicy, czyli do 1. Dla nieskalowalnych systemów równoległych funkcja izoefektywności nie istnieje, gdyż dla takich systemów niemożliwe jest utrzymanie efektywności na niezmiennym poziomie (ang. *constant*) przy zwiększającej się liczbie procesorów, bez względu na wzrost wielkości problemu [38].

2.4. Środowiska programowania systemów wieloprocesorowych

Z punktu widzenia programisty, wyróżnia się dwa główne modele programowania maszyn równoległych:

- z pamięcią dzieloną
- z przekazywaniem komunikatów.

Modele te zbudowane są na wspólnej koncepcji, współbieżnych wątków. Pojedynczy proces tworzy rozwidlenie (ang. *fork*) o określonej liczbie wątków, które w sposób równoległy wykonują część programu. Główną różnicę między modelem z pamięcią dzieloną a modelem z przesyłaniem komunikatów wyznacza sposób dostępu do wspólnej pamięci przez współbieżne wątki [78].

W modelu z pamięcią dzieloną, wątki odwołują się do pamięci o jednolitej przestrzeni adresowej. Wątki mogą zapisywać lub odczytywać dane z pamięci dzielonej (ang. *shared*) bez względu, na jakiej jednostce przetwarzającej są wykonywane.

Ze względu na poziom dostępu do danych, w systemach z pamięcią dzieloną wyróżniamy dane prywatne i dane dzielone. Dane dzielone (ang. *shared data*) są widoczne dla wszystkich procesorów wykonujących wspólne zadanie. Komunikacja między procesorami przybiera formę odczytu i zapisu informacji do danych dzielonych. Odmiennie traktowane są dane prywatne, które są lokalne dla każdego wątku i nie są dostępne dla innych wątków [78].

W modelu z przesyłaniem komunikatów równoległe procesory nie dzielą pamięci między sobą – wszystkie dane są prywatne dla każdego z wątków. W modelu tym wymagane jest, aby każdy proces posiadał wiedzę na temat właściciela konkretnych danych. W związku z tym, jeśli zajdzie potrzeba odczytu lub zapisu do danych należących do innego procesu, jednostka musi jawnie wysłać lub otrzymać komunikat [78].

Dla ułatwienia programowania, obydwa modele wspierane są przez konstrukcje wysokiego poziomu. Model z pamięcią dzieloną wspierany jest przez zbiór dyrektyw języka programowania, dzięki którym użytkownik określa, które fragmenty kodu maja zostać wykonane w sposób równoległy, np. OpenMP [49], [51], [75], [23] lub HPF [1], [42]. Użytkownik ma również możliwość programowania bezpośrednio na poziomie wątków, przy użyciu zbioru funkcji bibliotecznych. Model z przekazywaniem komunikatów wspierany jest zazwyczaj poprzez bibliotekę zawierającą zbiór funkcji, umożliwiających przekazywanie komunikatów, synchronizację, inicjalizację oraz grupowanie. Ogólnie znanymi standardami wspierającymi model programowania z przekazywaniem komunikatów jest biblioteka Message Passing Interface (MPI) oraz Parallel Virtual Machine (PVM) [78].

Model programowania z pamięcią dzieloną jest modelem programowania dedykowanym na architektury z pamięcią dzieloną, natomiast model z przekazywaniem komunikatów dla maszyn z pamięcią rozproszoną. Pomimo takiego podziału, wiele nowoczesnych komputerów równoległych jest kompatybilnych z dwoma modelami programowania, pomimo iż ich twórcy zakładali użycie jednego z wyżej wspomnianych modeli [78]. Poniżej omówione zostały najpopularniejsze środowiska wspierające programowanie dla maszyn równoległych.

2.4.1. PVM

Prace nad PVM (ang. *Parallel Virtual Machine*) rozpoczęły się w 1989 roku. Głównym celem, jaki przyświecał powstaniu PVM było wykorzystanie komputerów o różnych architekturach, połączonych ze sobą siecią, w celu utworzenia maszyny

równoległej, od strony użytkownika widocznej, jako pojedynczy komputer. Jednym z głównych wyzwań, jakie należało rozwiązać było przekazywanie danych między różnymi zadaniami – w rzeczywistości rozmieszczonych na różnych komputerach. W PVM rozwiązane zostało to poprzez przekazywanie komunikatów realizowane w całości przez system – niewidoczne dla programisty. W trakcie projektowania PVM większy nacisk położony został na przenośność kodu niż na wydajność. Na takie podejście główny wpływ miały następujące czynniki [36]:

- komunikacja przez internet w stosunku do mocy obliczeniowych maszyn jest kosztowna
- należało rozwiązać problem skalowalności, tolerancji na błędy, heterogeniczności (ang. *heterogeneity*) wirtualnych maszyn.

Użytkownik pisząc kod tworzy zbiór zadań współpracujących między sobą. Zadania odwołują się do zasobów PVM poprzez zbiór standardowych funkcji, pozwalających między innymi na inicjalizację, zakończenie, komunikację, synchronizację między zadaniami. W dowolnym punkcie wykonania programu, istniejące zadanie może rozpocząć lub zatrzymać wykonanie innego zadania, dodać lub usunąć fizyczny komputer z maszyny wirtualnej [44].

2.4.2. MPI

Programowanie z wykorzystaniem paradygmatu przesyłania komunikatów jest jednym ze starszych i najbardziej rozpowszechnionym podejściem w programowaniu maszyn równoległych. Dwie kluczowe właściwości charakteryzują wspomniane podejście. Pierwsza z nich zakłada rozproszoną przestrzeń adresową, natomiast druga zakłada zrównoleglenie jedynie jawnie oznaczonych fragmentów kodu [37]. Standard MPI definiuje zarówno składnię (ang. *syntax*) jak i semantykę (ang. *semantics*) zbioru podstawowych funkcji wykorzystywanych przy tworzeniu programów z przekazywaniem komunikatów [37], [55]. Interfejs MPI (ang. *Message-Passing Interface*) [50], [76], [39] definiuje standardową biblioteką przesyłania komunikatów, która może być użyta do napisania przenośnego programu działającego na wielu platformach. Początkowo standard MPI wpierany był przez języki Ansi-C oraz Fortran 77, obecnie wspierany jest (przynajmniej częściowo) przez większość dostępnych na rynku języków programowania. Programy napisane zgodnie ze standardem MPI mogą być wykonywane zarówno na maszynach z pamięcią rozproszoną jak również na maszynach z pamięcią dzieloną [84].

MPI jest przenośnym standardem programowani dostępnym na wielu platformach. Jednakże w ogólnym przypadku programowanie aplikacji z przekazywaniem komunikatów jest procesem skomplikowane. W trakcie tworzenia programu wymagany jest jawny podział danych na mniejsze paczki w związku, z tym kod musi zostać zmodyfikowany (zrównoleglony), aby mógł pracować na wspólnych danych. Dodatkowo w nowoczesnych wieloprocesorowych architekturach nieustannie zwiększa się wsparcie dla utrzymania spójności pamięci (ang. *cache coherence*). W takim przypadku przekazywanie komunikatów staje się niepotrzebne [26].

2.4.3. Posix Threads

POSIX Threads (ang. *Portable Operating System Interface*) jest standardem (POSIX Section 1003.1c) definiującym interfejs (dla języka C) służący programistom do tworzenia aplikacji wielowątkowych [64], [65], [74]. Biblioteki zaimplementowane zgodnie z tym standardem, zwane często Pthreads, dostępne są dla większości dystrybucji systemu UNIX (istnieje również implementacja dla systemu Windows).

Dostępnych jest ponad 60 funkcji, które ze względu na zastosowanie można podzielić na trzy główne klasy [74]:

- zarządzanie wątkami – np. tworzenie, niszczenie, łączenie wątków oraz zmiana atrybutów wątków,
- wzajemne wyłączenie (ang. *mutual exclusion, mutex*) – np. tworzenie, niszczenie, blokowanie, zwalnianie obszarów krytycznych oraz zbiór funkcji do zmiany wartości ich atrybutów,
- operacje na zmiennych warunkowych – funkcje adresowane do obsługi komunikacji pomiędzy wątkami współdzielącymi zmienne, funkcje do tworzenia, niszczenia, oczekiwania oraz sygnalizacji na podstawie wartości wybranej zmiennej.

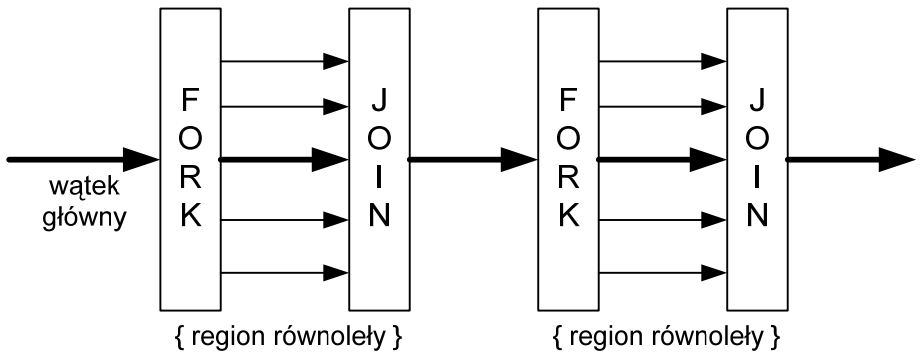
Posix Threads jest akceptowanym standardem niskiego poziomu dla systemów z pamięcią dzieloną. Ptheads pozwala na uzyskanie równoległości na poziomie zadań, nie na poziomie danych [26].

2.4.4. OpenMP

OpenMP (ang. *Open Multi-Processing*) jest standardem definiującym zbiór dyrektyw kompilatora, funkcji bibliotecznych, oraz zmiennych środowiskowych

umożliwiających programistom tworzenie przenośnych (portowalnych) wielowątkowych aplikacji na maszyny równoległe [49], [51], [75], [23]. OpenMP został zdefiniowany w latach 90-tych przez członków grupy OpenMP Architecture Review Board (ARB). Pierwsza wersja OpenMP, zawierająca zbiór dyrektyw dla języka Fortran, została opublikowana w 1997. W krótkim czasie pojawiły się kompilatory języka Fortran wspierające OpenMP. W kolejnym kroku powstała specyfikacja dla języka C i C++. W dniu dzisiejszym większość dużych producentów sprzętu, firm tworzących kompilatory oraz wiele placówek badawczych i grup naukowych należy do ARB [24].

Standard OpenMP definiuje interfejs programowania aplikacji (ang. *Application Programming Interface - API*) wpierający model programowania dla systemów z pamięcią dzieloną. Fragmenty kodu przeznaczone do zrównoleglenia muszą zostać wskazane przez programistę (brak jest wsparcia dla automatycznego zrównoleglenia) [51]. Zrównoleglenie wskazanego kodu odbywa się w oparciu o model fork-join (patrz rys. 2.13), wykorzystujący operacje rozwidlenia (ang. *fork*) oraz złączenia (ang. *join*). Program napisany zgodnie ze standardem OpenMP rozpoczyna się jako pojedynczy wątek (wykonywanego sekwencyjnie). W momencie napotkania regionu równoległego (ang. *parallel region*), główny wątek tworzy grupę wątków (ang. *team of threads*). Instrukcje zawarte wewnątrz regionu przydzielone zostają do poszczególnych wątków i wykonywane są w sposób równoległy (zakładając dostępność przynajmniej dwóch procesorów). Po wykonaniu wszystkich współbieżnych zadań, przy wyjściu z regionu równoległego wątki zostają zsynchronizowane i unicestwione, z wyjątkiem wątku głównego (ang. *master thread*).



Rysunek 2.13. Model rozwidlenia wątków (ang. *fork-join*) [51]

Kod programu może zawierać więcej niż jeden region równoległy. W takim przypadku dla każdego regionu tworzona jest oddzielna grupa wątków. Zgodnie z rys. 2.13 główny wątek przy wejściu do pierwszego regionu tworzy cztery dodatkowe

wątki, a przy wyjściu z regionu stworzone wątki są unicestwiane, natomiast wątek główny kontynuuje wykonywanie kodu. Analogicznie sytuacja występuje dla kolejnych regionów równoległych. Dodatkowo dozwolone jest również zagnieżdżanie regionów równoległych, jednakże nie jest to wspierane we wszystkich kompilatorach. Jeśli kompilator nie wspiera zagnieżdżeń regionów równoległych, to najczęściej zadania występujące w regionie zagnieżdżonym zostaną wykonane przez jeden wątek.

Jednym z głównych założeń przy tworzeniu standardu OpenMP było umożliwienie programistom wykorzystania wielowątkowości w istniejącym kodzie, bez konieczności przepisywania go na nowo. Spośród istniejących rozwiązań jedynie OpenMP pozwala na zrównoleglenie narastające (ang. *incremental parallelism*) w istniejącym kodzie z zachowaniem skalowalności rozwiązania (patrz tab. 2.3). W związku z tym, że OpenMP jest zbiorem dyrektyw, funkcji bibliotecznych oraz zmiennych środowiskowych, standard nie wprowadza zmian do języka bazowego [26].

Tabela 2.3. Porównanie modeli programowania równoległego

	MPI	**Pthreads**	**HPF**	**OpenMP**
Skalowalność (ang. scalable)	tak	zależne	tak	tak
Zrównoleglenia narastające (ang. Incremental parallelization)	nie	nie	nie	tak
Przenośność (ang. Portable)	tak	tak	tak	tak
Fortran binding	tak	nie	tak	tak
Wysokiego poziomu (ang. High level)	nie	nie	tak	tak
Wsparcie równoległości danych (ang. Support data parallelism)	nie	nie	tak	tak
Zorientowany na wydajność (ang. Performance orientem)	tak	nie	stara się	tak

2.5. Programowanie komputerów wieloprocesorowych

Tworzenie oprogramowania dla maszyny wieloprocesorowe wspierane jest obecnie przez wiele narzędzi naukowych jak i komercyjnych. Podstawowym narzędziem pozwalającym przekształcić kod napisany w języku wysokiego poziomu na język zrozumiały przez multiprocesory jest kompilator zrównoleglający. Czołowi producenci oprogramowanie np. Microsoft, Sun jak i producenci sprzętu np. Intel, AMD oferują zintegrowanie środowiska programistyczne wspierające proces tworzenia, testowania i utrzymania aplikacji wielowątkowych. Dostępne są również rozwiązania

niekomercyjne np. kompilator GNU (ang. *GNU Compiler Collection*) oraz rozwiązania powstałe w ośrodkach badawczych np. SUIF (Stanford University) [43], [73], [88], Parafrase-2 (Uniwersytecie Illinois) [45], [79] lub Polaris (Uniwersytecie Illinois) [77], [20], [30].

Część z wyżej wymienionych kompilatorów pozwala na automatyzację procesu zrównoleglenia programów sekwencyjnych (w szczególności zawartych w nich pętli programowych) na ich semantyczne odpowiedniki równoległe. Użytkownik zwolniony jest z konieczności ręcznego oznaczania miejsc możliwych do zrównoleglenia w programie sekwencyjnym. Dzięki takiemu podejściu możliwe jest zredukowanie:

- kosztów związanych z ręcznym i przez to czasochłonnym oznaczaniem równoległości w programach,
- błędów wynikających z ludzkich pomyłek.

Jednakże, żaden z powyższych kompilatorów nie pozwala na automatyczne wyznaczenie maksymalnej liczby niezależnych fragmentów kodu w pętlach sekwencyjnych dowolnie zagnieżdżonych. W związku z powyższym, istnieje potrzeba opracowania automatycznych bądź półautomatycznych transformacji (algorytmów) pozwalających na wyznaczenie maksymalnej liczby niezależnych fragmentów kodu, których wynikiem będzie kod zrównoleglony lub gotowy do zrównoleglenia zgodny z jednym z dostępnych standardów np. OpenMP. Podejście takie zostało zaproponowane, w postaci zbioru algorytmów dla pętli idealnie i dowolnie zagnieżdżonych, w kolejnym rozdziale.

2.6. Podsumowanie

W niniejszym rozdziale przedstawione zostały podstawowe pojęcia związane z przetwarzaniem równoległym. Zaprezentowano architekturę systemów równoległych ze szczególnym uwzględnieniem systemów z pamięcią dzieloną, które zostały użyte w trakcie prowadzonych badań eksperymentalnych (patrz rozdział 4). Szczegółowo omówiono zależności występujące w kodzie, ich rodzaje i sposoby reprezentacji z naciskiem na reprezentację zaproponowaną przez Pugha i Wonnacotta [81] wykorzystaną w niniejszej dysertacji. Ze względu na temat pracy, zagadnienie ziarnistości kodu przybliżono w oddzielnym podrozdziale. Przedstawiono zalety i wady podziału drobno- i gruboziarnistego, docelowe architektury przeznaczone dla różnych wielkości ziaren jak również transformacje kodu umożliwiające uzyskanie różnego stopnia ziarnistości. Zaprezentowano podstawowe wskaźniki wykorzystywane w przetwarzaniu równoległym tj. lokalność kodu, przyśpieszenie, efektywność, prawa

Amdahla i Gustafsona, jak również funkcję izoefektywności. Dodatkowo przedstawione zostały najpopularniejsze środowiska programowania równoległego i rozproszonego ze szczególnym naciskiem na standard programowania ze wspólną pamięcią OpenMP - wykorzystywany w trakcie badań eksperymentalnych (patrz rozdział 4).

3. ALGORYTMY WYSZUKIWANIA NIEZALEŻNYCH FRAGMENTÓW KODU

W niniejszym rozdziale przedstawiono algorytmy umożliwiające automatyczne wyszukiwanie niezależnych fragmentów kodu dla pętli dowolnie zagnieżdżonych. Zaproponowano dwa podejścia do wyznaczania niezależnych fragmentów kodu. Pierwsze podejście to zbiór algorytmów (patrz rozdziały 3.3.2, 3.3.3, 3.3.4) dedykowany dla pętli idealnie zagnieżdżonych, gdzie dla wszystkich instrukcji poziom zagnieżdżenia pętli jest identyczny. Drugie podejście to zbiór algorytmów (patrz rozdziały 3.4.1, 3.4.2, 3.4.3) odnoszący się do pętli nieidealnie zagnieżdżonej, gdzie wymiar przestrzeni dla każdej instrukcji może być różny. W pierwszym przypadku niezależne fragmenty kodu wyznaczane są między pojedynczymi iteracjami. Dla pętli nieidealnie zagnieżdżonych poszczególne instrukcje mogą znajdować się w różnych gniazdach pętli, w związku z tym, niezależne fragmenty kodu wyznaczane są między poszczególnymi instancjami instrukcji. W ogólnym przypadku zaproponowane algorytmy dla pętli nieidealnie zagnieżdżonych mogą zostać wykorzystane również dla pętli dowolnie zagnieżdżonych (dla pętli idealnie zagnieżdżonych każda instrukcja, pomimo tego samego wymiaru, będzie rozpatrywana oddzielnie).

Niniejszy rozdział zawiera szczegółowy opis zaproponowanych algorytmów automatycznego wyszukiwania niezależnych fragmentów kodu. Do każdego algorytmu dołączono przykład wraz z omówieniem kolejnych kroków. Opisane zostały dane wejściowe jak i wyjściowe algorytmów. Dodatkowo przedstawiony został przepływ danych między poszczególnymi algorytmami.

3.1. Projektowanie równoległych algorytmów

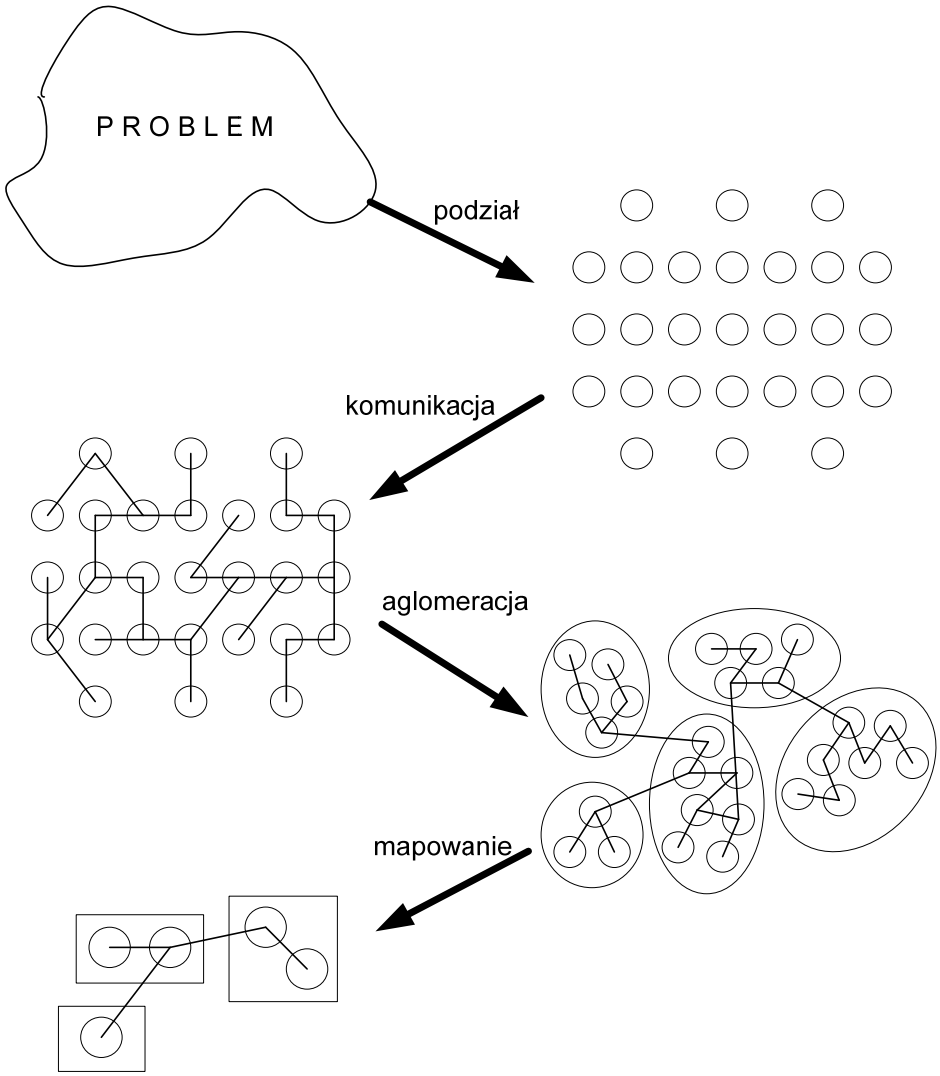
Większość problemów programistycznych może posiadać wiele rozwiązań równoległych, różniących się w znacznym stopniu od siebie, przy jednoczesnym zachowaniu poprawności uzyskiwanych wyników. Mnogość rozwiązań równoległych, jednego problemu sekwencyjnego, skutkuje możliwością wykonania obliczeń na różnych architekturach systemów wieloprocesorowych. Zgodnie z metodologią PCAM zaproponowaną w [34], przedstawioną na rys. 3.1, tworzenie algorytmów równoległych składa się z czterech kroków:

1. Podział (ang. *partitioning*) – obliczenia do wykonania oraz dane na których te obliczenia mają zostać wykonane poddane są podziałowi na mniejsze zadnia. Na tym etapie kwestie praktyczne (np. ilość procesorów) są ignorowane, uwaga skupia się na wyznaczeniu możliwej do uzyskania równoległości.
2. Komunikacja (ang. *communication*) – wyznaczana jest niezbędna komunikacja między zadaniami w celu ich prawidłowego wykonania. Definiowane są odpowiednie algorytmy oraz struktury umożliwiające komunikację.
3. Aglomeracja (ang. *agglomeration*) – zadania i komunikacja wyznaczone w poprzednich krokach oceniane są pod względem wymaganej wydajności oraz kosztów implementacji. Jeśli jest to wymagane zadania łączone są w większe w celu zwiększenia wydajności lub zredukowania kosztów implementacji.
4. Mapowanie (ang. *mapping*) – zadania mapowane są na procesowy w sposób pozwalający na maksymalizację utylizacji procesorów oraz minimalizację kosztów komunikacji. Mapowanie może zostać wykonane w sposób statyczny bądź w trakcie wykonania wykonywania obliczeń poprzez algorytmy balansujące obciążeniem (ang. *load-balancing algorithms*).

W pierwszych dwóch krokach główny nacisk położony jest na konkurencyjność oraz skalowalność algorytmów. Natomiast w dwóch ostatnich krokach uwaga skupiona się na kwestiach dotyczących wydajności (m.in. lokalność danych, koszty komunikacji, tworzenia i zarządzania wątkami).

W kontekście metodologii PCAM, zaproponowane algorytmy pozwalają na podział problemu na mniejsze zadania (fragmenty), nie zawierające komunikacji między sobą – odpowiedniki dwóch pierwszych kroków. Zdania takie mogą zostać poddane

aglomeracji w celu zwiększenia ziarna obliczeń (krok trzeci). Dowolna technika pozwalająca na mapowanie niezależnych zadań może zostać zastosowana w celu przeprowadzania ostatniego kroku.



Rysunek 3.1. Metodologia projektowania programów równoległych [34]

3.2. Idea wyznaczania niezależnych fragmentów kodu

Wyznaczanie niezależnych fragmentów kodu dla dowolnie zagnieżdżonych pętli, między iteracjami jak i instancjami instrukcji, w ogólnym przypadku można przedstawić, jako zbiór następujących po sobie kroków:

1. Wyznaczenie relacji
2. Redukcja nadmiarowych relacji zależności
3. Wyznaczenie podprzestrzeni z niezależnymi fragmentami kodu
4. Wyznaczenia wspólnych iteracji / instancji instrukcji

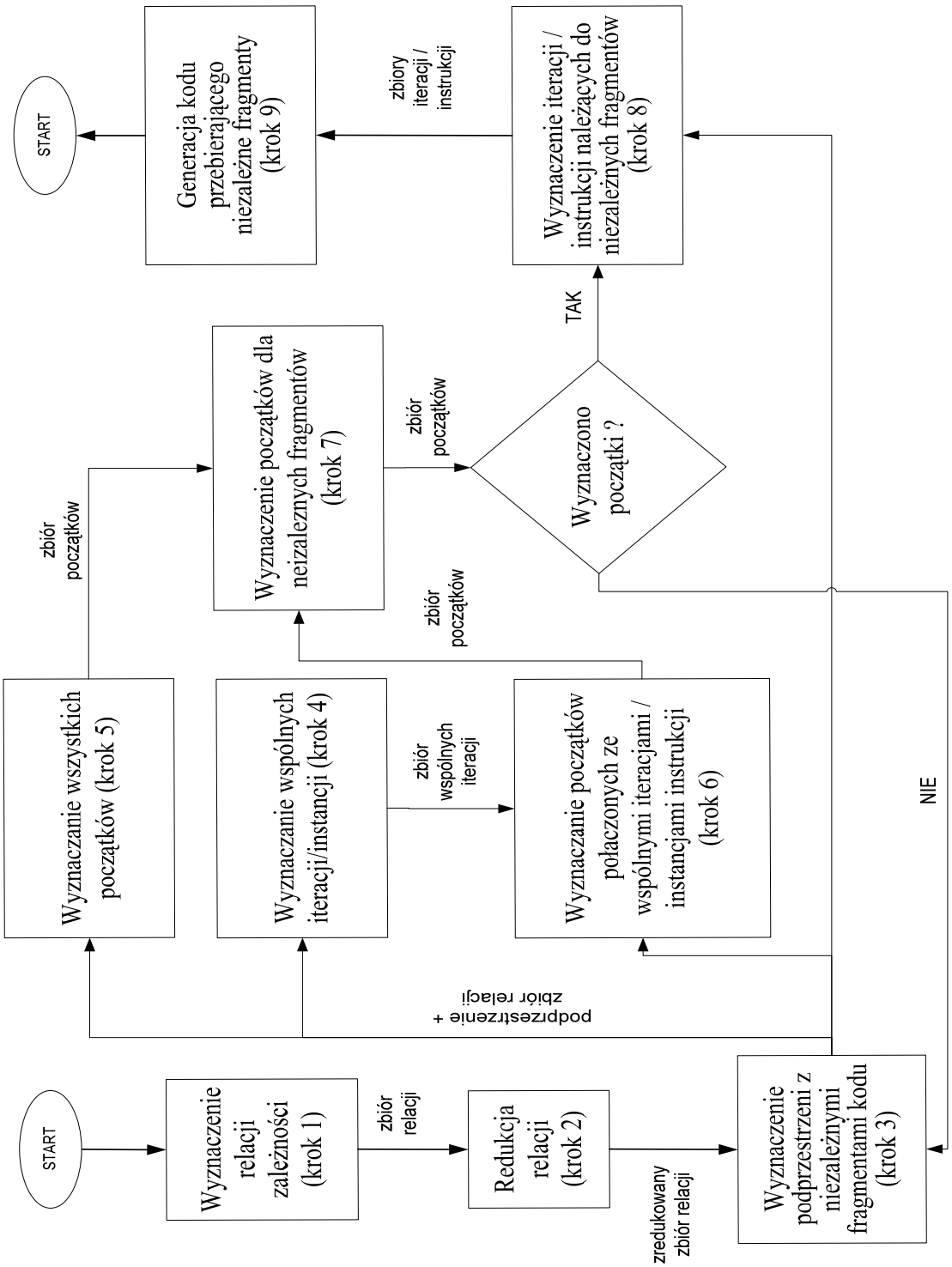
5. Wyznaczenie wszystkich początków niebędących jednocześnie końcami
6. Wyznaczenie początków krańcowych mających połączenie ze wspólnymi iteracjami / instancjami instrukcji
7. Wyznaczenie początków niezależnych fragmentów kodu
8. Wyznaczenie iteracji / instrukcji należących do niezależnych fragmentów
9. Generacja kodu przebierającego niezależne fragmenty

W zależności od typu pętli realizacja powyższych kroków przebiega na poziomie instrukcji lub iteracji. Dla pętli idealnie zagnieżdżonej niezależne fragmenty kodu wyznaczane są na poziomie iteracji, tzn. instrukcje należące do jednej iteracji są niepodzielne - atomowe. W odmienny sposób przebiega analiza dla pętli nieidealnie zagnieżdżonych, gdzie każda instrukcja rozpatrywana jest oddzielnie. Ze względu na komplikację realizacji powyższych kroków, realizacja ich przedstawiona w postaci sekwencji algorytmów. Ułatwia to w znaczący sposób zrozumienie i analizę zaproponowanego podejścia. Realizacja kroków 1 i 2 opisana został w podrozdziale 3.4.1. Kroki 4-7 realizowane są przez algorytmy 3.3 oraz 3.6 opisanych odpowiednio w rozdziałach 3.4.2 oraz 3.5.1. Wyznaczanie podprzestrzeni z niezależnymi fragmentami kodu (krok 3) realizowane jest przez oddzielny algorytm (patrz rozdziały 3.4.3, 3.5.3 i 3.5.4). Pozostałe kroki (tj. 8-9) realizowane są przez kolejny algorytm (patrz rozdziały 3.4.3 oraz 3.5.2). Podział zadań oraz przepływ danych między poszczególnymi krokami zaprezentowano na rys. 3.2.

Zadaniem drugiego kroku jest zredukowanie nadmiarowych zależności wyznaczonych w trakcie analizy zależności. Dla pętli idealnie zagnieżdżonych różne relacje opisują te same zależności. Jak już wcześniej wspomniano, analiza pętli idealnie zagnieżdżonych opiera się na za założeniu, iż pojedyncza iteracja jest niepodzielna (atomowa). W związku z powyższym, relacje opisujące zależności między instrukcjami, gdzie początki występuję w iteracji ‘m’, a końce w iteracji ‘n’, tworzą zbiór relacji nadmiarowych. Relacje takie można zredukować do jednej zależności. Analogiczna sytuacja występuje dla pętli nieidealnie zagnieżdżonych, jednakże w tym przypadku częścią atomową jest pojedyncza instancja instrukcji.

Kolejne kroki (tj. 4-5) mają na celu wyznaczenie początków dla niezależnych fragmentów kodu. Niemożliwe jest wyznaczenie początków w bezpośredni sposób, w związku z tym niezbędne jest wykonanie wstępnych obliczeń, aby ostatecznie uzyskać początki dla niezależnych fragmentów. Wyznaczenie wspólnych iteracji / instancji instrukcji będących częścią wspólną różnych relacji zależności, pozwala na wyznaczenie zależnych początków. Mając do dyspozycji zbiory wszystkich początków

i zależnych początków, możliwe jest poprzez obliczenie różnicy między tymi dwoma zbiorami, wyznaczenie zbioru początków dla niezależnych fragmentów kodu.



Rysunek 3.2. Schemat ideowy algorytmów (przepływ danych) [op. własne]

Jeśli wykonanie powyższych kroków nie doprowadziło do wyznaczenia początków dla niezależnych fragmentów kodu, to w kroku 3 zaproponowano wyznaczenie

niezależnych fragmentów kodu w podprzestrzeniach. Zależności występujące w całej przestrzeni iteracji, ze względu na wspólne końce, bardzo często uniemożliwiają wyznaczenie niezależnych fragmentów kodu. Poprzez zawężenie rozpatrywanej przestrzeni do mniejszych fragmentów, w których pewna liczba zależności może zostać pominięta w stosunku do całej podprzestrzeni, możliwe jest wyznaczenia niezależnych fragmentów kodu. Dla wyznaczonych podprzestrzeni niezbędne jest przeprowadzenie kroków 4-5, w celu wyznaczenia początków dla każdej z podprzestrzeni. Dzięki takiemu podejściu można zwiększyć ilość pętli, dla których możliwe jest wyznaczenie niezależnych fragmentów kodu. Potwierdzenie słuszności powyższego znajduje odzwierciedlenie w badaniach eksperymentalnych w rozdziale 4.

Ostatnie dwa kroki odpowiedzialne są za wyznaczenie iteracji / instancji instrukcji należących do każdego z niezależnych fragmentów kodu oraz generację kodu przebierającego wyznaczone iteracje. Wygenerowany kod musi zagwarantować przebieranie niezależnych fragmentów w sposób równoległy dla pojedynczej podprzestrzeni, o ile podprzestrzenie występują. Z drugiej strony elementy (iteracja / instancje instrukcji) wewnątrz niezależnego fragmentu muszą być przebierane sekwencyjnie z zachowaniem porządku leksykograficznego. W przypadku występowania podprzestrzeni, niezbędne jest zagwarantowanie przebierania ich zgodnie z porządkiem topologicznym w grafie zależności.

3.3. Pojęcia związane z wyznaczaniem niezależnych fragmentów kodu

Operacją T(I) bądź instancją instrukcji T dla iteracji I, nazywamy pojedyncze wykonanie instrukcji T dla określonej iteracji I z przestrzeni iteracji pętli.

Dwie operacje T1(I) oraz T2(J) (możliwe jest aby T1=T2 i I ≠ J lub T1 ≠ T2 i I=J) są zależne (istnieje zależność między T1(I) a T2(J)) jeśli obydwie odwołują się do tej samej komórki pamięci oraz przynajmniej jedno z tych odwołań jest zapisem. T1(I) oraz T2(J) traktujemy odpowiednio jako początek i koniec zależności, zapewniając że T1(I) jest wykonane przed T2(J). Istnieją dwie kategorie zależności w pętlach programowych:

- zależności wewnątrz iteracji,
- zależności między iteracjami.

Zależność między dwoma instancjami instrukcji T1(I) oraz T2(J) jest zależnością wewnątrz iteracji jeśli I = J (T1 i T2 wykonywane są dla tej samej iteracji). W przeciwnym wypadku (I ≠ J) zależność zachodzi między instrukcjami pętli dla różnych iteracji [27].

Podejście zaprezentowane w niniejszej pracy wymaga jednoznacznej (dokładnej) reprezentacji dla każdego rodzaju zależności. Niezbędne jest zastosowanie dokładnej analizy umożliwiającej wyznaczenie zależności jedynie w przypadku, gdy taka zależność zachodzi. W ogólnym przypadku dowolna technika umożliwiająca wyznaczenie dokładnych zależności może zostać użyta w celu implementacji zaproponowanych algorytmów. Jednakże opis algorytmów oraz wykonanie eksperymentów zależne jest od formatu, w jakim wybrana analiza zależności reprezentuje wyznaczone zależności.

Do opisu oraz implementacji algorytmów wybrana została analiza zależności zaproponowana przez Pugha oraz Wonncotta [81]. Zależności reprezentowane są przez relacje zależności zbudowane z formuł Presburgera. Formuły Presburgera zbudowane są z ograniczeń liniowych nad zmiennymi całkowitymi, logicznych połączeń, kwantyfikatora ogólnego (ang. *universal quantifier*) oraz kwantyfikatora szczegółowego (ang. *existantial quantifier*) [81].

W niniejszej pracy należy rozróżnić zależność między parą operacji (początek i koniec), a relacją zależności reprezentującą wszystkie zależności pomiędzy instancjami iteracji wywołanych przez pojedynczą instrukcję (zależność między jedną instrukcją – selfdependences) lub parą instrukcji.

Petit, narzędzie akademickie [58], pozwala na wyznaczenie relacji zależności zgodnie z analizą zależności zaproponowana przez Pugha oraz Wonnacotta. Narzędzie Omega Calculator [58] zostało użyte do wykonania eksperymentów pozwalających na weryfikacje poprawności zaproponowanego podejścia. Algorytmy bazują głównie na operacjach przedstawionych w tab. 2.1 (patrz rozdział 2), które zostały dokładnie omówione w publikacji [58]. Dodatkowo należy zaznaczyć, iż istnieją dwie relacje odnoszące się do tranzytywnego domknięcia: tranzytywne pozytywne domknięcie (R+) oraz tranzytywne domknięcie ($R^* = R+ \cup I$, gdzie I to relacja tożsamościowa). Relacja $R:= \{[i] \rightarrow [i+1]: 1 \leq i <= 3\}$ oraz R+ i R* utworzone na bazie R przedstawione zostały na rys. 3.3.

Krańcowy początek (ang. *ultimate source*) zależności to początek niebędący końcem żadnej innej zależności. Jeśli koniec nie jest końcem innej zależności to taki koniec określamy, jako koniec krańcowy (ang. *ultimate destination*). Krańcowe początki oraz końce zależności reprezentowane przez relację R mogą być wyznaczone poprzez obliczenie odpowiednio: (domain R - range R) i (range R - domain R).

W dysertacji przyjęto następująca definicję fragmentu kodu (ang. *slice*).

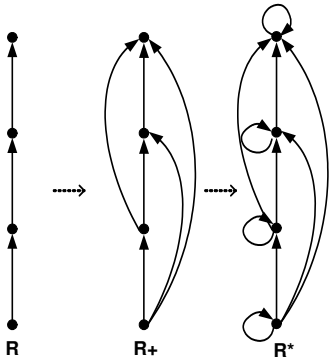
Definicja 1. Fragment kodu (ang. *slice*) jest zbiorem operacji zawierającym dokładnie jeden krańcowy początek oraz wszystkie końce zależności tranzytywnie zależne od początku krańcowego.

Fragment kodu jest wolny od synchronizacji (ang. *synchronization-free slice*) jeśli nie ma zależności między nim a operacjami należącymi do innych fragmentów kodu. Zaproponowane rozwiązanie pozwala na wyznaczenie fragmentów kodu pozbawionych synchronizacji, które nie zawierają końców należących do więcej niż jednej zależności, tj. każdy luźno połączony komponent (ang. *weakly connected component*) w grafie zależności, jeśli tworzy łańcuch lub drzewo.

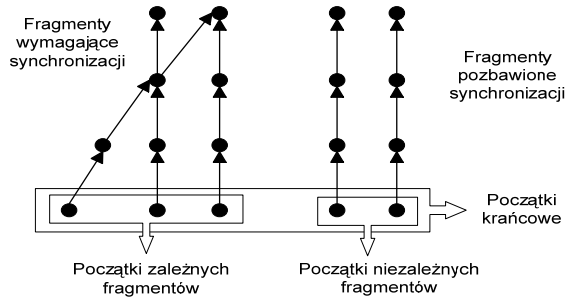
Zgodnie z rys. 3.3, w celu wyznaczenia operacji należących do niezależnego fragmentu kodu opisanego przez relację R, należy obliczyć przeciw dziedzinę (ang. *range*) dla tranzytywnego domknięcia pochodzącego z relacji R.

Definicja 2. Początek niezależnego fragmentu kodu jest to początek krańcowy zawierający się w tym fragmencie, tj. leksykograficznie minimalna operacja spośród wszystkich operacji należących do fragmentu kodu.

Graficzna interpretacja powyższej definicji przedstawiona została na rys. 3.4.



Rysunek 3.3. Relacja zależności R, tranzytywne pozytywne domknięcie oraz tranzytywne domknięcie (w kolejności od lewej) [op. własne]



Rysunek 3.4. Początki slice'ów, krańcowe początki, fragmenty wymagające i pozbawione synchronizacji [op. własne]

Należy rozróżnić graf reprezentujący wszystkie zależności występujące w pętli, a graf reprezentujący zredukowany graf, którego definicja przedstawiona została poniżej.

Definicja 3. Dla pętli zawierającej q instrukcji, zredukowany graf zależności (ang. *reduced dependence graph – RDG*) to multigraf, w ogólnym przypadku cykliczny, zbudowany z wierzchołków reprezentujących każdą instrukcję s_i, i=1,2,...,q oraz

krawędzi łączących wierzchołki s_i i s_j zgodnie z relacjami zależności $R_{s_i s_j}$, i, j ∈ [1,q], wyznaczonymi w trakcie analizy zależności.

Połączony zredukowany graf zależności (ang. *connected reduced dependence graphs – CRDG*) to graf, w którym istnieje ścieżka między wszystkimi parami wierzchołków lub w przypadku specyficznego grafu CRDG – ściśle połączonego grafu (ang. *strongly connected component, SCC*) – to podgraf składający się z maksymalnego podzbioru wierzchołków oraz krawędzi grafu, gdzie dla każdej pary wierzchołków istnieje skierowane połączenie (ang. *directed path*).

3.4. Pętle idealnie zagnieżdżone

Zaproponowane poniżej algorytmy umożliwiają wyznaczenie niezależnych fragmentów kodu. Jak już wcześniej zostało wspomniane, dzięki temu, iż każda instrukcja znajduje się w tym samym gnieździe (posiada ten sam wymiar) analiza odbywa się na poziomie iteracji. Wyznaczenie początków niezależnych fragmentów kodu pozwala na wyznaczenie iteracji należących do każdego niezależnego fragmentu przestrzeni iteracji pętli (patrz rozdział 3.4.3). Kod przebierający iteracje dla pojedynczego niezależnego fragmentu nie wymaga żadnych synchronizacji (o ile zostanie wykonany w sposób sekwencyjny). Dzięki takiemu podejściu możliwe jest zwiększenie lokalności kodu oraz zredukowanie zapotrzebowania na pamięć.

3.4.1 Redukcja nadmiarowych zależności

Do analizy zależności dla pętli idealnie zagnieżdżonych rozpatrywane są jedynie relacje zachodzące miedzy iteracjami. Iteracje pętli są niepodzielne, w związku z tym zależności występujące między poszczególnymi instrukcjami w pojedynczej iteracji mogą zostać pominięte. W celu zredukowania nadmiarowych relacji zaproponowano dwa podejścia:

1. Redukcja powtarzających się zależności w relacjach (algorytm 3.1)
2. Redukcja relacji opisujących zależności o równoległych wektorach dystansu (algorytm 3.2).

Ze względu na atomowość iteracji, relacje opisujące zależności między dwoma różnymi instrukcjami dla tych samych iteracji są nadmiarowe. W pierwszym przypadku (patrz algorytm 3.1) ze zbioru zawierającego n relacji (zgodnych z powyższym opisem) pozostanie jedynie jedna relacja, gdzie n > 1.

Algorytm 3.1 Redukcja nadmiarowych zależności

Wejście: R_1, R_2, ..., R_m – relacje opisujące zależności między iteracjami pętli (wyjście z Petita), gdzie m to liczba relacji

Wyjście: R_1, R_2, ... ,R_n ($R_i \neq R_j$, $i \neq j$, $i,j \leq n$, $n \leq m$) – zbiór zawierający zredukowana liczbę relacji

Dla każdej pary R_i, R_j ($i \neq j$, $i,j \leq m$):

1. Wyznacz relacje R_{ij} opisującą wspólne zależności: $R_{ij} = R_i \wedge R_j$.
2. Jeśli R_{ij} == False to relacje nie posiadają wspólnych zależności – przejdź do kolejnej pary relacji. Jeśli $R_{ij} \neq$ False to krok 3.
3. Oblicz:
   $Dif_i = R_i - R_{ij}$,
   $Dif_j = R_j - R_{ij}$.
4. Jeśli Dif_i == False to odrzuć relację R_i i przejdź do kolejnej pary relacji. Jeśli Dif_j == False to odrzuć relację R_j i przejdź do następnej pary relacji.
5. Jeśli żaden z warunków w punkcie 4 nie został spełniony, oznacza to że nie wszystkie zależności reprezentowane relacją R_i pokrywają zależności reprezentowane relacją R_j i na odwrót. W takim przypadku usuwamy zależności R_{ij} z relacji R_i:
   $R_i = R_i - R_{ij}$

W drugim przypadku (algorytm 3.2.) ze zbioru zawierającego n relacji o równoległych wektorach dystansu, możliwe jest zredukowanie relacji zależności, których wektory zależności mogę być przedstawione przez krotności równoległych wektorów zależności pochodzących z innych relacji. Dzięki takiemu podejściu możliwe jest zmniejszenie liczby rozpatrywanych relacji, a przez to wyznaczenie niezależnych fragmentów kodu dla większej liczby pętli.

Algorytm 3.2 Redukcja relacji o równoległych wektorach dystansu

Wejście: R_1, R_2, ... , R_n – relacje opisujące zależności między iteracjami pętli, n – liczba relacji, R_{1vec}, $R_{2vec,}$ ... , R_{nvec} – wektory zależności odpowiadające relacją zależności

Wyjście: R_1, R_2, ... , R_o ($R_i \neq R_j$, $i \neq j$, $i,j \leq o$, $o \leq n$)

Dla każdej pary relacji R_i, R_j ($i \neq j$, $i,j \leq n$), gdzie R_{ivec} i R_{jvec} są wektorami równoległymi wykonaj:

1. Jeśli oba wektory R_{ivec} i R_{jvec} nie są wektorami stałymi to przejdź do kolejnej pary relacji, w przeciwnym razie idź do punktu 2

2. Wyznacz koordynaty wsp_i, wsp_j różne od siebie odpowiednio dla wektorów R_{ivec} oraz R_{jvec}
3. Jeśli oba wektory R_{ivec} i R_{jvec} są stałe to mniejszy wektor oznacz jako R_{min} (wsp_{min}), natomiast drugi wektor oznacz jako R_{max} (wsp_{max}) i przejdź do kroku 5.
4. Jeśli jeden z wektorów jest wektorem zmiennym to jako R_{min} (wsp_{min}) oznacz wektor stały, a R_{max} (wsp_{max}) wektor zmienny i przejdź do 5.
5. Zbadaj czy R_{min} jest krotnością wektora R_{max} poprzez dodanie następującego ograniczenia do R_{max} (tworzony jest nowy wektor R_{temp})

   $R_{temp} = R_{max}$ && Exists(n : n*wsp_{min} = wsp_{max})
6. Jeśli R_{temp} == R_{max} to odrzuć relację o wektorze dystansu R_{max}.

Dla przykładu 3.1 przedstawiono działanie algorytmu 3.1 oraz algorytmu 3.2. Przestrzeń iteracji pętli wraz z zależności przedstawiona została na rys. 3.4a.

for i=1 to 6 do
for j=1 to 6 do
a(i,j) = a(i+2,j)
a(i,j) = a(3*i,3*j)
a(i,j) = a(i+4,j)
endfor
endfor
	$R_1 := \{[i,j] \rightarrow [i+2,j] : 1 \leq i \leq 4 \;\&\&\; 1 \leq j \leq 6\}$;
	$R_2 := \{[i,j] \rightarrow [i+2,j] : 1 \leq i \leq 4 \;\&\&\; 1 \leq j \leq 6\}$;
	$R_3 := \{[i,j] \rightarrow [i+2,j] : 1 \leq i \leq 4 \;\&\&\; 1 \leq j \leq 6\}$;
	$R_4 := \{[i,j] \rightarrow [3i,3j] : 1 \leq i \leq 2 \;\&\&\; 1 \leq j \leq 2\}$;
	$R_5 := \{[i,j] \rightarrow [3i,3j] : 1 \leq i \leq 2 \;\&\&\; 1 \leq j \leq 2\}$;
	$R_6 := \{[i,j] \rightarrow [3i,3j] : 1 \leq i \leq 2 \;\&\&\; 1 \leq j \leq 2\}$;
	$R_7 := \{[i,j] \rightarrow [i+4,j] : 1 \leq i \leq 2 \;\&\&\; 1 \leq j \leq 6\}$;
	$R_8 := \{[i,j] \rightarrow [i+4,j] : 1 \leq i \leq 2 \;\&\&\; 1 \leq j \leq 6\}$;
	$R_9 := \{[i,j] \rightarrow [i+4,j] : 1 \leq i \leq 2 \;\&\&\; 1 \leq j \leq 6\}$;

Przykład 3.1 Przykładowa pętla dla algorytmów 3.1 i 3.2 [op. własne]	Relacje zależności wyznaczone dla przykładu 3.1 [op. własne]

Algorytm 3.1

Wejście: Relacje R_1,R_2, ... R_9

1. Dla relacji R_1 i R_2 wyznaczamy wspólne zależności

   $R_{12} = R_1$ intersection $R_2 = \{[i,j] \rightarrow [i+2,j] : 1 \leq i \leq 4 \;\&\&\; 1 \leq j \leq 6\}$
2. W związku z tym, że $R_{12} \neq$ False to przechodzimy do kroku 3
3. Obliczamy różnice między relacjami R_1 i R_{12}

   $Dif_1 = R_1 - R_{12} = \{[i,j] \rightarrow [Out\_1,j'] : FALSE \}$
4. Dif_1 == False, dlatego odrzucamy relację R_1 i przechodzimy do kolejnej pary relacji.

Analogicznie postępujemy z pozostałymi relacjami. Na wyjściu algorytmu 3.1 uzyskujemy.

Wyjście:

$R_3 := \{[i,j] \rightarrow [i+2,j] : 1 \leq i \leq 4 \;\&\&\; 1 \leq j \leq 6\}$,

$R_6 := \{[i,j] \rightarrow [3i,3j] : 1 \leq i \leq 2 \;\&\&\; 1 \leq j \leq 2\}$,

$R_9 := \{[i,j] \rightarrow [i+4,j] : 1 \leq i \leq 2 \;\&\&\; 1 \leq j \leq 6\}$.

Zgodnie z algorytmem 3.1 dla powyższego przykładu możliwe jest zredukowanie liczby relacji z dziewięciu do trzech. Kolejną redukcję umożliwia algorytm 3.2, przy pomocy, którego możemy zredukować relacje zależności o równoległych wektorach dystansu.

Algorytm 3.2

Wejście:

$R_3 := \{[i,j] \rightarrow [i+2,j] : 1 \leq i \leq 4 \;\&\&\; 1 \leq j \leq 6\}$

$R_6 := \{[i,j] \rightarrow [3i,3j] : 1 \leq i \leq 2 \;\&\&\; 1 \leq j \leq 2\}$

$R_9 := \{[i,j] \rightarrow [i+4,j] : 1 \leq i \leq 2 \;\&\&\; 1 \leq j \leq 6\}$

Dla każdej relacji wyznaczamy zbiory reprezentujące wektory dystansu:

$R_{3vec} := \{[2,0]\}$;

$R_{6vec} := \{[In\_1,In\_2]$: Exists (alpha,beta : 0 = In_1+2alpha && 2beta = In_2 && $2 \leq In\_1 \leq 4$ && $2 \leq In\_2 \leq 4)\}$;

$R_{9vec} := \{[4,0]\}$;

1. Dla relacji R_3, R_9 oraz odpowiadającym im wektorom dystansu R_{3vec}, R_{9vec}, oba wektory są stałe, w związku z tym przechodzimy do kolejnego kroku
2. Wektory R_{3vec} i R_{9vec} różnią się od siebie pierwszymi koordynatami
3. Oba wektory są stałe, jako R_{min} przyjmujemy wektor R_{3vec} ($wsp_{min} = 2$), jako R_{max} wektor R_{9vec} ($wsp_{max} = 4$) i przechodzimy do kroku 5
4. pomijamy ten krok
5. Tworzymy ograniczenie następującej postaci

   $R_{temp} = \{[4,0]$: Exists(n : n*2 = 4)$\}$
6. Ponieważ R_{temp} jest równy R_{max} ($R_{temp} == R_{max}$) dlatego odrzucamy wektor dystansu R_{max} wraz z odpowiadająca mu relacją R9.

Wykonanie algorytmu 3.2 dla pary relacji zależności R_3 i R_6 nie pozwala na zredukowanie żadnej z nich

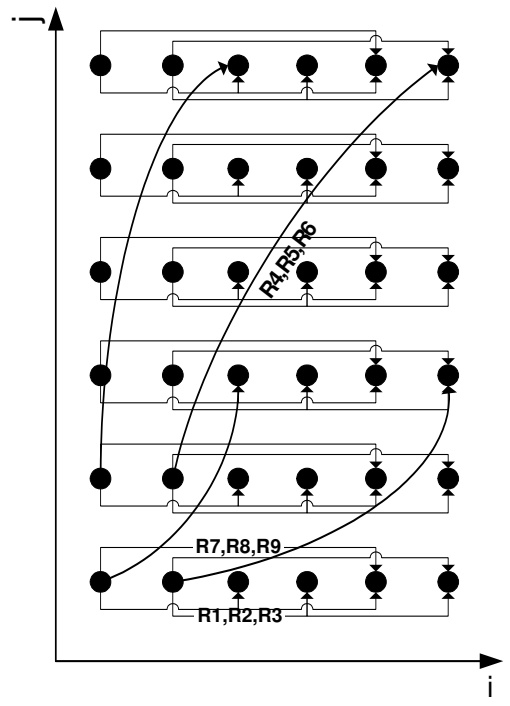
Wyjście:

$R_3 := \{[i,j] \rightarrow [i+2,j] : 1 \leq i \leq 4 \;\&\&\; 1 \leq j \leq 6\}$

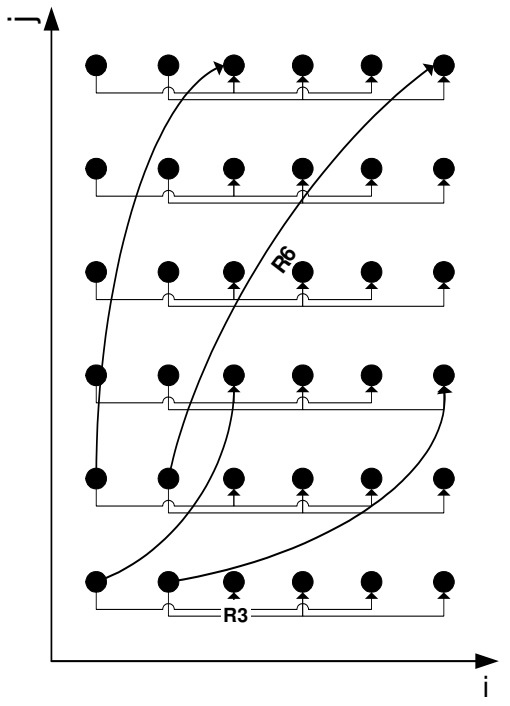
$R_6 := \{[i,j] \rightarrow [3i,3j] : 1 \leq i \leq 2 \;\&\&\; 1 \leq j \leq 2\}$

Zastosowanie zaproponowanych algorytmów dla wybranego przykładu pozwala na zredukowanie liczby zależności z dziewięciu do dwóch, bez utraty informacji o zależnościach między poszczególnymi instrukcjami. Na rys. 3.5a oraz 3.5b

przedstawiono przestrzenie iteracji pętli dla przykładu 3.1 z zależnościami odpowiednio przed oraz po wykonaniu algorytmów 3.1 i 3.2. Redukcja nadmiarowych zależności ma istotny wpływ na ilość możliwych do wyznaczenia niezależnych fragmentów kodu. Często pominięcie redukcji relacji zależności uniemożliwia jakąkolwiek analizę pętli biorąc pod uwagę zaproponowane algorytmy w dalszej części pracy.



Rysunek 3.5a Przestrzeń iteracji pętli wraz z zależnościami dla przykładu 3.1 [op. własne]



Rysunek 3.5b Zależności dla przykładu 3.1 po wykonaniu algorytmów 3.1 i 3.2 [op. własne]

3.4.2. Wyznaczanie początków dla niezależnych fragmentów kodu

W poniższym rozdziale przedstawiony został algorytm umożliwiający wyznaczenie leksykograficznie minimalnej iteracji spośród wszystkich należących do niezależnego fragmentu kodu. Iteracje te stanowią zbiór początków dla kodu niewymagającego synchronizacji z innymi fragmentami kodu.

Idea algorytmu przedstawia się następująco. Na początku wyznaczane są dwa zbiory opisujące iteracje pętli. Pierwszy zbiór zawiera iteracje zależne, natomiast drugi niezależne. Następnie, wykorzystując zbiór iteracji zależnych, wyznaczane są wszystkie początki zależnych jak i niezależnych fragmentów kodu. Zbiór z początkami zostaje podzielony na dwa podzbiory. Pierwszy z nich (oznaczony, jako RSS) zawiera początki dla kodu wymagającego synchronizacji, natomiast drugi (oznaczony, jako SFS) zawiera

początki dla kodu niewymagającego synchronizacji. Liczba iteracji zawartych w zbiorze SFS określa liczbę bloków kodu niewymagającego synchronizacji.

Na użytek poniższego algorytmu należy przyjąć, że:

i. Każda relacja reprezentuje zależności, gdzie każdy koniec ma tylko jeden odpowiadający początek. W przeciwnym wypadku, tzn. jeśli dwa różne końce zależności odnoszą się do wspólnego początku zależności, relacja musi zostać znormalizowana w celu spełniania powyższego warunku (nadmiarowe relacje zależności muszą zostać zredukowane, lub relacja musi zostać podzielona na kilka relacji).

ii. Dwie relacje R_i i R_j ($R_i \neq R_j$) gdzie jedna z nich opisuje zależności, których krańcowe końce są początkami krańcowymi drugiej należy połączyć w jedną relację poprzez zastosowanie unii relacji: $R_{ij} := R_i \cup R_j$.

Algorytm 3.3 Wyszukiwanie początków dla fragmentów kodu.

Wejście: zbiór S_In zawierający znormalizowane relacj R_1, R_2,…,R_m, reprezentujące zależności między iteracjami pętli, gdzie m oznacza liczbę relacji.

Wyjście: zbiór początków dla kodu wymagającego synchronizacji (RSS) oraz zbiór początków dla kodu niewymagającego synchronizacji (SFS).

1. Oblicz unię wszystkich relacji (R), relację odwrotną do relacji R (IR) i zainicjalizuj zbiór zawierający wspólne iteracja jako pusty (CI):

   $R := R_1 \cup R_2 \cup \ldots \cup R_m$;

   IR := inverse R ;

   CI := EMPTY.

2. Wyznacz zbiór wszystkich wspólnych początków i końców, jako zbiór CI w następujący sposób. Jeśli zbiór S_In zawiera jedną relację to przejdź do kroku 3, w przeciwnym razie dla każdej pary relacji R_i i R_j ze zbioru S_In, gdzie $i \neq j$, $i,j \leq m$ wykonaj:

   $CI = CI \cup (\text{domain } R_i \cap \text{domain } R_j) \cup (\text{range } R_i \cap \text{range } R_j)$.

3. Wyznacz początki zależności (I) jako dziedzinę relacji R, końce zależności (J) jako przeciwdziedzinę relacji R oraz początki krańcowe (UDS) jako różnicę między zbiorami I i J:

   I := domain R

   J := ragne R

   UDS := I – J.

4. Jeśli zbiór CI jest pusty (CI == EMPTY) to zbiór UDS zawiera wszystkie początki dla niezależnych bloków kodu. W takim przypadku SFS := UDS, zakończ algorytm. W przeciwnym razie przejdź do następnego kroku.
5. Przy użyciu kroku 5a lub 5b wyznacz zbiór początków należących do bloków wymagających synchronizacji

5a. przy użyciu tranzytywnego domknięcia	5b. nie używając tranzytywnego domknięcia (jedynie dla niesparametryzowanych CI oraz IR)
Oblicz: IR+ := Transitive Closure IR, RSS_1 := IR+(CI) /*stosując relację IR+ na zbirze CI*/	TEMP := CI, RSS_1:= CI, L: Oblicz: TEMP := IR(TEMP) /* stosując relację IR do zbioru TEMP*/ JEŚLI TEMP == EMPTY TO przejdź do kroku 6, w przeciwnym razie $RSS_1 := RSS_1 \cup TEMP$, przejdź do L:

6. Aby wyznaczyć początki zależnych fragmentów kodu (RSS) oblicz część wspólną zbiorów RSS_1 i UDS:
   $RSS := RSS_1 \cap UDS$.
7. Aby wyznaczyć początki dla niezależnych fragmentów kodu (SFS), oblicz część wspólną zbiorów UDS i RSS:
   SFS := UDS – RSS.

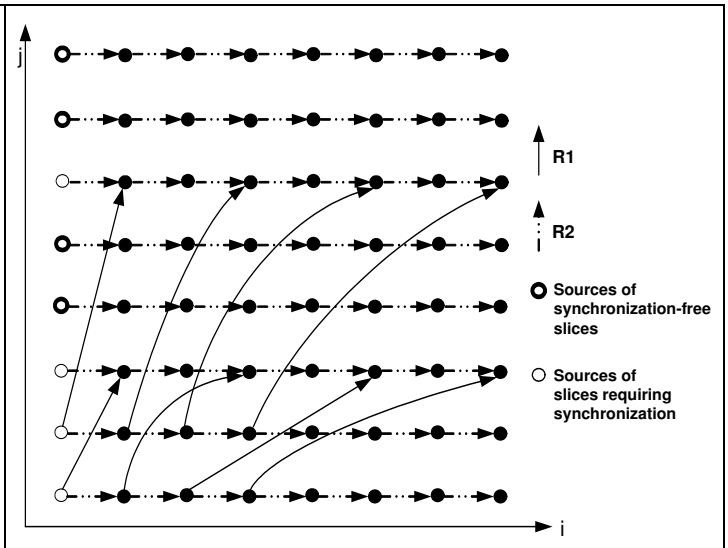
Jeśli zbiór CI opisujący wspólne iteracje jest pusty, wszystkie początki są początkami niezależnych fragmentów kodu. Dla pętli zawierającej wspólne iteracje (CI≠Empty), kroki 1-5 odpowiedzialne są za wyznaczenie iteracji wymagających synchronizacji (zbiór RSS_1). Zbiór RSS_1 zawiera iteracje należące do zbioru CI oraz iteracje w kierunku przeciwnym do kierunku zależności tzn. od końców do początków zależności. Krok 5a powyższego algorytmu może zostać zastosowany do pętli sparametryzowanych jak również niesparametryzowanych w czasie kompilacji jak i wykonania pętli. Krok 5b może zostać zastosowany dla pętli sparametryzowanej jedynie w czasie wykonania kodu, kiedy granice pętli stają się znane.

Wiadomym jest, że tranzytywne domknięcie dla relacji składających się z krotek opisanych wyrażeniami afinicznymi może zawierać wyrażenia nieafiniczne [61]. W związku z powyższym w ogólnym przypadku, tranzytywne domknięcie reprezentowane przez wyrażenia afiniczne jest niemożliwe do obliczenia dla afinicznej relacji zależności. Jednakże jest zawsze policzalne dla pętli afinicznych zawierających

zależności jednorodne [61]. W przypadku, gdy przy użyciu pakietu Omegi Calculatora niemożliwe jest wyznaczenie dokładnego tranzytywnego domknięcia, możliwe jest zastosowanie aproksymacji tranzytywnego domknięcie (dolna lub górna granica) w celu podjęcia próby wyznaczenia początków dla niezależnych fragmentów kodu.

Poniższy przykład demonstruje działanie algorytmu 3.3

```
for(i = 1; i ≤ n; i++)
  for(j = 1; j ≤ n; j++)
  {
     a(2*i, 3*j) = b(i,j)
     b(i+1, j) = a(i, j)
  }
```



Przykład 3.2 Pętla z zależnościami niejednorodnymi

Rysunek 3.6. Zależności dla pętli z przykładu 3.2, gdzie n=8 [op. własne]

Relacje reprezentujące zależności między iteracjami pętli przedstawiono poniżej:

$R_1 := \{[i,j] \rightarrow [2i,3j] : 1 \leq j \;\&\&\; 2i \leq n \;\&\&\; 1 \leq i \;\&\&\; 3j \leq n\}$,

$R_2 := \{[i,j] \rightarrow [i+1,j] : 1 \leq i < n \;\&\&\; 1 \leq j \leq n\}$.

Rysunek 3.6 przedstawia zależności występujące w pętli z przykładu 3.2. Przebieg algorytmu 3.3 dla powyższego przykładu przedstawia się następująco:

1. Wykonanie wstępnych obliczeń

   $R := R_1 \cup R_2 := \{[i,j] \rightarrow [2i,3j] : 1 \leq j \;\&\&\; 2i \leq n \;\&\&\; 1 \leq i \;\&\&\; 3j \leq n\}$ union $\{[i,j] \rightarrow [i+1,j]: 1 \leq i < n \;\&\&\; 1 \leq j \leq n\}$,

   IR := inverse R := {[In_1,j] → [i,j'] : j = 3j' && In_1 = 2i && 1 ≤ j' && 2i ≤ n && 1 ≤ i && 3j' ≤ n} union {[In_1,j] → [In_1-1,j] : 2 ≤ In_1 ≤ n && 1 ≤ j ≤ n},

   CI:=EMPTY.

2. Wyznaczenie wspólnych iteracji

   CI := EMPTY ∪ (domain R_1 ∩ domain R_2) ∪ (range R_1 ∩ range R_2) = {[i,j]: 1 ≤ j && 2i ≤ n && 1 ≤ i && 3j ≤ n} union {[i,j]: Exists (alpha,beta : j = 3alpha && 2beta = i && 2 ≤ i ≤ n && 3 ≤ j ≤ n)}.

3. Wyznaczenie początków

   I := domain R, J := range R,

   UDS := I – J := {[1,j]: 1 ≤ j ≤ n && 2 ≤ n}.
4. Zbiór CI nie jest zbiorem pustym, w związku z tym przechodzimy do kroku 5a (wykorzystanie tranzytywnego domknięcia).
5. TIR := IR+ union {[i,j]→[i,j]} := {[2,j] → [1,j] : n = 2 && 1 ≤ j ≤ 2} union {[i,j] → [i,j] } union {[i,j] → [i',j'] : 1 ≤ j' ≤ j ≤ n && i'+2 ≤ i ≤ n && 1 ≤ i' && 3j ≤ 2n+3j' && 2i ≤ 4+n+2i' && UNKNOWN} union ...

Relacja TIR zawiera w ograniczeniach warunki UNKNOWN, dlatego do dalszej analizy posługujemy się przybliżeniem relacji TIR, aby obliczyć dolną i górną granicę relacji. Przy użyciu górnej granicy (UNKNOWN=True) do stworzenia zbioru SFS zgodnie z zaproponowanym algorytmem 3.3 uzyskujemy następujący zbiór:

SFS := {[1,j]: 2 ≤ n ≤ 3, 3j-1 && j ≤ 2} union {[1,j]: Exists (alpha : 4, j ≤ n ≤ 3alpha+2 && 3alpha < j)}.

Powyższy zbiór SFS nie zawiera wszystkich początków dla kodu niewymagającego synchronizacji. Obliczając dolną granicę relacji TIR (UNKNOWN=False), LTIR:

LTIR := lower_bound TIR := {[2,j] → [1,j] : n = 2 && 1 ≤ j ≤ 2} union {[i,j] → [i,j]} union {[i,j] → [i',j'] : j = 3j' && 2i' ≤ i ≤ n && 3j' ≤ n && 1 ≤ j' && 1 ≤ i'} union {[i,j] → [i',j'] : j = 9j' && 4i' ≤ i ≤ n && 9j' ≤ n && 1 ≤ j' && 1 ≤ i'} union {[i,j] → [i',j] : 1 ≤ i' < i ≤ n && 1 ≤ j ≤ n && 3 ≤ n},

możliwe jest wyznaczenie wszystkich początków dla niezależnych fragmentów kodu, poniższy zbiór SFS przedstawia początki:

SFS := {[1,j]: Exists (alpha : j, 2 ≤ n ≤ 3j-1 && 3alpha+1 ≤ j ≤ 3alpha+2)}.

3.4.3. Wyznaczanie podprzestrzeni zawierających niezależne fragmentu kodu

Zaproponowany poniżej algorytm znajduje swoje zastosowanie dla pętli, w których niemożliwe jest wyznaczenie niezależnych fragmentów kodu w całej przestrzeni iteracji pętli. Algorytm pozwala na wyznaczenie podprzestrzeni w całej przestrzeni iteracji pętli, w których możliwe jest wyznaczenie niezależnych fragmentów kodu. Podprzestrzenie wyznaczane są poprzez analizę wektorów zależności. Zerowe koordynaty w wybranym zbiorze wektorów pozwalają na wyznaczenie niezależnych

fragmentów kodu dla sekwencyjnych podprzestrzeni wyznaczonych przez resztę koordynat wektorów (koordynaty mogą być niezerowe).

Algorytm 3.4 Wyznaczanie podprzestrzeni zawierających niezależne fragmenty

Wejście: pętla źródłowa; zbiór S_In zawierający relacje reprezentujące zależności między iteracjami pętli R_1, R_2,..., R_m oraz zbiór D zawierający wektory zależności D_1, D_2,...., D_m dla relacji ze zbioru R, gdzie m oznacza liczbę relacji.

Wyjście: współrzędne pętli definiujące podprzestrzenie, gdzie możliwe jest wyznaczenie początków dla niezależnych fragmentów kodu; zbiór S_Out zawierający relacje umożliwiające wyszukanie niezależnych fragmentów kodu w podprzestrzeniach; początki dla niezależnych fragmentów kodu opisanych przez relacje w zbiorze S_Out.

1. Jeśli w zbiorze D istnieje jeden lub więcej wektorów zależności z jedną lub więcej współrzędnymi o zerowych wartościach to przejdź do kolejnego kroku. W przeciwnym razie algorytm nie wyznacza podprzestrzeni z niezależnymi fragmentami kodu, zakończ algorytm.
2. Zainicjalizuje zbiór S_Out := EMPTY
   2.1. Do zbioru S_Out dodaj relację ze zbioru S, dla której w wektorze zależności D_i występuje minimalna sekwencja zerowych koordynat ze zbioru wszystkich wektorów zależności występujących w zbiorze D (jeśli występuje więcej niż jedna taka relacja to wybierz dowolną z nich, jeśli dla wybranej relacji wektor zależności zawiera więcej niż jedną sekwencję zerowych koordynat to wybierz jedną z nich)
   2.2. Dodaj do zbioru S_Out wszystkie relacje ze zbioru S_In, dla których odpowiednie wektory zależności mają identyczne zerowe koordynaty jak wektor D_i wybrany w poprzednim kroku (reszta współrzędnych wektorów może być dowolnej wartości)
   2.3. Jeśli zbiór S_Out zawiera tą samą liczbę relacji zależności to zakończ działanie algorytmu, niemożliwe jest wyznaczenie podprzestrzeni z niezależnymi fragmentami kodu. W przeciwnym razie przejdź do kolejnego kroku.
3. Zastosuj algorytm 3.3 dla zbioru S_Out

   Jeśli dla zbioru S_Out wyznaczono niezależne fragmenty kodu, to sprawdź czy pętla źródłowa jest semantycznie zgodna z pętlą, której ciało jest zgodne z pętlą źródłową i której pętle zewnętrzne i wewnętrzne reprezentowane są przez współrzędne tworzące minimalną zerową sekwencję koordynat wektora D_i, oraz reszty odpowiednio (sprawdzenie poprawności transformacji zamiany kolejności

pętli); dowolna znana technika może zostać użyta np. opisana w [1].

Jeśli powyższe jest spełnione, to zapamiętaj zerowe współrzędne z wybranego ciągu zerowych współrzędnych w wektorze D_i, początki dla niezależnych fragmentów kodu dla zbioru S_Out oraz sam zbiór S_Out i zakończ algorytm. W przeciwnym razie przejdź do kolejnego kroku.

4. Sprawdź czy w zbiorze D istnieje wektor zależności D_i z inna sekwencją zerowych koordynat. Jeśli tak to, wyzeruj zbiór S_Out (S_Out := EMPTY), dodaj odpowiadającą wybranemu wektorowi relację R_i do zbioru S_Out i przejdź do kroku 2.2. W przeciwnym razie algorytm nie pozwala na wyznaczenie podprzestrzeni zawierających niezależne fragmenty kodu, zakończ algorytm.

Wybór minimalnej sekwencji zerowych koordynat w wektorze D_i (krok 2.1) ma na celu umożliwienie wyznaczenia maksymalnej liczby niezależnych fragmentów kodu. Relacje zależności nie zawierające się w zbiorze S_Out, ze względu na niezerowe współrzędne w odpowiednich współrzędnych wektorów zależności, nie opisują zależności w wyznaczonych podprzestrzeniach. Wszystkie te zależności posiadają początki i końce leżące w różnych podprzestrzeniach (początki i końce zależności nie leżą w tej samej podprzestrzeni). W związku z tym, zależności te mogą zostać pominięte w trakcie wyznaczania niezależnych fragmentów kodu w podprzestrzeniach pętli. Jednakże podczas generacji kodu, niezbędne jest zagwarantowanie, aby przebieranie podprzestrzeni przebiegało sekwencyjnie zgodnie z porządkiem leksykograficznym, dzięki temu zapewniamy, iż wspomniane wcześniej zależności nie zostaną naruszone.

Poniższy przykład przedstawia działanie algorytmu 3.4.

for(i = 1; i ≤ n; i++)
  for(j = 1; j ≤ n; j++)
  for(k = 1; k ≤ n; k++){
  a(i,j,k)=a(i,j-1,k)
  b(i,j,k)=b(i,j,k-1)
  }
	$R_1 := \{[i,j,k] \rightarrow [i,j+1,k] : 1 \leq i \leq n \;\&\&\; 1 \leq j < n \;\&\&\; 1 \leq k \leq n\}$,
	$D_1 := \{[0,1,0]\}$,
	$R_2 := \{[i,j,k] \rightarrow [i,j,k+1] : 1 \leq i \leq n \;\&\&\; 1 \leq j \leq n \;\&\&\; 1 \leq k < n\}$.
	$D_2 := \{[0,0,1]\}$.

Przykład 3.3 Pętla zawierająca dwie zależności	Relacje zależności wraz z odpowiadającymi im wektorami zależności dla przykładu 3.3

Dane wejściowe dla powyższej pętli to zbiory S_In = { R_1, R_2 } oraz D = { D_1, D_2 }. Postępując zgodnie z algorytmem 3.4 otrzymujemy.

1. W zbiorze D istnieje wektor zależności z zerowymi koordynatami, w związku z tym przechodzimy do kroku 2.
2. S_Out: = EMPTY.

2.1. Wybieramy indeks k reprezentowany przez zerową koordynate w wektorze D1, w związku z czym zbiór S_Out zawiera jedną relacje R_1 (S_Out := { R_1 }),

2.2. W zbiorze S_In tylko jeden wektor ma zerowe koordynaty w miejscu zmiennej indeksowej k, dlatego do zbioru S_Out nie dodajemy żadnej nowej relacji (S_Out: = { R_1 }),

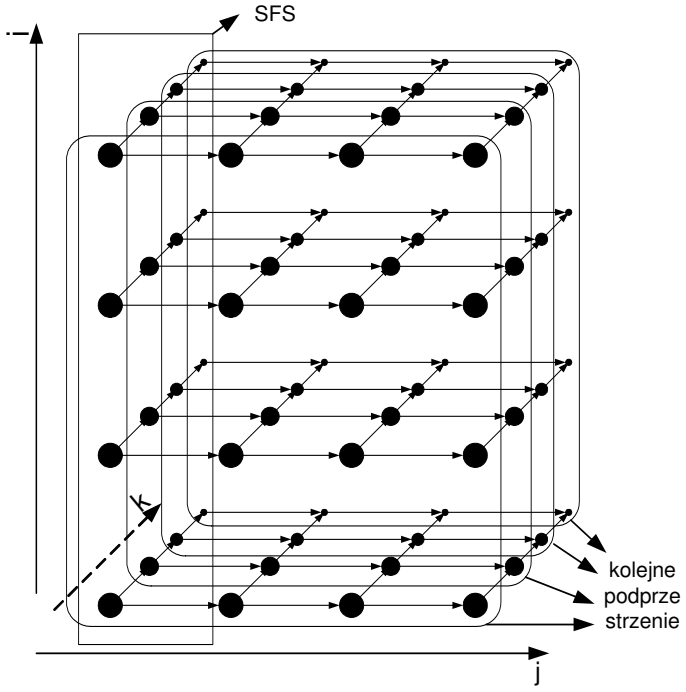
2.3. Liczba relacji zależności zawartych w zbiorach S_Out oraz S_In jest różna, w związku z tym przechodzimy do kroku 3.

3. Wykonując algorytm 3.3 dla zbioru relacji S_Out uzyskujemy poniższy zbiór opisujący początki dla niezależnych fragmentów kodu:

   SFS: = { [i,j,k] :j = 1 && 1 ≤ i ≤ n && 1 ≤ k ≤ n && 2 ≤ n }.

3.1. Zamiana kolejności pętli jest dozwolona, dlatego zapamiętujemy indeks k, zbiór SFS oraz zbiór S_Out.

Jak pokazano na powyższym przykładzie, zawężenie przestrzeni iteracji pętli pozwala na wyznaczenie początków dla niezależnych fragmentów kodu, w przypadku gdy analiza zależności dla całej przestrzeni pętli zawodzi. Graficzna reprezentacja podprzestrzeni dla przykładu 3.3 przedstawiona została na rys. 3.7. Początki zawarte w pojedynczej podprzestrzeni wyznaczają liczbę niezależnych fragmentów kodu, które mogą zostać wykonane w sposób równoległy bez jakiejkolwiek synchronizacji między pozostałymi fragmentami. Przejście między poszczególnymi podprzestrzeniami odbywa się sekwencyjnie, przez co zagwarantowane jest nienaruszenie zależności pominiętych w kroku 2.1 oraz 2.2.



Rysunek 3.7. Przestrzeń iteracji pętli dla przykładu 3.3, gdzie n = 4 [op. własne]

3.4.4. Wyznaczanie oraz generowanie kodu dla niezależnych fragmentów

Poniżej zaprezentowany został sposób wyznaczania zbiorów zawierających iteracje dla niezależnych fragmentów kodu. Przedstawiono również sposób generowania kodu przebierającego niezależne fragmenty wraz z iteracjami należącymi do nich zgodnie z porządkiem leksykograficznym.

Zapis postaci {X,Y} oznacza produkt wektorowy zbiorów X, Y , tzn. {X, Y} := X $\times$ Y: ={[x,y] | x $\in$ X, y $\in$ Y}.

W celu zademonstrowania komplikacji podczas generowania kodu skanującego niezależne fragmenty pętli, rozważmy przykładową relację zależności:

$R_1 := \{[i,j] \rightarrow [i+1,j+1] : 1 \leq i < 4 \;\&\&\; 1 \leq j < 4\}$.

Zbiór zawierający początki dla niezależnych fragmentów kodu (SFS), obliczony na podstawie powyższej relacji zależności przedstawia się następująco:

$SFS = (\text{domain } R_1) - (\text{range } R_1) = \{[1,j]: 1 \leq j \leq 3\} \text{ union } \{[i,1]: 2 \leq i \leq 3\}$.

Generacja kodu dla pojedynczego niezależnego fragmentu możliwa jest poprzez zastosowanie relacji R1 na zbiorze SFS:

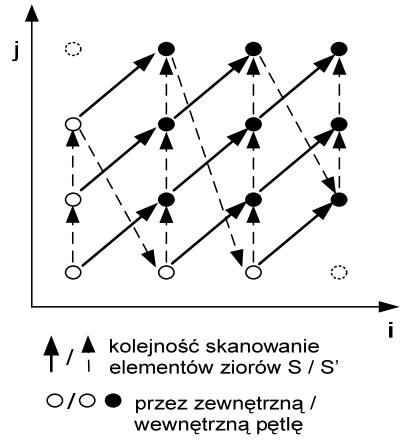
R_1 ({[an element $\in$ SFS]}).

Niemożliwe jest w bezpośredni sposób wygenerowanie kodu przebierającego początki niezależnych fragmentów za pośrednictwem zewnętrznych pętli, natomiast iteracji

należących do poszczególnych fragmentów przy pomocy wewnętrznych pętli. Zbiór reprezentujący iteracje dla wszystkich niezależnych fragmentów kodu przedstawiono poniżej (wygenerowany poprzez bezpośrednie zastosowanie polecenia 'codegen' [59]):

$S = R_1^*(SFS) = \{[i,j]: 2 \le i \le j \le 4\}$ union $\{[i,j]: 2 \le j < i \le 4\}$ union $\{[1,j]: 1 \le j \le 3\}$ union $\{[i,1]: 2 \le i \le 3\}$.

Kod przebierający elementy zbioru S zgodnie z porządkiem leksykograficznym, przebiera iteracje należące do różnych niezależnych fragmentów jednocześnie (rys. 3.8a). Przedstawiony sposób przebierania iteracji nie realizuje przyjętych założeń.




```
for(j = 1; j ≤ 3; j++)
  s1(1,j);
for(i = 2; i ≤ 4; i++) {
  if (i ≤ 3)
    s1(i,1);
  for(j = 2; j ≤ i-1; j++)
    s1(i,j);
  for(j = i; j ≤ 4; j++)
    s1(i,j);
}
```

```
par for(j = 1; j ≤ 3; j++) {
  for(i' = 1; i' ≤ -j+5; i'++)
    s1(i', i'+j-1);
}
par for(i = 2; i ≤ 3; i++) {
  for(i' = i; i' ≤ 4; i'++)
    s1(i', i'-i+1);
  }
```

Rysunek 3.8a Kolejność przebierania elementów zbiorów S i S' [op. własne]

Iteracje zbioru S

Iteracje zbioru S'

Rysunek 3.8b Kod przebierający niezależne fragmenty [op. własne]

W celu wygenerowania poprawnego kodu, niezbędne jest zastosowanie schematu:

Kod przebierający wektory iteracji należące do zbioru SFS

Kod przebierający elementy zbioru R_1^* ({[wektor iteracji elementu z SFS]}).

Aby zaimplementować powyższy schemat, wykorzystano rozwiązanie pozwalające na zastosowanie znanych metod generacji kodu. W tym celu należy stworzyć nowy zbiór S' będący unią wszystkich produktów wektorowych postaci:

$\{[e_i, R^*(e_i)]\}$, i=1,2,....,n, gdzie

n – określa liczbę niezależnych fragmentów kodu,

e_i – reprezentuje początek danego fragmentu kodu (i'ty element zbioru SFS),

$R^*(e_i)$ – zbiór iteracji należących do niezależnego fragmentu kodu o początku w iteracji e_i

Dla relacji R_1 uzyskano następujący zbiór S':

S':= {[1,j,i',i'+j-1]: 2 ≤ i' ≤ -j+5 && 1 ≤ j} union {[i,1,i',i'-i+1]: 2 ≤ i < i' ≤ 4} union {[1,j,1,j]: 1 ≤ j ≤ 3} union {[i,1,i,1]: 2 ≤ i ≤ 3}.

Kod przebierający elementy zbioru S' przedstawiony został na rys. 3.8b, gdzie współrzędne reprezentujące początki niezależnych fragmentów kodu zostały usunięte z wyrażeń pętli.

Poniżej przedstawiony został algorytm wyznaczający niezależne fragmenty kodu. Dane wejściowe algorytmu stanowi wyjście jednego ze wcześniejszych algorytmów.

Algorytm 3.5. Wyznaczanie iteracji należących do niezależnych fragmentów oraz generacja kodu

Wejście: wyjście z algorytmu 3.3 (zbiór początków dla niezależnych fragmentów kodu, SFS oraz zbiór S_In zawierający relacje zależności), lub wyjście z algorytmu 3.4 (współrzędne pętli definiujące podprzestrzenie gdzie wyznaczane są początki dla niezależnych fragmentów kodu; zbiór SFS zawierający początki dla niezależnych fragmentów kodu opisanych przez relacje w zbiorze S_Out oraz sam zbiór S_Out)

Wyjście: kod przebierający niezależne fragmenty oraz iteracje należące do poszczególnych fragmentów

1. Jeśli wejściem algorytmu jest wyjście z algorytmu 3.4 to:
   1.1. Dla każdej relacji ze zbioru S_Out usuń ograniczenia nałożone na zmienne indeksowe wyznaczone w algorytmie 3.4, zadeklaruje te zmienne jako symboliczne.
   1.2. Stwórz gniazdo pętli przebierającej sekwencyjnie wartości zmiennych indeksowych pętli wyznaczony w algorytmie 3.4 poprzez przepisanie i modyfikacje odpowiedniego gniazda z oryginalnej pętli.
2. Utwórz relację R jako unię wszystkich relacji ze zbioru S_In (jeśli wejściem algorytmu jest wyjście algorytmu 3.3) lub ze zbioru S_Out (jeśli wejściem algorytmu jest wyjscie algorytmu 3.4), pod warunkiem że $R_i(SFS) \neq EMPTY$, gdzie R_i pochodzi ze zbioru S_In lub S_Out, $i \in [1,m]$.

3. Utwórz relację S na podstawie kroków 3a lub 3b

3a. przy użyciu tranzytywnego domknięcia	3b. bez użycia tranzytywnego domknięcia (dla niesparametryzowanego zbioru SFS i relacji R)
3.1. Oblicz tranzytywne domknięcie dla relacji R, $R^*:=R+ \cup I$. 3.2. Stwórz zbiór S następującej postaci: $S := \{[e_i , R^*(e_i)]: \forall e_i \in SFS \}$	3.1. Stwórz zbiór SFSD jak poniżej: $SFSD := \{ [e_i, e_i] : \forall e_i \in SFS \}$, 3.2. Stwórz relacje RD w następujący sposób: $RD:=\{[e_i,In] \rightarrow [e_i,R(In)]:\forall e_i \in SFS\}$, gdzie In jest krotką zmiennych o wymiarze równym krotce reprezentującej e_i. 3.3. S := SFSD, Temp := SFSD, 3.4. Temp := RD(Temp) – S, 3.5. Jeśli Temp ≠ EMPTY to $S := S \cup Temp$, przejdź do 3.4, w przeciwnym razie zakończ algorytm

4. Wygeneruj kod przebierający elementy zbioru S zgodnie z porządkiem leksykograficznym przy użyciu jednej ze znanych technik, np. [3], [21], [25], [82].
5. Usuń z instrukcji pętli, wygenerowanej we wcześniejszym kroku, n/2 pierwszych zmiennych (zmienne krotek reprezentujące początki niezależnych fragmentów kodu).
6. Jeśli w kroku 1.2 wygenerowane zostało gniazdo pętli to w gniazdo pętli wygenerowanej w kroku 1.2 wstaw pętle przebierające niezależne fragmenty kodu oraz iteracje należące do nich, wygenerowane w krokach 3-5.

Pierwsze n/2 zmiennych ze zbioru S, stworzonego w kroku 3a lub 3b (gdzie n to liczba wszystkich zmiennych zbioru S), reprezentuje początki kodu pozbawione synchronizacji. Pozostałe zmienne zbioru S odpowiedzialne są za skanowanie iteracji należących do niezależnych fragmentów kodu. Kod wygenerowany na podstawie zbioru S (wynik kroku 4) przebiera iteracje należące do fragmentów kodu zgodnie z porządkiem leksykograficznym, rozpoczynając od leksykograficznie minimalnej iteracji. Jednakże wygenerowany kod zawiera nadmiarowe zmienne indeksowe. Po wykonaniu kroku 5 nadmiarowe zmienne zostają usunięte z wyrażeń pętli. Pozwoli to na zredukowanie nadmiarowych zmiennych, potrzebnych podczas generacji kodu do zachowania porządku leksykograficzne w obrębie pojedynczego fragmentu.

Krok 3b powyższego algorytmu może zostać zastosowany dla pętli sparametryzowanych jedynie w trakcie wykonania pętli, gdy górne granice pętli stają się znane. W kroku 3a wykorzystano operację tranzytywnego domknięcia. Jak już

wcześniej zostało wspomniane, reprezentacja dokładnego tranzytywnego domknięcia w ogólnym przypadku dla relacji opisujących zależności niejednorodne, niemożliwa jest do zapisania przy pomocy wyrażeń afinicznych. Jeśli dla danej relacji, niemożliwe jest obliczenie dokładnej reprezentacji tranzytywnego domknięcia przy pomocy narzędzia Omega Calculator, możliwe jest obliczenie przybliżenia dla tranzytywnego domknięcia (dolne lub górne przybliżenie) pozwala to na wyznaczenia niezależnych fragmentów kodu.

Poniżej przedstawiono działanie algorytmu 3.5 dla przykładowej pętli.

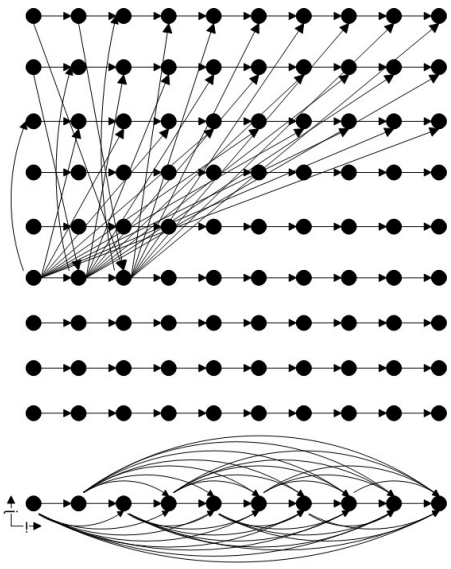
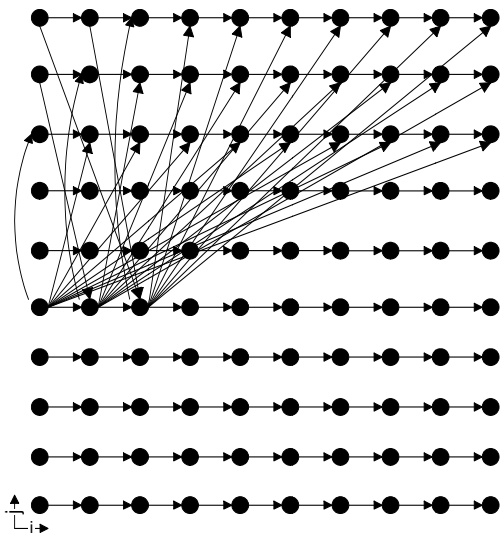
for(i = 1; i ≤ n; i++)
  for(j = 1; j ≤ n; j++)
  a(j+4,j+1) = a(i+2*j+1,i+j+3)
	$R_1 := \{[i,5] \rightarrow [i,i+7] : 1 \leq i \leq n-7\}$,
	$R_2 := \{[i,5] \rightarrow [i',i+7] : 1 \leq i < i' \leq n \;\&\&\; i \leq n-7\}$,
	$R_3 := \{[i,i+8] \rightarrow [i+1,5] : 1 \leq i \leq n-8\}$,
	$R_4 := \{[i,j] \rightarrow [j-7,5] : 1 \leq i \leq j-8 \;\&\&\; j \leq n\}$,
	$R_5 := \{[i,j] \rightarrow [i',j] : 1 \leq i < i' \leq n \;\&\&\; 1 \leq j \leq n\}$.

Przykład 3.4 Pętla z pięcioma zależnościami	Relacje zależności dla przykładu 3.4

Po zredukowaniu nadmiarowych tranzytywnych zależności reprezentowanych przez relację R_5, uzyskujemy relację R_5' następującej postaci:

$R_5' := \{[i,j] \rightarrow [i+1,j] : 1 \leq i < n \;\&\&\; 1 \leq j \leq n\}$.

Rysunki 3.9a i 3.9b przestawia zależności dla powyższego przykładu przed i po redukcji nadmiarowych zależności (na rys. 3.9a, zależności opisane przez relacje R_5, w celu zachowania czytelności rysunku, pokazane są jedynie dla j=1).

	
Rysunek 3.9a Zależności opisane przez relacje R_1, R_2, R_3, R_4, R_5, n=10 [op. własne]	Rysunek 3.9b Zależności opisane przez relacje R_1, R_2, R_3, R_4, R_5', n=10 [op. własne]

Zgodnie z algorytmem 3.3 otrzymujemy:

1. R := R_1 union R_2 union R_3 union R_4 union R_5' = {[i,5] → [i,i+7] : 1 ≤ i ≤ n-7} union {[i,5] → [i',i+7] : 1 ≤ i < i' ≤ n && i ≤ n-7} union {[i,j] → [j-7,5] : 1 ≤ i ≤ j-8 && j ≤ n} union {[i,j] → [i+1,j] : 1 ≤ i < n && 1 ≤ j ≤ n}

   IR := inverse R := {[i,i+7] → [i,5] : 1 ≤ i ≤ n-7} union {[i,j] → [j-7,5] : 8 ≤ j ≤ i+6, n && i ≤ n} union {[i,5] → [i',i+7] : 1 ≤ i' < i ≤ n-7} union {[i,j] → [i-1,j] : 2 ≤ i ≤ n && 1 ≤ j ≤ n},

   CI:=EMPTY.
2. CI := {[i,i+7]: 1 ≤ i ≤ n-7} union {[i,5]: 1 ≤ i ≤ n-7} union {[i,j]: 8 ≤ j ≤ i+6, n && i ≤ n} union {[i,j]: 1 ≤ i ≤ j-8 && j ≤ n}.
3. UDS := domain R - range R := {[1,j]: 1≤ j ≤ 7, n && 2 ≤ n} union {[1,j]: 9 ≤ j ≤ n}.
4. CI nie jest zbiorem pustym, więc przechodzimy do punktu 5a (z użyciem tranzytywnego domknięcia).
5. Nie możliwe jest wyznaczenie dokładnego tranzytywnego domknięcia przy pomocy Omega Calculatora dla relacji IR. Przy użyciu górnego przybliżenia (UNKNOWN=True) dla tranzytywnego domknięcia ze zbioru IR, uzyskujemy następujący zbiór

   SFS := {[1,j]: j, 2 ≤ n ≤ 7 && 1 ≤ j} union {[1,j]: n = 8 && 6 ≤ j ≤ 7} union {[1,j]: n = 8 && 1 ≤ j ≤ 4}

   Zbiór ten nie zawiera wszystkich początków dla niezależnych fragmentów kodu, pewien zbiór początków został pominięty.

   Stosując dolne przybliżenie dla relacji TIR (UNKNOWN=False) możliwe jest wyznaczenie wszystkich początków dla niezależnych fragmentów kodu (poniższy zbiór SFS):

   SFS := {[1,j]: j, 2 ≤ n ≤ 7 && 1 ≤ j} union {[1,j]: 6 ≤ j ≤ 7 && 8 ≤ n} union {[1,j]: 1 ≤ j ≤ 4 && 8 ≤ n}.

Stosując algorytm 3.5 z wykorzystaniem kroku 3a, uzyskano kod przebierający niezależne fragmenty wraz z iteracjami należącymi do nich.

```
if (n ≥ 2 && n ≤ 7) {
  par for(t2 = 1; t2 ≤ n; t2++) {
    s1(1,t2);
    seq for(t3 =2; t3 ≤ n; t3++)
      s1(t3,t2);
  }
}
```

```
if (n ≥ 8) {
  par for(t2 = 1; t2 ≤ 4; t2++) {
    s1(1,t2);
    seq for(t3 = 2; t3 ≤ n; t3++)
      s1(t3,t2);
  }
}
```

```
if (n ≥ 8) {
  par for(t2 = 6; t2 ≤ 7; t2++) {
    s1(1,t2);
    seq for(t3 =2; t3 ≤ n; t3++)
      s1(t3,t2);
  }
}
```

Stosując algorytm 3.5 dla wyjścia z algorytmu 3.4 dla przykładu 3.4 uzyskano następujący kod:

```
if (n ≥ 2)
    seqfor(k = 1; k ≤ n; k++)
        parfor(i = 1; i ≤ n; i++)
            seqfor(j = 1; j ≤ n; j++)
                a(i,j,k)=a(i,j-1,k)
                b(i,j,k)=b(i,j,k-1).
```

Pętla przebiera n niezależnych fragmentów zgodnie z przyjmowanymi wartościami zmiennej indeksowej i oraz j w każdej podprzestrzeni zdefiniowanej przez zmienną indeksowa k. Podprzestrzenie przebierane są w sposób sekwencyjny przez najbardziej zewnętrzną pętlę.

3.5. Pętle dowolnie zagnieżdżone

W poniższym rozdziale zaproponowano rozwiązanie umożliwiające wyszukiwanie niezależnych fragmentów dla pętli dowolnie zagnieżdżonych. Przez pętlę dowolnie zagnieżdżoną rozumie się pętle, dla której zmienne indeksowe, górne i dolne granice pętli, wyrażenia indeksujące zmienne tablicowe oraz wyrażenia warunkowe są funkcjami afinicznymi wyrażonymi przez zmienne indeksowe pętli. Krok pętli jest dodatnią stałą znaną w czasie kompilacji pętli.

3.5.1. Wyznaczanie początków dla niezależnych fragmentów kodu

Zaproponowany poniżej algorytm pozwala na wyznaczenie początków niezależnych fragmentów kodu w zadanym grafie CRDG w następujących przypadkach:

A) Część kodu tworzy łańcuch bądź drzewo zależnych instancji iteracji dwóch lub więcej instrukcji nieuwikłanych w żaden cykl w grafie CRDG (patrz rys. 3.10a). Przykładem ciała pętli generującej powyższy przykład są następujące instrukcje:

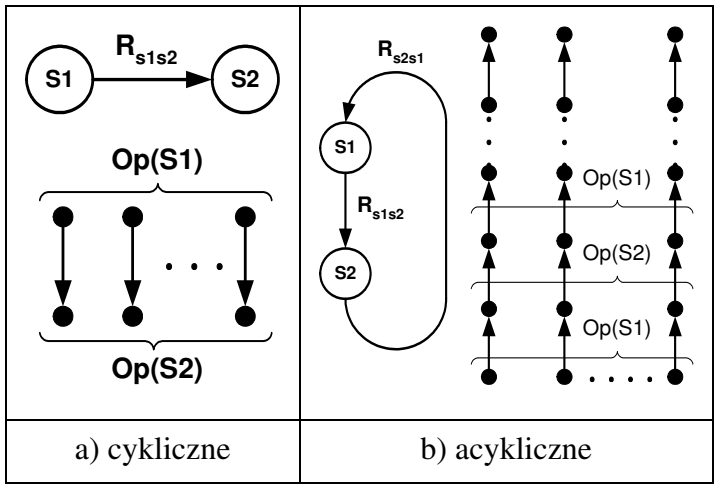
   a(i,j) = b(i,j)

   c(i,j) = a(i,j)

B) Część kodu tworzy łańcuch bądź drzewo zależnych instancji iteracji dwóch lub więcej wyrażeń uwikłanych w graf cykliczny zawarty w grafie CRDG (patrz rys. 3.10b). Przykładem cyklicznego grafu zawierającego niezależne fragmenty kodu jest ciało poniższej pętli:

a(i,j) = a(i,j) + b(i-1,j)

b(i,j) = a(i,j-1) + b(i,j)



a) cykliczne | b) acykliczne

Rysunek 3.10. Rodzaje zależności w grafie CRDG [op. własne]

Algorytm 3.6. Wyszukiwanie początków niezależnych fragmentów w grafie zredukowanych relacji (CRDG)

Wejście: zbiór $S = \{R^1_{i_1 j_1}, R^2_{i_2 j_2}, ..., R^n_{i_n j_n}\}$ zawierający relacje opisujące graf CRDG, gdzie $R^k_{i_k j_k}$, k=1,2,...,n, oznacza k'tą relację w zbiorze S opisującą zależności pomiędzy operacjami dla wyrażeń s_{i_k} oraz s_{j_k}, $i_k, j_k \in [1, q]$, n≥0 jest liczbą relacji zależności w zbiorze S.

Wyjście: zbiór S' zawierający relacje zależności możliwe do użycia w celu generacji niezależnych fragmentów kodu; zbiór $SSF(s_{i_k}, s_{j_k})$ składający się z początków niezależnych fragmentów zbudowanych z instancji iteracji wyrażeń s_{i_k} reprezentowanych przez relację $R^k_{i_k j_k}$; wartości zmiennych "slices" (1 jeśli wyznaczono początki dla fragmentów, w przeciwnym razie 0) oraz zmienna "par" (1 jeśli wszystkie operacje pętli są niezależne, 0 jeśli kod jest sekwencyjny).

1. slice:=0, par:=0. Jeśli zbiór S jest pusty (graf zależności CRDG jest grafem acyklicznym reprezentowanym przez pojedynczy wierzchołek), to wszystkie operacje grafu mogą zostać wykonane równolegle (par:=1), koniec algorytmu
2. Stwórz kopię zbioru S (S' := S). Jeśli zbiór S' zawiera dwie lub więcej różnych relacji opisujących zależności między tymi samymi indeksami oraz część wspólna zakresów dla tych relacji jest zbiorem pustym (oznacza to że relacje opisują różne pary zależności o różnych końcach), to należy usunąć wszystkie takie relacje ze

zbioru S', a w miejsce ich dodać nową relację będącą unią wszystkich usuwanych relacji.

S':= S

L: **DLA** t1 = 1,n-1

DLA t2 = t1+1,n

JEŻELI dla $R^{t1}_{i_{t1}j_{t1}}$ i $R^{t2}_{i_{t2}j_{t2}}$ ze zbioru S' ($i_{t1} == i_{t2}$ i $j_{t1} == j_{t2}$) **TO**

T := range ($R^{t1}_{i_{t1}j_{t1}}$) $\cap$ range ($R^{t2}_{i_{t2}j_{t2}}$)

JEŻELI T==EMPTY **TO**

S':= S' – $R^{t1}_{i_{t1}j_{t1}}$ – $R^{t2}_{i_{t2}j_{t2}}$;

$R^{t1}_{i_{t1}j_{t1}} := R^{t1}_{i_{t1}j_{t1}} \cup R^{t2}_{i_{t2}j_{t2}}$;

S':= S' $\cup$ $R^{t1}_{i_{t1}j_{t1}}$;

n = n – 1;

IDŹ DO L .

3. Jeśli istnieje więcej niż jedna relacja opisująca końce relacji reprezentowane przez instancje iteracji tego samego wyrażenia pętli, to zakończ działanie algorytmu. Nie można wyznaczyć początków dla niezależnych fragmentów.

DLA t1=1,p-1

DLA t2=t1+1,p

JEŻELI dla $R^{t1}_{i_{t1}j_{t1}}$ i $R^{t2}_{i_{t2}j_{t2}}$ ze zbioru S' ($j_{t1} == j_{t2}$) TO

zakończ algorytm

4. Dla każdej relacji $R^{k}_{i_k j_k}$, k=1,2,....,p, stwórz zbiór SSF(s_{i_k}, s_{j_k}), zawierający wszystkie początki fragmentów opisanych przez relację $R^{k}_{i_k j_k}$:

DLA k=1, p

J(s_{i_k}):= FALSE; /* i_k jest indeksem relacji $R^{k}_{i_k j_k}$ */

DLA k=1, p

I(s_{i_k}, s_{j_k}):= domain $R^{k}_{i_k j_k}$;

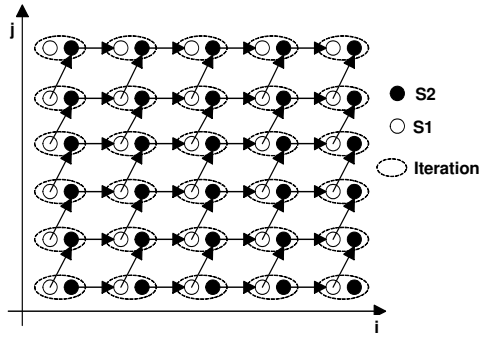
J(s_{j_k}):= range $R^{k}_{i_k j_k}$;

DLA m=1, p

SSF(s_{i_k}, s_{j_k}) := I(s_{i_k}, s_{j_k}) – J(s_{i_k}) .

Przebieg algorytmu 3.6 przedstawiono na przykładzie 3.5 dla pętli zapożyczonej z publikacji [70].

```
for i=1 to n
    for j=1 to m
        s1: a(i,j)=a(i,j)+b(i-1,j)
        s2: b(i,j)=a(i,j-1)+b(i,j)
```



Przykład 3.5 Pętla z dwoma zależnościami

Rysunek 3.11. Niezależne fragmenty dla pętli z przykładu 3.5, gdzie n=5 i m=6 [op. własne]

Na rys. 3.11 przedstawiono niezależne fragmenty kodu, dostępne dla powyższej pętli. Strzałkami oznaczono zależności między instancjami instrukcji pętli. Zależności opisane są przez relacje należące do zbioru S (zbiór S stanowi dane wejściowe dla algorytmu 3.6):

Wejście

$R_{s1s2} = \{[i,j] \rightarrow [i,j+1] : 1 \leq i \leq n \;\&\&\; 1 \leq j < m\}$;

$R_{s2s1} := \{[i,j] \rightarrow [i+1,j] : 1 \leq i < n \;\&\&\; 1 \leq j \leq m\}$.

1. Zbiór S jest zbiorem niepustym, w związku z tym przechodzimy do kolejnego kroku.
2. Zbiór S' := S, relacje R_{s1s2} i R_{s2s1} nie zawierają wspólnych końców dlatego nie dokonujemy złączenia żadnych relacji, zbiór S' pozostaje niezmieniony.
3. W zbiorze S' nie istnieją dwie relacje zależności opisujące końce relacji reprezentowane przez instancje iteracji dla tego samego wyrażenia, w związku z tym przechodzimy do kolejnego kroku.
4. Dla każdej relacji należącej do zbioru S' tworzymy zbiór $SSF(s_{i_k}, s_{j_k})$ zawierający początki dla niezależnych fragmentów:

   4.1. relacja R_{s1s2}

   $I(s_1,s_2) = \text{domain } R_{s1s2} = \{[i,j]: 1 \leq i \leq n \;\&\&\; 1 \leq j < m\}$,

   $J(s_1) = \text{range } R_{s2s1} = \{[i,j]: 2 \leq i \leq n \;\&\&\; 1 \leq j \leq m\}$,

   $SSF(s_1,s_2) = I(s_1,s_2) - J(s_1) = \{[1,j]: 1 \leq j < m \;\&\&\; 1 \leq n\}$

   4.2. relacja R_{s2s1}

   $I(s_2,s_1) = \text{domain } R_{s2s1} = \{[i,j]: 1 \leq i < n \;\&\&\; 1 \leq j \leq m\}$,

   $J(s_2) = \text{range } R_{s1s2} = \{[i,j]: 1 \leq i \leq n \;\&\&\; 2 \leq j \leq m\}$,

   $SSF(s_2,s_1) = I(s_2,s_1) - J(s_2) = \{[i,1]: 1 \leq i < n \;\&\&\; 1 \leq m\}$.

Wyjście:

$S' = \{R_{s1s2}, R_{s2s1}\}$

$SSF(s_1,s_2) = \{[1,j]: 1 \leq j < m \;\&\&\; 1 \leq n\}$

$SSF(s_2,s_1) = \{[i,1]: 1 \leq i < n \;\&\&\; 1 \leq m\}$.

Wyjście algorytmu 3.6 dla przykładowej pętli stanowią dwa zbiory początków niezależnych fragmentów. Zbiór $SSF(s_1,s_2)$ opisuje początki dla niezależnych fragmentów kodu rozpoczynające się w instrukcji s_1. Zgodnie z ograniczeniami zbioru $SSF(s_1,s_2)$ początki umieszczone są wzdłuż osi j dla i = 1 (rys. 3.11). Niezależne fragmenty kodu mające swój początek w instancjach instrukcji s_2 reprezentowane są przez zbiór $SSF(s_2,s_1)$. Zgodnie z ograniczeniami zbioru $SSF(s_2,s_1)$ ułożone są wzdłuż osi i dla j = 1 (rys. 3.11). Ograniczając przestrzeń iteracji pętli z przykładu 3.4 zgodnie z rys. 3.11 (n=5 i m=6), wykonanie algorytmu 3.6 pozwala na wyznaczenie 11 początków dla niezależnych fragmentów, gdzie sześć z nich to początki w instancjach instrukcji s_1 pozostałe pięć to początki w instancjach instrukcji s_1.

3.5.2. Wyznaczanie operacji należących do niezależnych fragmentów oraz generacja kodu

Na użytek algorytmu wyznaczającego niezależne fragmenty kodu należy przyjąć następujące założenia:

i. Zbiór S reprezentowany jest w następujący sposób:

   S := { [ListaKrotek] : ograniczenia(S) },

   gdzie [ListaKrotek] oznacza listę zmiennych opisujących zbiór, natomiast zapis ograniczenia(S) oznacza zbiór ograniczeń nałożonych na zmienne opisujące krotki.

ii. Użycie zapisu Z := {[X, Y]} oznacza że lista krotek zbioru Z składa się z krotek zbioru X oraz Y, np. mając dwa zbiory X i Y następującej postaci:

   X: = {[i,j] : 1<i,j<n}, Y: = {[k,l] : 1<k,j<n},

   to zbiór Z = {X, Y} zawiera następujące krotki:

   Z: = {X, Y} = {[i,j,k,l] : 1<i,j,k,l<n}

iii. Relacja R reprezentowana jest w następujący sposób R := {[In(R)]→ [Out(R)]: ograniczenia(R)}, gdzie In(R) i Out(R) opisuje odpowiednio listę krotek wejściowych i wyjściowych relacji. Zapis ‘ograniczenia(R)’ oznacza zbiór

ograniczeń nałożonych na zmienne opisujące krotki wejściowe i wyjściowe relacji.

Idea zaproponowanego algorytmu przedstawia się następująco. Dla każdej instrukcji reprezentowanej przez wierzchołek w grafie SCC/CRDG tworzony jest zbiór S, którego krotki złożone są z dwóch grup zmiennych S := {X, Y}. Pierwsza grupa zmiennych tworzy krotki X reprezentujące początki niezależnych fragmentów kodu, zgodnie z wyjściem algorytmu 3.6. W związku z tym zmienne należące do krotki X tworzą zbiór tożsamy ze zbiorem SSF. Zbiór zmiennych tworzących krotki należące do grupy zmiennych Y reprezentuje instancje instrukcji należące do fragmentu o początku w krotkach X. Dzięki takiemu podejściu uzyskujemy zbiór S, którego wymiar (liczba zmiennych indeksowych) jest dwa razy większy od liczby zmiennych indeksowych pętli. Pierwsza część zbioru (X) zapewnia, iż instancje instrukcji reprezentujące niezależne fragmenty (Y) będą przebierane zgodnie z porządkiem leksykograficznym.

Jeśli wszystkie elementy krotek reprezentowane są w postaci ograniczeń liniowych lub afinicznych to możliwe jest wygenerowanie kodu przebierającego niezależne fragmenty kodu. Zewnętrzne pętle (zmienne należące do krotek X) przebierają początki dla niezależnych fragmentów, natomiast wewnętrzne pętle (zmienne należące do krotek Y) przebierają instancje instrukcji należące do niezależnego fragmentu kodu z zachowaniem porządku leksykograficznego. Do generacji kodu możliwe jest zastosowanie jednej z następujących technik [3], [8], [21], [25]. Ze względu na to, iż wielkość krotek zbiorów reprezentujących instancje iteracji jest dwa razy większa od wymiaru analizowanej pętli (pierwsza połowa zmiennych indeksowych odpowiedzialna jest za wybieranie niezależnych fragmentów, natomiast druga odpowiada za przebieranie instancji instrukcji należących do pojedynczego fragmentu), z wygenerowanego kodu należy usunąć pierwszą grupę zmiennych.

Algorytm 3.7. Wyznaczanie oraz generowanie kodu dla niezależnych fragmentów dla grafu CRDG

Wejście: zbiór S' oraz zbiory SSF(s_i, s_j), i,j $\in$ [1,q], jako wyjście z algorytmu 3.6; graf CRDG zbudowany na podstawie relacji zawartych w zbiorze S'.

Wyjście: kod przebierający niezależne fragmenty oraz operacje w każdym z nich zgodnie z porządkiem leksykograficznym.

1. Dla każdego zbioru SSF(s_i, s_j), i,j $\in$ [1,q], zbuduj relację R(s_i, s_k), i,k $\in$ [1,q], dla której zakres reprezentowany jest przez końce zależności utworzonych przez

instancje iteracji wyrażenia s_k i należące do fragmentu o początkach w zbiorze $SSF(s_i, s_j)$.

Dla każdego s_k gdzie istnieje ścieżka z s_i do s_k w grafie CRDG wykonaj:

1.1. Utwórz $R(s_i, s_k)$ jako kompozycje wszystkich relacji tworzących ścieżkę w grafie CRDG z instrukcji s_i to s_k. Po zastosowaniu algorytmu 3.6, istnieje tylko jedna ścieżka z s_i do s_k z maksymalnie jednym cyklem z grafie. Relacja $R(s_i, s_i)$ jest również tworzona jeśli s_i wplątany jest w cykl.

1.2. Stwórz zbiór $S(s_i, s_i)$ w następujący sposób:

Jeśli s_i zawiera się w grafie cyklicznym to:

$S(s_i, s_i)$:= { [In(PTC^2 $R(s_i, s_i)$), Out(PTC $R(s_i, s_i)$)]: ograniczenia(PTC $R(s_i, s_i)$) && ograniczenia $SSF(s_i, s_j)$ } union {$SSF(s_i, s_j)$, $SSF(s_i, s_j)$},

w przeciwnym razie:

$S(s_i, s_i)$:= { $SSF(s_i, s_j)$, $SSF(s_i, s_j)$ };

1.3. Stwórz zbiór $S(s_i, s_k)$ w następujący sposób:

Dla każdego $k \neq i$ dla którego istnieje ścieżka z s_i do s_k w grafie CRDG stwórz:

$S(s_i, s_k) := R'(s_i, s_k)\ (S(s_i, s_i))$,

gdzie pierwsza połowa zmiennych wejściowych i wyjściowych relacji $R'(s_i, s_k)$ reprezentowana jest przez wspólne zmienne, natomiast druga połowa zmiennych stanowi wejście i wyjście zmiennych relacji $R(s_i, s_k)$.

2. Dla każdego zbioru $SSF(s_i, s_j)$, $i,j \in [1,q]$, wykonaj:

Jeśli wyrażenie s_i zawarte jest w cyklicznym grafie w CRDG, to sprawdź czy PTC dla relacji $R(s_i, s_i)$ może być reprezentowane przy pomocy ograniczeń afinicznych. Jeśli tak to wygeneruj kod na podstawie zbiorów $S(s_i, s_i)$ i $S(s_i, s_k)$ wykorzystując jedna ze znanych technik generacji kodu, np. jedna z opisanych w [3], [8], [21], [25], [59]. W przeciwnym razie zakończ algorytm.

3. Z wygenerowanego kodu usuń pierwszą część zmiennych odpowiedzialnych za reprezentację początków dla niezależnych fragmentów w zbiorach $S(s_i,s_i)$ i $S(s_i,s_j)$.

[2] PTC – (ang. Positive Transitive Closure) Pozytywne tranzytywne domknięcie

Poniżej przedstawiono działanie algorytmu 3.7, jako kontynuację przykładu z rozdziału 3.5.1

Wejście:

Wyjście z algorytmu 3.6, zbiory:

$S' = \{R_{s1s2}, R_{s2s1}\}$

$SSF(s_1,s_2) = \{[1,j]: 1 \leq j < m \;\&\&\; 1 \leq n\}$

$SSF(s_2,s_1) = \{[i,1]: 1 \leq i < n \;\&\&\; 1 \leq m\}$.

1. Dla każdego zbioru $SSF(s_i, s_j)$, tworzymy relację $R(s_i, s_k)$ w następujący sposób

   1.1. Tworzymy kompozycję relacji tworzących ścieżki w grafie CRDG:

   $$R(s_1,s_1)=R_{s2s1} \bullet R_{s1s2}=\{[i,j]\rightarrow[i+1,j+1]:1 \leq i < n \;\&\&\; 1 \leq j < m\},$$

   $$R(s_2,s_2)=R_{s1s2} \bullet R_{s2s1}=\{[i,j]\rightarrow[i+1,j+1]:1 \leq i < n \;\&\&\; 1 \leq j < m\}.$$

   1.2. Dla relacji stworzonych w kroku 1.1 obliczamy Dodatnie Tranzytywne Domknięcie (DTD):

   $$DTD\ R(s_1,s_1)= \{[i,j] \rightarrow [i',j-i+ i'] : 1 \leq i < i' \leq n \;\&\&\; 1 \leq j \;\&\&\; j+ i' \leq m+i\},$$

   $$DTD\ R(s_2,s_2)= \{[i,j] \rightarrow [i',j-i+i'] : 1 \leq i < i' \leq n \;\&\&\; 1 \leq j \;\&\&\; j+i' \leq m+i\},$$

   następnie tworzymy zbiory $S(s_i, s_i)$:

   $$S(s_1,s_1)= \{[1,j,i',i'+j-1]: 2 \leq i' \leq n \;\&\&\; i' + j \leq 1 + m \;\&\&\; 1 \leq j\} \text{ union } \{[1,j,1,j]: 1 \leq j < m \;\&\&\; 1 \leq n\},$$

   $$S(s_2,s_2)= \{[i,1,i',i'-i+1]: 1 \leq i < i' \leq n \;\&\&\; i' < m + i\} \text{ union } \{[i,1,i,1]: 1 \leq i < n \;\&\&\; 1 \leq m\}.$$

   1.3. Tworzymy zbiory $S(s_i, s_k)$:

   $$S(s_1,s_2)= \{[1,j1,i,i+j1]: 2 \leq i \leq n \;\&\&\; 1 \leq j1 \;\&\&\; i+j1 \leq m\} \text{ union } \{[1,j1,1,j1+1]: 1 \leq j1 < m \;\&\&\; 1 \leq n\},$$

   $$S(s_2,s_1)= \{[i1,1,i2,i2-i1]: 1 \leq i1 \leq i2-2 \;\&\&\; i2 \leq n \;\&\&\; i2 \leq m + i1\} \text{ union } \{[i1,1,i1+1,1]: 1 \leq i1 < n \;\&\&\; 1 \leq m\}.$$

Krok 2,3. Na podstawie zbiorów wygenerowanych w kroku 1.2 oraz 1.3, przy użyciu funkcji „codegen” z narzędzia Omega Calculator, generujemy kod przebierający niezależne fragmenty kodu:

if (n≥1)
  for(i=1;i≤m-1;i++) {
  s1(1,i);
  s2(1,i+1);
  for(j=2;j≤min(m-i,n);j++){
  s1(j,j+i-1);
  s2(j,j+i);
  }.
  if (m≤n+i-1)
  s1(m-i+1,m);
  }
	if (m ≥ 1)
	  for(i=1;i≤n-1;i++) {
	  s2(i,1);
	  s1(i+1,1);
	  if (m≥2)
	  s2(i+1,2);
	  for(j=i+2;j≤min(m+i-1,n);j++) {
	  s1(j,j-i);
	  s2(j,j-i+1);
	  }
	  if (m ≥ 2 && n ≥ m+i)
	  s1(m+i,m); }

Kod reprezentujący niezależne fragmenty o początkach w instancjach instrukcji s_1.	Kod przebierający niezależne fragmenty o początkach w instancjach instrukcji s_2.

3.5.3. Wyznaczanie niezależnych fragmentów kodu dla grafu zależności o dowolnej strukturze

Dla wielu pętli zredukowany graf zależności (RDG) nie spełnia warunków nałożonych w podrozdziale 3.5.1 (rys. 3.10). Ze względu na wspólne końce dla różnych zależności niemożliwe jest wyznaczenie zgodnie z zaproponowanymi algorytmami początków dla niezależnych fragmentów kodu. Z przeprowadzonych obserwacji wynika, iż w wielu przypadkach, możliwe jest wyznaczenie początków dla niezależnych fragmentów kodu dla pewnego fragmentu (lub też fragmentów) grafu zależności. Niezależne fragmenty kodu dla pojedynczego podgrafu mogą być wykonane współbieżnie. Jednakże w celu zapewnienia poprawności kodu, poszczególne podgrafy muszą być przebierane zgodnie z kolejnością topograficzną wyznaczoną przez graf, w którym są zawarte.

Poniżej zaproponowano algorytm umożliwiający wyznaczenie niezależnych fragmentów kodu w dowolnym grafie zależności.

Algorytm 3.8. Wyznaczanie niezależnych fragmentów kodu dla pętli o dowolnym grafie zależności

Wejście: zbiór S_{all} zawierający wszystkie relacje zależności wyznaczone podczas analizy zależności dla danej pętli.

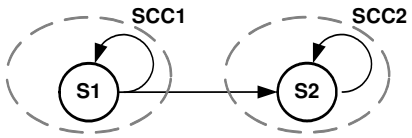
Wyście: kod przebierający grafy SCC/CRDG zgodnie z kolejnością wynikającą z topografii grafu RDG, niezależne fragmenty kodu wyznaczone dla każdego SCC/CRDG.

1. Na podstawie relacji zawartych w zbiorze S_{all} stwórz graf RDG oraz dokonaj podziału powstałego grafu na grafy cykliczne (SCC)
2. Stwórz graf CRDG poprzez złączenie połączonych grafów SCC zawierających pojedynczy wierzchołek z pojedynczą krawędzią wejściową (celem tego kroku jest zmniejszenie granulacji kodu)
3. Dla każdego grafu SCC/CRDG wykonaj:
   3.1. Zastosuj algorytm 3.6
   3.2. W zależności od uzyskanych wartości parametrów wyjściowych z algorytmu 3.6 wykonaj:
      3.2.1. „par" = 1 – wygeneruj niezależny kod
      3.2.2. „slices" = 0 – wygeneruj kod sekwencyjny
      3.2.3. „par" ≠ 1 i „slices" ≠ 0 – na podstawie uzyskanych wyników z kroku 1.1 wykonaj algorytm 3.7, w celu wygenerowanie kodu przebierającego niezależne fragmenty dla danego SCC/CRDG.
4. Wygeneruj kod wynikowy biorąc pod uwagę kolejność grafów SCC/CRDG w grafie RDG oraz kod dla każdego grafu SCC/CRDG na postawie kroku 3.

Przykład zastosowania algorytmu 3.8 przedstawiono dla poniższej pętli. Bezpośrednie zastosowanie algorytmów 3.6 i 3.7 nie pozwala na wyznaczenie niezależnych fragmentów, ze względu na relacje opisujące zależności o wspólnych końcach (patrz rys. 3.12). Po zastosowaniu algorytmu 3.8, możliwe jest wyznaczenie 'n' niezależnych fragmentów kodu dla każdego cyklicznego grafu (SCC1 oraz SCC2).

<table>
<tr><td>

```
for i=1 to n by 1 do
    for j=1 to n by 1 do
        s1: a(i,j)=a(i,j-1)
        s2: b(i,j)= b(i-1,j)+a(i,j)
```

</td><td> 
</td></tr>
<tr><td>Ciało pętli</td><td>Rysunek 3.12. Graf RDG dla pętli po lewej [op. własne]</td></tr>
<tr><td>

```
for i=1 to n by 1 do
    par for j=1 to n by 1 do
        s1: a(i,j)=a(i,j-1)
```

</td><td>

```
par for i=1 to n by 1 do
    for j=1 to n by 1 do
        s2: b(i,j)= b(i-1,j)+a(i,j)
```

</td></tr>
<tr><td>Niezależne fragmenty dla podgrafu SCC1</td><td>Niezależne fragmenty dla podgrafu SCC2</td></tr>
</table>

Zaproponowane rozwiązanie pozwala na zwiększenie zakresu stosowalności przedstawionych algorytmów. W przypadku, gdy dla całego grafu niemożliwe jest wyznaczenie niezależnych fragmentów, podział grafu RDG zgodnie z algorytmem 3.8

w wielu przypadkach pozwala na wyznaczenie niezależnych fragmentów kodu w obrębie pojedynczych podgrafów.

3.5.4. Wyznaczanie niezależnych fragmentów kodu w gnieździe pętli

W przypadku, gdy niemożliwe jest wyznaczenie początków dla niezależnych fragmentów kodu przy zastosowaniu algorytmu 3.6, poniższy algorytm umożliwia wyszukanie początków w wewnętrznym gnieździe pętli. Idea algorytmu polega na wyznaczeniu zbioru wektorów wraz z odpowiadającymi im relacjami zależności, gdzie kolejne koordynaty wektorów zależności traktowane są jako stałe. Liczba stałych wartości w wektorach zależności zależna jest od ilości wykonanych iteracji algorytmu. Wynikowy kod przebiera niezależne fragmenty kodu w sposób równoległy, natomiast gniazda pętli, odpowiadające przyjętym stałym, przebierane są w sposób sekwencyjny.

Algorytm 3.9. Wyszukiwanie niezależnych fragmentów kodu w gnieździe pętli

Wejście: ciało pętli do analizy, zbiór S_1 zawierający zmienne indeksowe $i_1, i_2, ..., i_m$ pętli, gdzie m określa liczbę zmiennych indeksowych.

Wyjście: w przypadku wykrycia niezależnych fragmentów wyjściem jest kod, którego wewnętrzne pętle przebierają niezależne fragmenty

1. $S_1' := S_1$, $S_2 :=$ EMPTY, k:=1
2. Wyznacz zbiór relacji (S) opisujących zależności w ciele pętli opisanej przez zmienne indeksowe zawarte w zbiorze S_1', przy zapewnieniu, że zmienne zawarte z zbiorze S_2 zadeklarowane są jako zmienne symboliczne. Dla wyznaczonego zbioru zastosuj algorytm 3.8.
3.

Wyznaczono początki niezależnych fragmentów	
tak	nie
Stwórz kod przebierający indeksy pętli 1, 2, ..., k-1 jako pętlę zewnętrzną oraz kod przebierający niezależne fragmenty wygenerowane przez algorytm 3.8 jako gniazdo wewnętrzne pętli.	Przenieś indeks i_k ze zbioru S_1' do zbioru S_2. Jeśli zbiór S_1' jest zbiorem pustym to zakończ algorytm (nie wyznaczono niezależnych fragmentów), w przeciwnym razie k = k + 1 i przejdź do kroku 2.

Zastosowanie algorytmu 3.9 przedstawiono na przykładzie 3.5.

for i=1 to n for j=1 to m s1: a(i,j)=a(i,j-2)+a(i-1,j).	$R_{s1s1} = [i,j] \rightarrow [i,j+2] : 1 \le i \le n \;\&\&\; 1 \le j \le m-2\}$, $R'_{s1s1} = \{[i,j] \rightarrow [i+1,j] : 1 \le i < n \;\&\&\; 1 \le j \le m\}$.
Przykład 3.5. Ciało pętli	Wyznaczono następujące relacji zależności

Algorytm 3.6 nie pozwala na wyznaczenie początków dla niezależnych fragmentów kodu. Zgodnie z alg. 3.9 relacje zależności wyznaczone są dla zmodyfikowanej pętli:

```
for j=1 to m
    s1: a(i,j)=a(i,j-2)+b(i-1,j),
```

gdzie zmienna ‘i’ zadeklarowana jest jako stała. Dla powyższej pętli Petit wyznaczył jedną relację zależności, postaci:

$R_{s1s1}=\{[j]\rightarrow[j+2]:1\le j\le m-2\}$.

W takim przypadku algorytm 3.6 pozwala na wyznaczenie dwóch początków (j=1, j=2) dla niezależnych fragmentów. Po wykonaniu kroku 3 algorytmu 3.9 uzyskujemy następujący kod przebierający niezależne fragmenty:

```
for i=1 to n
    par for k=1 to 2
        for j=k to m by 2
            s1: a(i,j)=a(i,j-2)+a(i-1,j).
```

Zewnętrzna pętla odpowiedzialna jest za przebieranie iteracji reprezentowanych przez pętla o zmiennej indeksowej ‘i’ zadeklarowanej w algorytmie 3.9 jako stałej. Kolejna pętla odpowiada za wyznaczanie niezależnych fragmentów kodu. W powyższym przykładzie dla każdej wartości indeksu ‘i’ otrzymujemy dwa niezależne fragmenty kodu. Liczbę iteracji w każdym z niezależnych fragmentów kodu wyznacza ostatnia pętla o zmiennej indeksowej ‘j’.

3.6. Tworzenie kodu reprezentującego drobnoziarnistą równoległość

Zaproponowane algorytmy dla pętli idealnie zagnieżdżonych pozwalają na wyznaczenie kodu o większym ziarnie niż algorytmy dla pętli dowolnie zagnieżdżonych. Pętle idealnie zagnieżdżone analizowane są na poziomie iteracji, w związku z tym, dla iteracji zawierającej więcej niż jedną instrukcję ziarnistość kodu zwiększa się w naturalny sposób. Analiza pętli dowolnie zagnieżdżonych odbywa się na poziomie instrukcji, dlatego też uzyskany kod charakteryzuje się mniejszym ziarnem w stosunku do kodu dla pętli idealnie zagnieżdżonych. Jednakże w obu podejściach większy nacisk położony został na wyznaczenie możliwe największego ziarna, w celu redukcji kosztów związanych z zarządzaniem i komunikacją pomiędzy większą ilością

współbieżnie wykonywanych zadań. Celem algorytmów 3.5 oraz 3.7 jest wyznaczenie kodu przebierającego niezależne fragmenty w taki sposób, aby możliwe było zrównoleglenie najbardziej zewnętrznych pętli, a przebieranie fragmentów odbywało się w wewnętrznym gnieździe pętli. Poniżej przedstawiono przykład obrazujący powyższe podejście.

for i=1 to n by 1 do
for j=1 to n by 1 do
for k=1 to n by 1 do
s1: a(i,j,k) = a(i,j-1,k)
	$R_{s1s1} = \{[i,j,k] \rightarrow [i,j+1,k] : 1 \leq i,j,k \leq n\}$

Przykład 3.6. Ciało pętli	Zależności występujące w pętli

Wynikiem analizy na podstawie algorytmów 3.5 i 3.7 jest następujący kod:

```
par for i=1 to n by 1 do
    par for k=1 to n by 1 do
        seq for j=1 to n by 1 do
            s1: a(i,j,k) = a(i,j-1,k)
```

Z powyższego zapisu wynika, iż możliwe jest zrównoleglenie dwóch pierwszych pętli. Trzecia pętla odpowiedzialna jest za przebieranie sekwencyjne iteracji należących do pojedynczego fragmentu. W związku z powyższym możliwe jest wykonanie kodu na n*n niezależnych wątkach (przy jednoczesnym zrównolegleniu dwóch pierwszych pętli), gdzie w pojedynczym ziarnie zawarte jest n iteracji. W przypadku, gdy n będzie stosunkowo dużą liczbą, pojedynczy fragment kodu (w tym przypadku ziarno) może zawierać dużą liczbę iteracji. Dlatego też, taki podział przestrzeni iteracji pętli może zostać uznany za podział gruboziarnisty. W celu zmiany ziarnistości powyższego kodu na drobne ziarna należy dokonać przekształceń umożliwiających przebieranie w sposób równoległy pętli wewnętrznych. Możliwe jest to do osiągnięcia poprzez zmianę kolejności wykonania pętli, gdzie pętla o indeksie j będzie pętlą najbardziej zewnętrzną, natomiast pętle o indeksach i oraz k staną się pętlami wewnętrznymi. Przed zmianą kolejności pętli konieczne jest sprawdzenie czy taka transformacja może zostać przeprowadzona. Warunki poprawności takiej transformacji przytoczone zostały m.in. w książce Allena R i Kennedy'iego K [1]. Poniżej przedstawiono kod reprezentujący drobnoziarnisty podział dla przykładu 3.6.

```
seq for j=1 to n by 1 do
    par for i=1 to n by 1 do
        par for k=1 to n by 1 do
            s1: a(i,j,k) = a(i,j-1,k)
```

Ziarnistość kodu została w znacznym stopniu zredukowana, gdyż pojedyncze ziarno zawiera teraz tylko jedną instrukcję. Uzyskany w ten sposób kod pozwala na efektywne balansowanie obciążeniem jednostek obliczeniowych.

Algorytmy 3.4, 3.8 oraz 3.9 pozwalają na wyznaczenie niezależnych fragmentów kodu w podprzestrzeniach iteracji pętli. W wyniku analizy, zgodnie z powyższymi algorytmami, n pętli ($n > 0$) tworzących podprzestrzenie wykonywane jest w sposób sekwencyjny, przez co zależności występujące względem tych pętli mogą zostać zignorowane. Niezależne fragmenty wyznaczane są w obrębie pojedynczej podprzestrzeni. Zaproponowany sposób analizy skutkuje zmniejszeniem ziarnistości wynikowego kodu. Poniżej przedstawiono przykład obrazujący powyższe podejście.

for i=1 to n by 1 do
  for j=1 to n by 1 do
  for k=1 to n by 1 do
  s1: a(i,j,k) = b(i,j,k-1)
  s2: b(i,j,k) = a(i,j-1,k)
	$R_{s2s1} = \{[i,j,k] \rightarrow [i,j+1,k] : 1 \leq i,j,k \leq n\}$
	$R_{s1s2} = \{[i,j,k] \rightarrow [i,j,k+1] : 1 \leq i,j,k \leq n\}$

Przykład 3.7. Ciało pętli	Zależności występujące w pętli

Wyznaczenie niezależnych fragmentów kodu w całej podprzestrzeni iteracji pętli jest niemożliwe. Analiza zgodnie z algorytmem 3.4 umożliwiła wyznaczenie podprzestrzeni względem zmiennej indeksowej k. Wynikiem analizy powyższej pętli jest następujący kod:

```
seq for k=1 to n by 1 do
  par for i=1 to n by 1 do
    seq for j=1 to n by 1 do
      s1: a(i,j,k) = b(i,j,k-1)
      s2: b(i,j,k) = a(i,j-1,k)
```

Pierwsza pętla odpowiedzialna jest za sekwencyjne przebieranie podprzestrzeni w obrębie, których wyznaczono n niezależnych fragmentów kodu. Zadaniem pętli o indeksie i jest przebieranie niezależnych fragmentów kodu w sposób umożliwiający ich zrównoleglenie. Ostatnia pętla przebiera iteracji należące do pojedynczego fragmentu kodu w sposób sekwencyjny. Ilość iteracji występujących w pętli o indeksie j wyznacza liczbę iteracji określających wielkość ziarna, którego wielkość równa jest n iteracji.

Analogicznie jak dla przykładu 3.6, tak i dla przykładu 3.7, możliwe jest dodatkowe zmniejszenie ziarnistości kodu, poprzez zamianę kolejności wykonania dwóch ostatnich pętli. Poniżej przedstawiono kod reprezentujący drobnoziarnisty podział dla przykładu 3.7, gdzie wielkość ziarna zredukowana została do pojedynczej iteracji

```
seq for k=1 to n by 1 do
    seq for j=1 to n by 1 do
        par for i=1 to n by 1 do
            s1: a(i,j,k) = b(i,j,k-1)
            s2: b(i,j,k) = a(i,j-1,k)
```

3.7. Podsumowanie

W niniejszym rozdziale zaprezentowano algorytmy pozwalające na wyznaczenie niezależnych fragmentów kodu dla pętli idealnie zagnieżdżonych, jak i pętli dowolnie zagnieżdżonych. Algorytmy 3.1 – 3.5, przedstawione w podrozdziale 3.4, pozwalają na wyznaczenie niezależnych fragmentów kodu na poziomie iteracji dla pętli idealnie zagnieżdżonych. Ze względu na atomowość iteracji (instrukcje w jednej iteracji są niepodzielne) w naturalny sposób zwiększona jest ziarnistość kodu. Z drugiej strony takie podejście ogranicza możliwość wyznaczenia pełnej równoległości, gdyż nie wszystkie instrukcje pętli muszą być uwikłane w zależności. Analiza na poziomie instrukcji została zaproponowana w algorytmach 3.6 – 3.9. Dzięki takiemu podejściu możliwe jest analizowanie pętli dowolnie zagnieżdżonych. Dla pętli idealnych wszystkie instrukcje są na tym samym poziomie zagnieżdżenie, natomiast w pętlach nieidealnie zagnieżdżonych warunek ten może, choć nie musi być spełniony.

W przypadku, gdy niemożliwe jest wyznaczenie niezależnych fragmentów kodu w całej przestrzeni iteracji pętli, zaproponowane rozwiązanie pozwala na zawężenie przestrzeni do określonych wymiarów. Kolejne podprzestrzenie przebierane są w sposób sekwencyjny, natomiast równoległość wyznaczona jest wewnątrz poszczególnych podprzestrzeni, o ile taka istnieje.

W podrozdziale 3.6 pokazano, sposób jak poprzez zamianę kolejności wykonania pętli zmniejszyć ziarnistość uzyskanego kodu. Ma to duże znaczenie dla systemów równoległych, w których balansowanie obciążeniem jednostek przetwarzających odgrywa kluczową rolę w uzyskaniu optymalnych wyników.

4. BADANIA EKSPERYMENTALNE

Do przeprowadzenie badań eksperymentalnych dla pętli idealnie zagnieżdżonych wybrany został zbiór pętli testowych Livermore [48]. Dla algorytmów dedykowanych dla pętli dowolnie zagnieżdżonych testy przeprowadzone zostały na zestawie testów NPB (ang. *NAS Parallel Benchmarks - NPB*) [6], [54], [87].

Zbiór pętli testowych Livermore składa się z 24 pętli (załącznik B), pozwalających na zmierzenie wydajności jednostki obliczeniowej lub systemu jednostek obliczeniowych tworzących maszynę wieloprocesorową. Pętle z tego zbioru bardzo często wykorzystywane są do pomiaru jakości kompilatorów umożliwiających automatyzację procesu transformacji pętli sekwencyjnych na ich odpowiedniki równoległe.

Zestaw testów NPB (załącznik B) jest to zbiór programów ułatwiających szacowanie wydajności systemów równoległych. Testy pochodzą z aplikacji do symulacji dynamiki przepływu (ang. *Computational Fluid Dynamics - CFD*), zaprojektowanej w centrum rozwojowym NASA. W skład testów NPB wchodzą źródła pięciu jąder systemu oraz trzech aplikacji CFD. Pięć jąder systemu służy do symulacji pięciu metod numerycznych wykorzystywanych w aplikacji CFD. Trzy aplikacje CDF służą do symulacji przepływu danych oraz symulacji obliczeń wykorzystywanych w pełnej aplikacji CFD. Testy NPB często uważane są jako wyznacznik wydajności komputerów, zarówno sekwencyjnych jak i równoległych. Specyfikacja powyższych testów dostępna jest w postaci algorytmów, a implementacja pozostała otwartą kwestią w celu uniezależnienia się od konkretnej architektury systemowej. Problem rozwiązywany przez algorytmy został opisany w taki sposób, aby istniało jedno unikalne rozwiązanie, a poprawność danych wyjściowych możliwa była do zweryfikowania w jednoznaczny sposób. Wybór struktur danych, algorytmów, alokacji

procesorów oraz sposobu użycia pamięci pozostaje po stronie implementującego. Poniżej przedstawiona została krótka charakterystyka siedmiu dostępnych testów.

- **BT** (aplikacja CFD) – jest symulacją aplikacji CFD do rozwiązania trójwymiarowego układu równań Navier-Stokes'a. Skończona liczba różnic rozwiązań problemu bazuje na przybliżonej faktoryzacji metody kierunków naprzemiennych (ang. *Alternating Direction Implicit - ADI*), rozdzielającej wymiary x,y, z. System wynikowy to bloki 5x5 macierzy trójdiagonalnej, rozwiązywane sekwencyjnie wzdłuż każdej z osi.
- **SP** (aplikacja CFD) – symulacja aplikacji CFD o podobnej strukturze jak test BT. Skończona liczba różnic rozwiązań problemu bazuje na przybliżonej faktoryzacji schematu Beam-Warming'a, która rozdziela wymiary x, y i z. Wynikowy system jest pentadiagonalnym układem równań, rozwiązywanych sekwencyjnie wzdłuż każdej z osi.
- **LU** (aplikacja CFD) – symulacja aplikacji CFD wykorzystującą symetryczną metodę kolejnych nadrelaksacji (ang. *symmetric successive over-relaxation - SSOR*) do rozwiązania siedmio-blokowego diagonalnego systemu, wynikającego ze skończonej różnicy dyskretyzacji trójwymiarowego układu równań Navier-Stokes'a, poprzez podział na dolne i górne trójkątne systemy.
- **LU-HP** – jest to hiperpłaszczyznowa (ang. *hyper-plane*) wersja testów LU
- **FT** (jądro) – test dotyczy pomiaru czasu rozwiązywania równań trójwymiarowej różniczki cząstkowej z wykorzystaniem prostej i odwrotnej Szybkiej Transformacji Fouriera (ang. *forward/inverse Fast Fourier Transform FFT*).
- **MG** (jądro) – W teście wykorzystywane są cztery iteracje pięcio-cyklicznej metody wielosiatkowej w celu rozwiązania trójwymiarowego układu równań Poisson'a. Test ma za zadanie sprawdzenie przemieszczania danych na krótkie jak i długie dystansy.
- **CG** (ang. *Conjugate Gradient*) - przy użyciu metody Gradientu Sprzężonego obliczane jest przybliżenie do najmniejszej wartości własnej dla dużej, nieregularnej rzadkiej macierzy (and. *sparse matrix*). Test zawiera nieustaloną strukturę obliczeń oraz komunikacji przy użyciu macierzy z losowo wygenerowanymi lokalizacjami wpisów.
- **EP** (ang. Embarrassingly Parallel Benchmark) - W teście tym generowana jest losowa para odchyleń Gaussowskich (ang. *Gaussian Random Deviates*) zgodnie

z zadanym schematem. Test ten pozwala oszacować górną granicę wydajności obliczeń zmiennoprzecinkowych, nie wymaga znaczącej komunikacji międzyprocesorowej.

- **UA** (ang. Unstructured Adaptive) – w teście mierzony jest wpływ ciągłych i nieregularnych odwołań do pamięci.

4.1. Kryteria wyboru pętli do testów

Wyboru pętli dokonano na podstawie struktury pętli oraz występujących w nich relacji opisujących zależności. Ze względu na strukturę do testów wybrano pętle programowe dowolnie zagnieżdżone (idealnie i nieidealnie zagnieżdżone), o stałym kroku znanym w trakcie kompilacji. Górna granica pętli mogła być podana jako wartość stała lub jako parametr. W ciele pętli akceptowane są odwołania do zmiennych skalarnych oraz do zmiennych tablicowych dowolnego wymiaru. Ciało pętli nie może zawierać instrukcji skoku (np. goto, continue), instrukcji zmieniających krok pętli w czasie wykonania oraz wywołań funkcji.

Ze względu na zależności do testów zostały wybrane pętle zawierające przynajmniej jedną zależność prostą (ang. *flow dependences*). Jeśli pętle zawierała jedynie zależności odwrotne (ang. *anti dependences*), zależności po wyjściu (ang. *output dependences*) lub nie zawierała żadnych zależności, to nie została przeanalizowana pod kątem tworzenia niezależnych fragmentów kodu. Pętla taka została jedynie zaliczona do ogólnych statystyk (tab. 4.2). Pętle nie zawierające żadnych zależności mogą zostać zrównoleglone w dowolny sposób. Dla pętli zawierających zależności odwrotne lub po wyjściu możliwe zastosowanie odpowiednich transformacji [1], [22] pozwalających na ich eliminację np. stworzenie kopii zmiennych wywołujących zależności między poszczególnymi instrukcjami.

Tabela 4.1. Przykład redukcji zależności odwrotnych oraz po wyjściu

1. a = b 2. b = c + 3	b’ = b 1. a = b’ 2. b = c + 3
zależność odwrotna	**brak zależności odwrotnej**
1. a = 2 * b 2. x = a 3. a = 3 * y	1. a1 = 2 * b 2. x = a1 3. a = 3 * y
zależność po wyjściu	**brak zależności po wyjściu**

W tabeli 4.1. przedstawiono przykład instrukcji z zależnościami: odwrotna i po-wyjściu, oraz zmodyfikowany kod nie zawierający tych zależności.

Zgodnie z powyższymi kryteriami z całego zbioru NPB (431 pętli), pętli spełniających pierwszy warunek (struktura pętli) zakwalifikowanych zostało 257 pętli, co stanowi ~60% wszystkich pętli NPB. Liczba pętli zawierających przynajmniej jedną zależność prostą, ze zbioru pętli spełniających pierwszy warunek wynosi 149, co stanowi ~35% wszystkich pętli jak również ~58% pętli spełniających pierwsze kryterium wyboru. W tab. 4.2 przedstawione zostały dane statystyczne dla każdego z testów w zależności od ilości przeanalizowanych pętli wraz z ogólnym podsumowaniem u dołu tabeli. W pierwszej kolumnie przedstawiono nazwę testu, druga kolumna przedstawia liczbę wszystkich pętli typu „do" dla konkretnego testu, w trzeciej przedstawiono ilość pętli spełniających kryterium budowy. Kolejna kolumna przedstawia ostateczną ilość pętli zakwalifikowanych do analizy (pętle spełniają ograniczenia pod względem struktury oraz warunek nałożony na zależności) w odniesieniu do wszystkich pętli dla konkretnego testu. W ostatniej kolumnie przedstawiono ilość pętli niezakwalifikowanych do analizy.

Tabela 4.2. Statystyki ilościowe i procentowe dla testu NPB

Nazwa testu	**Wszystkich pętli „do"**	**Pętli spełniających kryterium struktury**		**Pętli zakwalifikowanych do analizy**		**Pętli odrzuconych**	
		Ilość	**Procent (%)**	**Ilość**	**Procent (%)**	**Ilość**	**Procent (%)**
BT	57	37	64,9	23	40,4	34	59,6
CG	24	9	37,5	4	16,7	20	83,3
EP	1	0	0	0	0,0	1	100
FT	13	3	23,1	2	15,4	11	84,6
LU-HP	46	32	69,6	17	37,0	29	63
LU	45	32	71,1	18	40,0	27	60
MG	31	16	51,6	11	35,5	20	64,5
SP	50	37	74	34	68,0	16	32
UA	164	91	55,5	40	24,4	124	75,6
Zbiorczo:	**431**	**257**	**59,6**	**149**	**34,6**	**281**	**65,5**

4.2. Modyfikacje pętli

W przypadku, gdy niemożliwe było wyznaczenie niezależnych wątków zgodnie z zaproponowanymi algorytmami, pętla została poddana modyfikacji. W trakcie badań okazało się, iż najczęstszą przyczyną uniemożliwiającą wyznaczenie niezależnych wątków jest zbyt duża liczba zależności, powodujących wyznaczenie skomplikowanego grafu z wieloma zależnościami o wspólnych końcach. Najczęstszą modyfikacją wprowadzoną do pętli była zamiana zmiennych skalarnych na zmienne tablicowe. W wielu przypadkach pozwoliło to na znaczną redukcję liczby zależności, co przełożyło się bezpośrednio na uproszczenie grafu zależności oraz zmniejszenie relacji o wspólnych końcach.

Ze zbioru 149 pętli zakwalifikowanych do eksperymentów, dla 53 pętli wyznaczone zostały niezależne wątki (liczba wątków >1), co stanowi 12% wszystkich pętli ze zbioru NPB, zaś 21% ze zbioru pętli poddanych analizie. Modyfikacje pętli źródłowych, opisane w poniższym punkcie, pozwoliły na zwiększenie liczby pętli, dla których wyznaczono niezależne wątki. Liczba pętli, dla których uzyskano niezależne wątki, wzrosła do 59, co stanowi 14% całości, natomiast 23% pętli poddanych analizie. Omówione wyniki zostały podsumowane w tab. 4.3.

Tabela 4.3. Zestawienie ilościowe pętli oryginalnych i zmodyfikowanych

	Pętle z niezależnymi wątkami		
	Ilość pętli	**W odniesieniu do całości NPB (%)**	**W odniesieniu do analizowanego zbioru (%)**
Oryginalna wersja	53	12	21
Zmodyfikowana wersja	59	14	23

4.3. Transformacje kodu wynikowego

W celu zwiększenia lokalności kodu wynikowego zastosowane zostały trzy transformacje: aglomeracja (ang. a*gglomeration*), skalaryzacja tablic (ang. *scalar replacement*) i podział na bloki / blokowanie (ang. *cache blocking tiling*). Pierwsza z transformacji umożliwia zwiększenie ziarna kodu. Skalaryzacja tablic umożliwia efektywne wykorzystanie rejestrów procesora, natomiast podział na bloki pozwala na redukcję chybień do pamięci podręcznej (ang. *cache*).

Aglomeracja (ang. *agglomeration*) [34] – pozwala na zwiększenie ziarna kodu poprzez łączenie zadań, co prowadzi do redukcji liczby zadań przy jednoczesnym wzroście wielkości pojedynczego zadania. W trakcie badań zaobserwowano, że wykonanie fragmentu kodu pozbawionego synchronizacji na pojedynczej jednostce przetwarzającej jest bardzo często nieopłacalne. W przypadku, gdy fragment kodu zawiera dwie instrukcja, utrata wydajności była szczególnie widoczna. Złączenie niezależnych fragmentów w większe paczki i przydzielenie takiej paczki do pojedynczego procesora pozwoliło na:

- zredukowanie kosztów zarządzania wątkami
- zwiększenie lokalności kodu, w szczególności redukcję chybień do pamięci podręcznej procesora.

W konsekwencji przyczyniło się to do wzrostu wydajności na systemach wieloprocesorowych. Podstawowym warunkiem prawidłowej aglomeracji, w kontekście prowadzonych badań, jest prawidłowy dobór zadań tworzących jedno większe ziarno. Zadania poddane aglomeracji powinny odwoływać się do obszarów pamięci przylegających do siebie lub możliwie bliskich sobie, o ile jest to możliwe. Przy spełnieniu powyższego warunku, możliwe jest zmniejszenie ilości chybień do pamięci podręcznej procesora. Wynika to ze sposobu przenoszenia danych z pamięci do pamięci podręcznej jednostki przetwarzającej.

Skalaryzacja tablic (ang. *scalar replacement*) [1], [85] – technika ta polega na konwersji wybranych odwołań do zmiennych tablicowych na odwołania do zmiennych skalarnych. Kompilatory traktują odwołania do zmiennych tablicowych jak wyrażenia. Każdorazowe odwołanie do tablicy skutkuje obliczeniem adresu komórki pamięci. W przypadku, gdy w ciele pętli odwołujemy się to tej samej komórki pamięci, niezależnie od zmiany indeksu pętli, wielokrotne obliczenie adresu wprowadza niepotrzebne opóźnienie. W związku z tym, iż kompilatory dzięki zastosowaniu metody kolorowania grafów potrafią efektywnie przypisać zmienne skalarne do rejestrów procesora, powyższe odwołania do zmiennych tablicowych można przekształcić na odwołania do zmiennych skalarnych. Po wykonaniu opisanej transformacji uzyskujemy wielokrotne odwołanie do rejestru. W przykładzie 4.1 przedstawiono zastosowanie skalaryzacji zmiennych dla tablicy A[]. W oryginalnym kodzie odwołania do elementów tablicy A[] zmieniają się wraz ze zmianą zmiennej indeksowej dla najbardziej zewnętrznej pętli, natomiast pozostają niezmienne względem zmiennej indeksowej ‘j’. Obliczenie adresu tej samej komórki tablicy dla każdej iteracji wewnętrznej pętli jest nadmiarowe. Zmodyfikowany kod pętli redukuje w znaczny sposób ilość obliczeń komórki tablicy A[]. Analizując bezpośrednio zmodyfikowany

kod wewnątrz pętli z indeksem ‘j’ nie mamy żadnego odwołania do tablicy A[], lecz odwołanie do tablicy A[] przed i po wewnętrznej pętli należałoby przypisać do pętli z indeksem ‘j’. W związku z tym, dla oryginalnej pętli mamy 2 x N x M odwołań do tablicy A[], natomiast dla pętli zmodyfikowanej uzyskujemy jednie 2 x N odwołań. W takim przypadku jeśli M (górna granica drugiej pętli) będzie odpowiednio duża, różnica odwołań między dwoma powyższymi pętlami może być istotna. Kompilator wykorzystujący algorytm kolorowania grafów powinien prawidłowo przydzielić zmienną skalarną T do jednego z rejestrów procesora.

```
for ( i = 0; i < N; i++ )
  for ( j = 0; j < M; j++)
    A[ i ] = A[ i ] + B[ j ]
```

```
for ( i = 0; i < N; i++ ) {
  T = A[ i ]
  for ( j = 0; j < M; j++)
    T = T + B[ j ]
  A[ i ] = T
}
```

```
for ( j = 0; j < M; j += S)
  for ( i = 0; i < N; i++ )
    for (jj=j; j<min(M, j+S); jj++)
      A[ i ] = A[ i ] + B[ jj ]
```

Przykład 4.1. Oryginalny kod pętli (po lewej), kod poddany skalaryzacji tablic (kolumna środkowa) oraz kod poddany blokowaniu (po prawej)

Podział na bloki / blokowanie (ang. *blocking / tiling*) [1] – transformacja umożliwia redukcję chybień do pamięci podręcznej (ang. *cache*) jednostki przetwarzającej. Większość dzisiejszych procesorów oprócz rejestrów zawiera również szybką pamięć podręczną pierwszego i drugiego poziomu. Upraszczając, w trakcie wykonania programu przy odwołaniu do konkretnych danych sprawdzane jest czy znajdują się one w rejestrze. Jeśli nie, to sprawdzane jest czy dane znajdują się w pamięci podręcznej procesora. W przypadku, gdy pamięć podręczna procesora nie zawiera żądanych danych mamy do czynienia z chybieniem do pamięci podręcznej (ang. *cache misses*) i niezbędne jest sięgnięcie do pamięci współdzielonej. Ze względu na to, iż prędkość dzisiejszych procesorów wielokrotnie przewyższa czas dostępu do pamięci współdzielonej, procesor oczekuje na załadowanie potrzebnych danych do szybkiej pamięci podręcznej w celu wykonania obliczeń. Dla instrukcji wykonywanych wielokrotnie, np. instrukcji zawartych w pętlach programowych, ilość chybień do pamięci cache może mieć znaczący wpływ na czas wykonania obliczeń. Jeśli odwołania do pamięci cache mają charakter sekwencyjny tzn. każda z iteracji odwołuje się do komórki pamięci przylegającej do komórki z poprzedniej iteracji to podział przestrzeni iteracji pętli na bloki nie przekraczające wielkości pamięci cache może przynieść znaczną poprawę trafień do pamięci podręcznej. Przy czym należy pamiętać o sposobie składowania tablic w pamięci w zależności od języka programowanie. W Fortranie tablice alokowane są w pamięci kolumnami (ang. *column-major order*), natomiast

w języku C/C++ rzędami (ang. *row-major order*). Kod po zastosowaniu transformacji blokowania dla przykładu 4.1 przedstawiono wcześniej. Jeśli zmienna S będzie wystarczająco mała, to odwołanie do tablicy B[] spowoduje chybienie do pamięci cache jedynie dla każdej wartości J (czyli B[jj], gdzie jj = J), dla odwołań do tablicy B[jj] gdzie jj ≠ J uzyskamy trafienie do pamięci cache. Dla odpowiednio dużych wartości stałych N i M i prawidłowo dobranej wartości S uzyskujemy znaczącą redukcję chybień do pamięci podręcznej procesora.

4.4. Badania eksperymentalne wybranych pętli – zbiór pętli Livermore

W celu dokonania pomiarów efektywności algorytmów, dedykowanych dla pętli idealnie zagnieżdżonych, przeprowadzono badania dla pętli idealnie zagnieżdżonych pochodzących ze zbioru pętli Livermore [48]. W tab. 4.4.1. przedstawiono wyniki uzyskane poprzez zastosowanie algorytmów 3.1 – 3.5. Pierwsza kolumna tabeli przedstawia nazwę pętli. Dla wszystkich pętli, na których przeprowadzono badania, górne granice zostały sparametryzowane. W celu zredukowania liczby zależności lub umożliwienia wyszukania zależności przy pomocy narzędzia Petit, niektóre z pętli zostały ręcznie zmodyfikowane poprzez zastosowanie ogólnie znanych transformacji:

- rozszerzenie skalaru (ang. *scalar expansion*) – dla pętli K10 i K23,
- propagacja stałych (ang. *constant propagation*) – dla pętli K8,
- zastąpienie nieliniowych funkcji w ciele pętli liniowymi dla tych samych argumentów – dla pętli K22.

Kolejna kolumna przedstawia czy źródło pętli zostało zmodyfikowane. Trzecia kolumna zawiera informację czy konieczne było zastosowanie kroku 5a z algorytmu 3.3 (ze względu na to, że badane pętle zostały sparametryzowane, użycie kroku 5b było niemożliwe). Z tych samych powodów w algorytmie 3.5 skorzystano z kroku 3a. W czwartej kolumnie przedstawiono, który z algorytmów (3.3 lub 3.4) został wykorzystany do wyznaczenia początków krańcowych dla niezależnych fragmentów kodu. Ostatnia z kolumn przedstawia liczbę niezależnych fragmentów kodu wyznaczonych w trakcie prowadzonych badań. Zapis w postaci „loop” $\geq X \Rightarrow Y$ oznacza, że dla każdej podprzestrzeni „loop” jeśli warunek „loop” $\geq X$ jest spełniony (gdzie ‘loop’ jest górną granicą pętli) to liczba niezależnych fragmentów kodu wyznaczonych dla określonej pętli w każdej podprzestrzeni „loop” wynosi Y.

Badania dla pętli ze zbioru testowego Livermore przeprowadzone zostały w środowisku równoległym (załącznik A):

- maszyna równoległa: 2 x Intel Xeon E5310, 2 GB RAM
- kompilator: GCC (ver. 4.2)

Tabela 4.4.1. Wyniki działania algorytmów 3.1 - 3.5 dla pętli ze zbioru Livermore

Pętla	**Zmodyfikowano**	**Krok 5 Alg. 3.3**	**Alg.**	**Liczba niezależnych fragmentów**
K1 - hydro fragment	nie	-	Alg. 3.3	loop $\geq 2 \Rightarrow$ n
K5 - tri-diagonal elimination, below diagonal	nie	5a	Alg. 3.4	loop $\geq 2 \Rightarrow$ 1 w każdej podprzestrzeni loop
K6 – general linear recurrence equations	nie	5a	Alg. 3.4	i $\geq 2 \Rightarrow$ 1 w każdej podprzestrzeni loop*n
K7 – equation of state fragment	nie	nie	Alg. 3.3	loop $\geq 2 \Rightarrow$ n
K8 – ADI integration	tak	5a	Alg. 3.4	n – 1 w każdej podprzestrzeni loop
K9 – integrate predictors	nie	-	Alg. 3.3	loop $\geq 2 \Rightarrow$ n
K10 - difference predictors (vec)	tak	-	Alg. 3.3	loop $\geq 2 \Rightarrow$ n
K12 – first difference	nie	-	Alg. 3.3	loop $\geq 2 \Rightarrow$ n
K21 – matrix*matrix product	nie	5a	Alg. 3.4	n $\geq 1 \Rightarrow$ 24*n w każdej podprzestrzeni loop
K22 – Planckian distribution	nie	-	Alg. 3.3	loop $\geq 2 \Rightarrow$ n
K23 – 2-D implicit hydrodynamics fragment	tak	5a	Alg. 3.4	n $\geq 3 \Rightarrow$ 1 w każdej podprzestrzeni 5* loop

Wyniki uzyskane poprzez zastosowanie algorytmów 3.6 – 3.9 dla pętli idealnie zagnieżdżonych (rozdział 4.5), zawierających pojedynczą instrukcję są zbieżne z wynikami przy zastosowaniu algorytmów 3.3 – 3.5 dla tych samych pętli. Wynika to z faktu, iż dla takich pętli iteracja zbudowana jest z pojedynczej instrukcji. W związku z powyższym, wyniki dla pętli Livermore zostaną omówione w sposób zbiorczy dla całego zestawu pętli testowych, ze szczególnym naciskiem na pętle idealnie zagnieżdżone, zawierające więcej niż jedną instrukcję.

Dla pętli K5, K6, K23 (patrz tab. 4.4.1) ze zbioru Livermore wyznaczono jeden niezależny fragment kodu. W innych przypadkach możliwe było wyznaczenie większej liczby niezależnych fragmentów kodu:

- K1, K7, K9, K10, K12, K22 - w pełnej przestrzeni iteracji pętli,
- K8, K21, K23 – z zredukowanej przestrzeni iteracji pętli.

Poniżej omówiono wynik dla pętli idealnie zagnieżdżonych zawierających więcej niż jedną instrukcję – pętle K8, K10 i K22. Testy przeprowadzono zgodnie z zaproponowanymi algorytmami oraz z zastosowaniem transformacji kafelkowania (ang. *bloking*).

4.4.1. Pętla K8

Postać pętli wejściowej przedstawiona została w tab. 4.4.2(a), gdzie pojedynczą iterację pętli stanowi sześć instrukcji. Górne granice skrajnych pętli (zewnętrznej i wewnętrznej) są sparametryzowane, natomiast pętla wewnętrzna zawiera trzy iteracje. W trakcie analizy pętli wykorzystane zostały algorytmy 3.3, 3.4, oraz 3.5. Wyznaczenie niezależnych fragmentów kodu w całej przestrzeni iteracji pętli było niemożliwe. W związku z tym przestrzeń iteracji, zgodnie z algorytmem 3.4, została zawężona do dwóch wewnętrznych pętli (indeksy kx i ky), dzięki czemu zredukowana została liczba występujących zależności. Zgodnie z tab. 4.4.1 i 4.4.2(b) wyznaczono n-1 niezależnych fragmentów kodu w każdej iteracji pętli najbardziej zewnętrznej (o zmiennej indeksowej ‘l’).

Badania przeprowadzono zgodnie z zaproponowanymi algorytmami oraz przy zastosowaniu dodatkowej transformacji kafelkowania dla różnej wielkości bloku B (B = 8, 16, 32). Pozytywne przyśpieszenie (większe od 1) zgodnie z rys. 4.4.1 uzyskano dla 4 i 8 procesorów, przy czym dla 8 procesorów przy zrównolegleniu względem zmiennej indeksowej ‘ky’ było najwyższe. Zastosowanie dodatkowej transformacji kafelkownia, wpłynęło negatywnie na uzyskane wyniki. W takim przypadku wraz ze wzrostem liczby wątków przyśpieszenie malało, co może świadczyć o zbyt dużych kosztach ponoszonych na zarządzanie wątkami w stosunku do ilości wykonywanych obliczeń przez pojedynczy wątek. Dodatkowo należy zwrócić uwagę na koszty ponoszone na tworzenie i unicestwianie regionu równoległego z każdą iteracja zewnętrznej pętli (patrz tab. 4.4.2b).

Tabela 4.4.2. Postać sekwencyjna i równoległa pętli K8

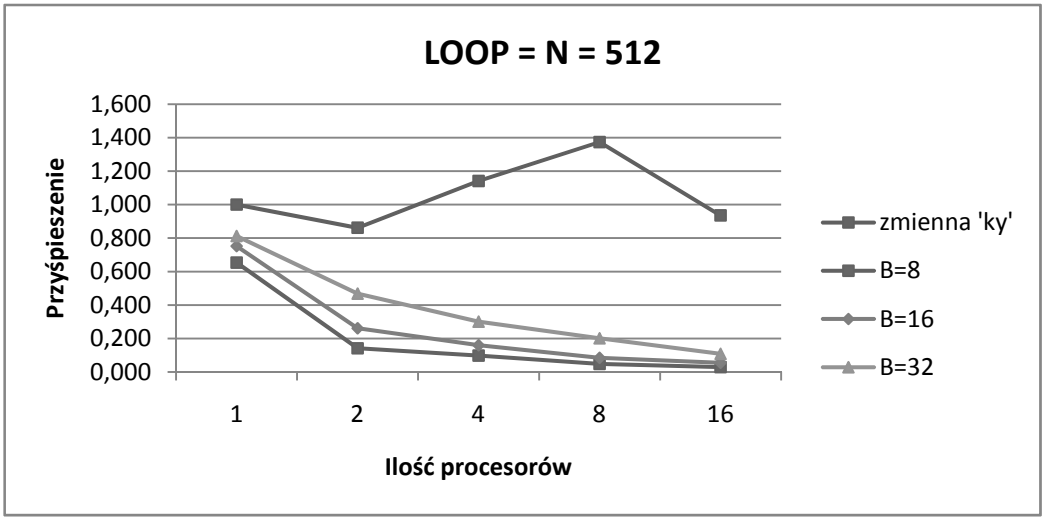
```
for l=1 to loop do
  for kx=1 to 2 do
    for ky=1 to n-1 do
      du1(ky) = u1(0,ky+1,kx) - u1(0,ky-1,kx);
      du2(ky) = u2(0,ky+1,kx) - u2(0,ky-1,kx);
      du3(ky) = u3(0,ky+1,kx) - u3(0,ky-1,kx);
      u1(1,ky,kx)=u1(0,ky,kx)+a11*du1(ky)+a12*du2(ky)+a13*du3(ky)    +    sig*[u1(0,ky,kx+1)-
                  2.0*u1(0,ky,kx)+u1(0,ky,kx-1)];
      u2(1,ky,kx)=u2(0,ky,kx)+a21*du1(ky)+a22*du2(ky)+a23*du3(ky) + sig*[u2(0,ky,kx+1)-
                  2.0*u2(0,ky,kx)+u2(0,ky,kx-1)];
      u3(1,ky,kx)=u3(0,ky,kx)+a31*du1(ky)+a32*du2(ky)+a33*du3(ky) + sig*[u3(0,ky,kx+1)-
                  2.0*u3(0,ky,kx)+u3(0,ky,kx-1)];
    endfor
  endfor
endfor
```

(a) Pętla poddana analizie

```
if (n >= 2) {
  seq for(l = 1; l <= loop; l++) {
    par for(ky = 1; ky <= n-1; ky++) {
      seq for(kx = 1; kx <= 2; kx++) {
        s1(l,kx,ky);
}
```

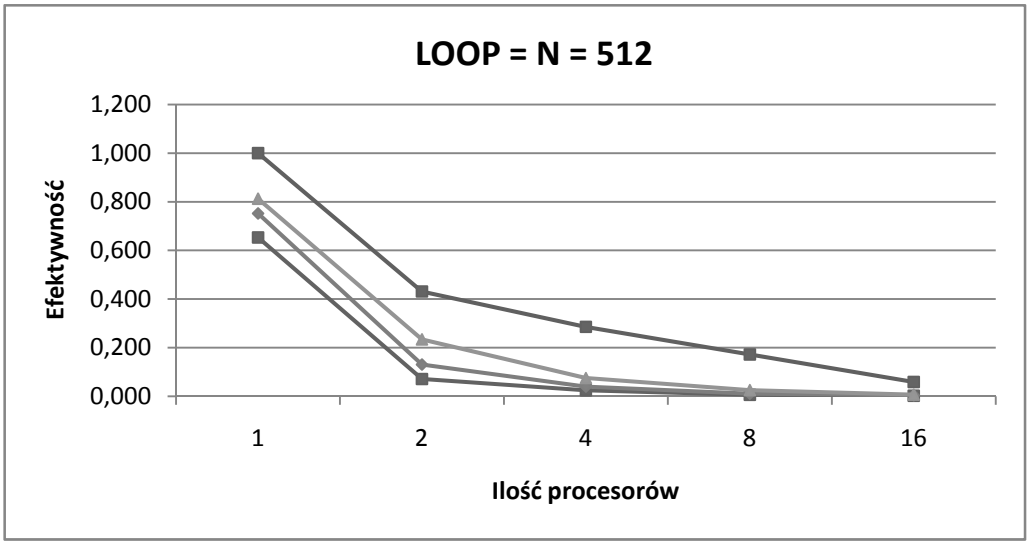
(b) Pętla gotowa do zrównoleglenia (s1 oznacza ciało oryginalnej pętli)

Najwyższą efektywność uzyskano dla zrównoleglenia zgodnie z zaproponowanymi algorytmami (bez dodatkowych transformacji) dla pojedynczego procesora (rys. 4.1.2).



Rysunek 4.1.1. Przyśpieszenie dla pętli K8 [op. własne]

Wraz ze wzrostem liczby procesorów efektywność malała. Jednakże, dla każdego pomiaru wartości przyśpieszenia jak i efektywności były największe przy bezpośrednim zrównolegleniu względem zmiennej ‘ky’. Zastosowanie transformacji kafelkowania w każdym z przypadków powodowało spadek wydajności obliczeń.



Rysunek 4.1.2. Efektywność dla pętli K8 [op. własne]

4.4.2. Pętla K10

Oryginalna pętla została zmodyfikowana do postaci przedstawionej w tab. 4.4.3a. Modyfikacji poddane zostały zmiennie skalarne względem, których zastosowano transformację wektoryzacji. Umożliwiło to znaczną redukcję liczby zależności, co w konsekwencji pozwoliło na wyznaczenie niezależnych fragmentów kodu (np. zmienna skalarna ‘ar’ została przekształcona na odwołanie do macierzy ‘ar(l,i)’). Do analizy pętli wykorzystane zostały algorytmy 3.3 oraz 3.5. Analiza dotyczyła pełnej przestrzeni iteracji pętli, w której wyznaczono ‘n’ niezależnych fragmentów kodu.

Tabela 4.4.3. Postać sekwencyjna i równoległa pętli K10

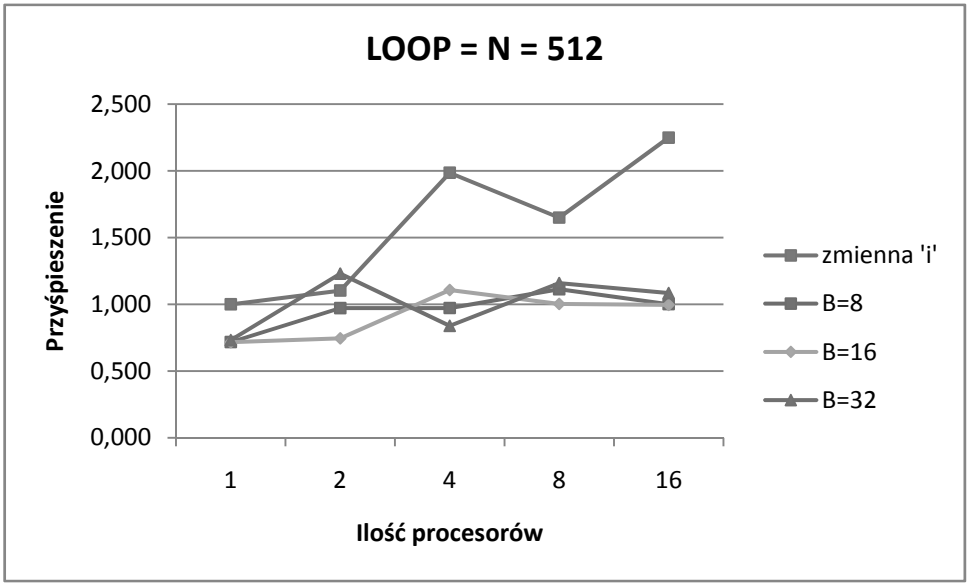
```
for l=1 to loop do
  for i=0 to n-1 do
    ar(l,i) = cx(i,4)
    br(l,i) = ar(l,i) - px(i,4)
    px(i,4) = ar(l,i);
    cr(l,i) = br(l,i) - px(i,5)
    px(i,5) = br(l,i);
    ar(l,i) = cr(l,i) - px(i,6)
    px(i,6) = cr(l,i);
    br(l,i) = ar(l,i) - px(i,7)
```

```
if (loop >= 2) {
  par for(i = 0; i <= n-1; i++) {
    seq for(l = 1; l <= loop; l++) {
      s1(l,i);
}
```

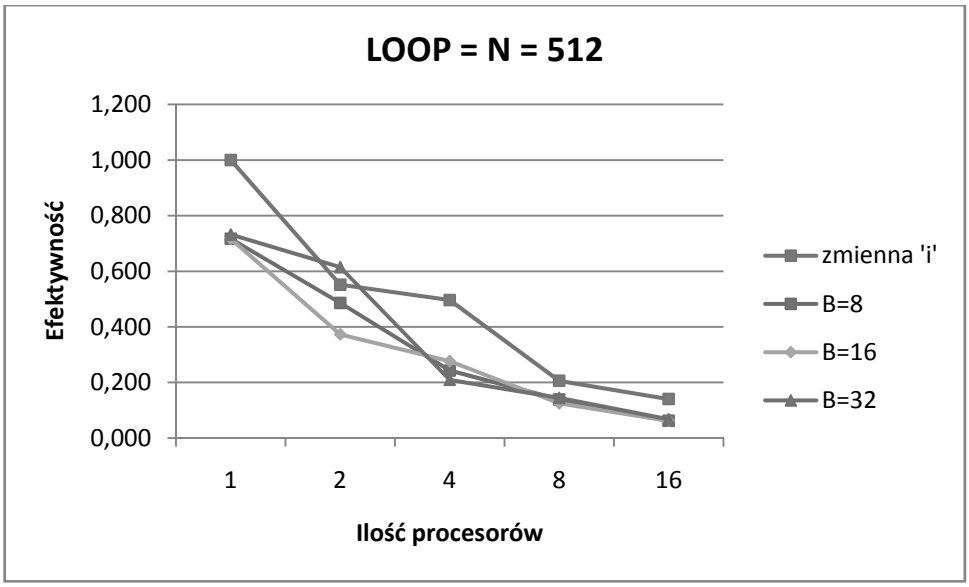
px(i,7) = ar(l,i);
cr(l,i) = br(l,i) - px(i,8)
px(i,8) = br(l,i);
ar(l,i) = cr(l,i) - px(i,9)
px(i,9) = cr(l,i);
br(l,i) = ar(l,i) - px(i,10)
px(i,10) = ar(l,i);
cr(l,i) = br(l,i) - px(i,11)
px(i,11) = br(l,i);
px(i,13) = cr(l,i) - px(i,12)
px(i,12) = cr(l,i);
endfor
endfor	

(a) Pętla poddana analizie	(b) Pętla gotowa do zrównoleglenia (s1 oznacza ciało oryginalnej pętli)

Pomiar czasu przeprowadzono zgodnie z uzyskanym kodem (wyjście z algorytmu 3.5) oraz przy zastosowaniu dodatkowej transformacji kafelkowania dla różnej wielkości bloku B (B = 8, 16, 32). Najwyższe przyśpieszenie uzyskano dla 16 procesorów przy bezpośrednim zrównolegleniu niezależnych fragmentów kodu - bez dodatkowej transformacji (patrz rys. 4.2.1). Najwyższą efektywność (rys. 4.2.2), dla więcej niż jednego procesora uzyskano przy zrównolegleniu niezależnych fragmentów połączonych z kafelkowaniem.



Rysunek 4.2.1. Przyśpieszenie dla pętli K10 [op. własne]



Rysunek 4.2.2. Efektywność dla pętli K10 [op. własne]

4.4.3. Pętla K22

Pętla K22 została przeanalizowana zgodnie z zaproponowanymi algorytmami w oryginalnej postaci (tab. 4.4.4a). Do analizy pętli użyto algorytmów 3.3 oraz 3.5, dla całej przestrzeni iteracji pętli. W pętli wyznaczono ‘n’ niezależnych fragmentów kodu, gdzie każdy z fragmentów zawiera ‘loop’ iteracji (tab. 4.4.4b).

Tabela 4.4.4. Postać sekwencyjna i równoległa pętli K22

```
for l=1 to loop do
 for k=0 to n-1 do
  y(k) = u(k) / v(k)
  w(k) = x(k) / ( exp( y(k) ) -1.0 )
 endfor
endfor
```

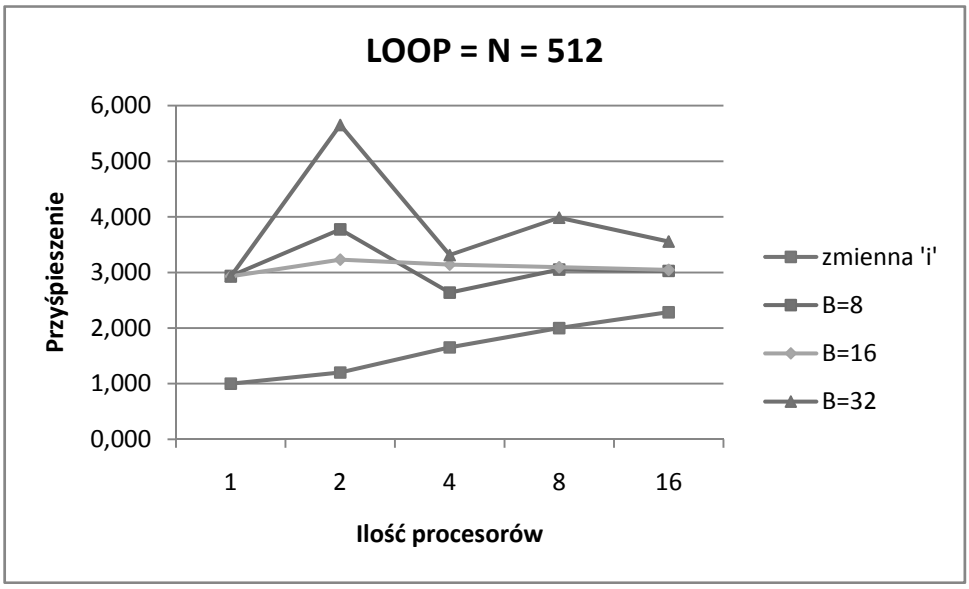
(a) Pętla poddana analizie

```
if (loop >= 2) {
  par for(k = 0; k <= n-1; k++) {
    seq for(l = 1; l <= loop; l++) {
      s1(l,k);
}
```

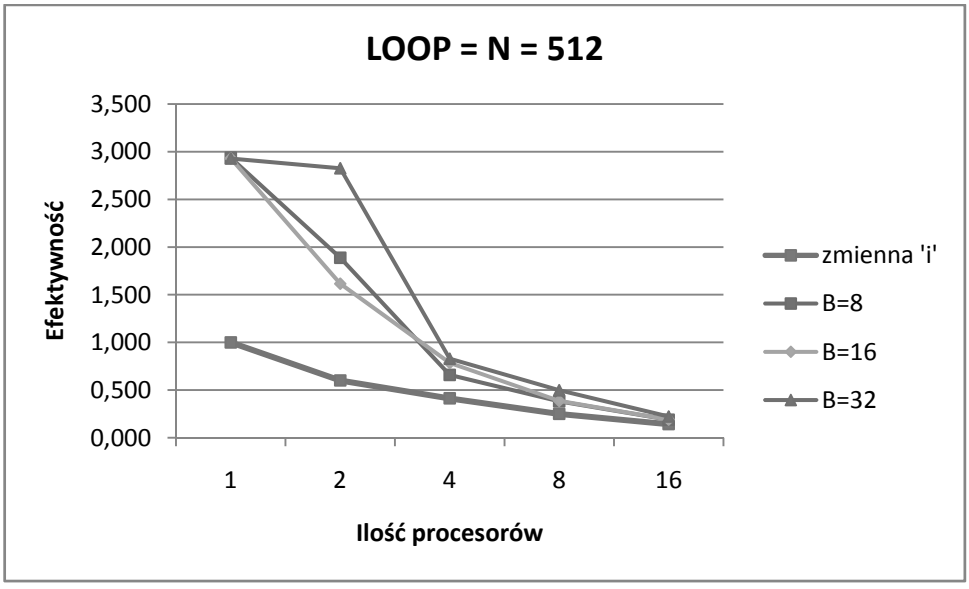
(b) Pętla gotowa do zrównoleglenia (s1 oznacza ciało oryginalnej pętli)

Zmierzono czas wykonania pętli zgodnie z wyjściem algorytmu 3.5 oraz dodatkowo z zastosowaniem transformacji kafelkowania dla różnej wielkości bloku B

(B = 8, 16, 32). Zastosowanie transformacji kafelkowania pozwoliło na uzyskanie wyższych wartości przyśpieszenia (dla wszystkich wartości bloku B) w odniesieniu do wyników uzyskanych poprzez bezpośrednie zrównoleglenie niezależnych fragmentów kodu (rys. 4.3.1). W szczególności dla dwóch procesorów, gdzie przyśpieszenie wyniosło ~5,6 (dla B=32). Jaka można było przypuszczać z uzyskanych wartości przyśpieszenia, najwyższą efektywność uzyskano również dla dwóch procesorów i transformacji kafelkowania (B=32). Patrząc jednak na rys. 4.3.2, można zauważyć jak efektywność drastycznie maje dla transformacji kafelkowania wraz ze wzrostem liczby procesorów, aby w konsekwencji zbliżyć się do efektywności przy bezpośrednim zrównolegleniu względem zmiennej indeksowej ‘i’ dla 16 procesorów.



Rysunek 4.3.1. Przyśpieszenie dla pętli K22 [op. własne]



Rysunek 4.3.2. Efektywność dla pętli K22 [op. własne]

4.5. Badanie eksperymentalne wybranych pętli – zbiór testów NPB

W niniejszym rozdziale przedstawiono szczegółową analizę dziesięciu pętli ze zbioru NPB. Osiem z nich przeanalizowano w oryginalnej postaci, natomiast w dwóch dokonano modyfikacji w celu redukcji zależności. Główne kryteria wyboru pętli do zrównoleglenia to:

- ilość wyznaczonych niezależnych fragmentów kodu,
- ilość obliczeń w ciele pętli.

Zrównoleglenie pętli, których ciało zawiera niewiele obliczeń, okazuje się być często nieopłacalne ze względu na czas potrzebny do utworzenia i synchronizacji wątków oraz czas potrzebny na kopiowanie niezbędnych danych do pamięci podręcznej procesorów w celu wykonania operacji arytmetycznych. Korzyści ze zrównoleglenia pętli muszą przewyższać straty niezbędne do zarządzania wielowątkowością.

Wyniki pomiarów dla wszystkich pętli przedstawione zostały w załączniku B w tab. B1 oraz B2 (katalog stat_NPB). Pierwsza tabela zawiera wyniki pomiarów dla oryginalnych wersji pętli, natomiast druga przedstawia wyniki dla pętli zmodyfikowanych. Modyfikacji poddane zostały pętle, dla których nie wyznaczono niezależnych fragmentów w oryginalnej postaci. Przekształcenia pętli zawartych w tab. B2 miały na celu redukcję zależności, a w konsekwencji umożliwienie wyznaczenia niezależnych fragmentów kodu dla jak największej liczby pętli. Tabele zbudowane są z siedmiu głównych kolumn. Pierwsza kolumna tabel przedstawia nazwę pętli poddanej analizie. Kolejne trzy kolumny przedstawiają odpowiednio liczbę wszystkich relacji, relacji opisujących zależności proste przed redukcją oraz po redukcji. Czwarta z głównych kolumn zawiera liczbę cyklicznych grafów dla relacji po redukcji. Kolejne dwie kolumny opisują wątki, pierwsza z nich przedstawia liczbę instrukcji tworzących początki, natomiast druga liczbę wątków o początkach w kolumnie poprzedniej. Ostatnia z głównych kolumn opisuje, które algorytmy zostały użyte do analizy. W powyżej opisanych tabelach zapis postaci ‘x / y / ... / z’ (gdzie x, y, z to liczby całkowite) oznacza, że obiekt zgodnie z opisem kolumny tabeli, w pierwszym przebiegu analizy pętli występuje x razy. W każdym kolejnym przebiegu (redukcja zewnętrznej pętli, zgodnie z alg. 3.9) ilość wystąpień obiektu równa jest od ‘y’ do wartości ‘z’ np. dla pierwszej pętli ‘BT_error.f2p_2.t’ z tab. B1 dla kolumny opisującej wszystkie relacje zapis postaci ‘110 / 67 / 33’ oznacza, że w pierwszym przebiegu analizy wyznaczono 110 relacji opisujących zależności. W kolejnych przebiegach wyznaczono 67 i 33 relacje. Analogiczna zasada dotyczy pozostałych kolumn.

Poniżej zaprezentowano badania eksperymentalne wybranych pętli ze zbioru NPB. W każdym z przypadków przedstawiono ciało pętli źródłowej, stanowiącej podstawę analizy zgodnie z zaproponowanymi algorytmami. Przedstawiono ciało pętli przygotowanej do zrównoleglenia. Dodatkowo pętle, względem których możliwe jest zrównoleglenie oznaczono znacznikiem 'par'. Wyniki przyśpieszenia w zależności od ilości wątków (1, 2, 4, 8, 16) oraz liczby iteracji (wartości górnych granic pętli 64, 128, 256, 512) pokazano w sposób tabelaryczny jak i graficzny (wykresy). Dla transformacji blokowania zbadano czas dla trzech wartości bloku B: 8, 16, 32. W niniejszym rozdziale umieszczone zostały wyniki w postaci wykresów przedstawiających czas wykonania pętli, przyśpieszenia oraz efektywności dla górnej granicy pętli równej 512. Wyniki w postaci tabelarycznej jak również wyniki dla pozostałych wielkości pętli (górne granice 64, 128, 256, 512) przedstawione zostały w załącznikach do pracy.

Badanie przeprowadzono w środowisku (załącznik A):

- maszyna równoległa : SGI Altix 3700, 128 x Itanium2 1.5GHz (NUMA), 256 GB RAM, HDD 1TB,
- kompilator : Intel C/C++ (ver. 9.1)

Czas obliczeń wyznaczono poprzez obliczenie różnicy miedzy czasem końca obliczeń równoległych (bezpośrednio po zamknięciu regionu równoległego), a czasem rozpoczęcia obliczeń (pobranie czasu bezpośrednio przed otwarciem regionu równoległego). Poniżej przedstawiono pseudokod obrazujący sposób pomiaru czasu, gdzie T oznacza mierzony czas:

```
gettimeofday( TLS , NULL )
   #pragma omp parallel private(...) shared(...) .....
   {
      .....
   }
gettimeofday( TLE , NULL )
T = TLE – TLS
```

Poprawność zastosowanych transformacji została zweryfikowana poprzez porównanie wyników wersji równoległej z wersją oryginalną (sekwencyjną). Wartości początkowe dla pętli oryginalnych zostały wygenerowane w sposób losowy. Następnie utworzono kopię zmiennych dla pętli równoległych. Po wykonaniu kodu sekwencyjnego jak i równoległego dane wyjściowe zostały porównane. We wszystkich badanych przypadkach wyniki dla pętli zrównoleglonych zgadzały się z wynikami dla pętli sekwencyjnych.

Postać równoległa pętli została zapisana zgodnie ze standardem OpenMP [75], [23]. Do przeprowadzenia testów wybrano harmonogramowanie statyczne (ang. *static*

schedule) z domyślną wielkością paczki (ang. *chunk size*). W takim przypadku iteracje dla pętli poddanej zrównolegleniu, zostaną podzielone pomiędzy istniejące wątki w sposób równomierny tuż przed wykonaniem pętli. Do każdego wątku przydzielona zostanie jedna paczka zawierająca $N = \frac{M}{W}$ iteracji, gdzie M określa ilość iteracji pętli, W oznacza liczbę wątków wykonujących pętlę. Jeśli liczba iteracji jest niepodzielna przez liczbę wątków to, zgodnie ze specyfikacją, podział i przydział pozostałych iteracji (reszta z dzielenia M/W) zależny jest od kompilatora. Harmonogramowanie statyczne, w odróżnieniu od dynamicznego (ang. *dynamic, guided*) pozwala na obniżenie kosztów związanych z zarządzaniem dynamicznym przydziałem paczek oraz pozwala na zwiększenie lokalności kodu. Koszt poniesiony na dynamiczne zarządzenie paczkami byłby uzasadniony w przypadku, gdy: ilość obliczeń w poszczególnych iteracjach jest różna, system jest obciążony innymi zadaniami lub ilość wątków jest większa od liczby procesorów. W przypadku przeprowadzanych testów moc obliczeniowa jednostek przetwarzających była w 100% przydzielona do określonego zadania, liczba procesorów była równa liczbie utworzonych wątków, zadania wykonywane w obrębie pojedynczego wątku wymagały bardzo zbliżonych mocy obliczeniowych. W związku z powyższym badania ujęte w dalszej części rozdziału przeprowadzone zostały zgodnie z harmonogramowanie statycznym o domyślnej wielkości paczki. Porównanie harmonogramowania w kontekście systemu, na którym przeprowadzono badania (SGI Altix 3700) przedstawione zostały w tab. 4.5

Tabela 4.5. Zestawianie typów harmonogramowania

Opis	**Statyczne**	**Dynamiczne**
lokalność	większa (+)	mniejsza (-)
koszt zarządzania paczkami iteracji (wyznaczanie i przydzielanie kolejnych paczek)	mniejszy (+)	większy (-)
brak wywłaszczania wątków	+	-
liczba wątków większa od liczby procesorów	-	+

Pętle poddane testom zgrupowane zostały w kolejnych podrozdziałach. Tytuł każdego podrozdziału jednoznacznie określa pętlę oraz jej umiejscowienie w zbiorze testowym NPB. Przykładowo tytuł podrozdziału „... BT_error_2” oznacza, iż pętla pochodzi z testu BT (patrz opis NPB, rozdział 4) i jest to druga z analizowanych pętli z

pliku o nazwie „error.f”. Do pomiaru czasu wykonania pętli użyta została funkcja gettimeofday(), umożliwiająca dokonanie pomiaru z dokładnością do mikrosekund.

4.5.1. Pętla BT_error_2

Dane wejściowe dla algorytmów stanowi kod źródłowy przedstawiony w tab. 4.5.1(a) [47]. Do analizy zgodnie z zaproponowanymi algorytmami uwzględniono jedynie zależności proste (ang. *flow*). Jak wiadomo zależności odwrotne (ang. *anti*) jak i po wyjściu (ang. *output*) można zawsze usunąć poprzez zastosowanie odpowiednich przekształceń.

W celu zmniejszenia liczby zależności oryginalna pętla została zmodyfikowana, poprzez zredukowanie odwołań do zmiennych skalarnych, do postaci przedstawionej w tab. 4.5.1(a). Dzięki przekształceniu pętli liczba zależności została zredukowana z 21 do 7 w pierwszym przebiegu algorytmu 3.9.

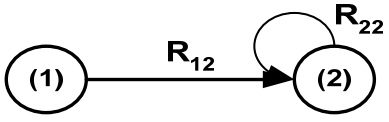
Przebieg analizy

Do analizy wykorzystane zostały algorytmy: 3.6, 3.7, 3.8, 3.9. W dwóch pierwszych przebiegach algorytmu 3.9 nie uzyskano niezależnych wątków, dopiero w trzecim przebiegu uzyskano niezależne wątki. W związku z tym, iż w każdym kolejnym przebiegu algorytmu 3.9 (oprócz pierwszego) redukowana jest zewnętrzna pętla, to w kolejnych trzech przebiegach uzyskano odpowiednio 7, 5, 3 zależności prostych. W trzecim przejściu uzyskano następujące zależności:

flow $1 \rightarrow 2$: $\{[i,n] \rightarrow [i,n] : 0 \le i \le N3 \;\&\&\; 1 \le n \le 5\}$

flow $1 \rightarrow 2$: $\{[i,n] \rightarrow [i',n] : 0 \le i < i' \le N3 \;\&\&\; 1 \le n \le 5\}$

flow $2 \rightarrow 2$: $\{[i,n] \rightarrow [i+1,n] : 0 \le i < N3 \;\&\&\; 1 \le n \le 5\}$



Dla powyższego grafu zależności, początki krańcowe znajdują się w instrukcji (2) w grafie cyklicznym wyznaczonym zgodnie z zależnością opisaną relacją R_{22}. Początki dla niezależnych wątków w instrukcji (2) opisuje poniższy zbiór:

$SSF\_2 = \{[0,n]: 1 \le n \le 5 \;\&\&\; 1 \le N3\}$.

Zbiór opisujący niezależne wątki przedstawia się nastepująco:

$\{[0,n,0,n]: 1 \le n \le 5 \;\&\&\; 1 \le N3\}$ union $\{[0,n,i,n]: 1 \le i \le N3 \;\&\&\; 1 \le n \le 5\}$.

Wynika z tego, iż możliwe jest zrównoleglenie pętli o zmiennej indeksowej ‘n’, przyjmującej wartości z zakresu n = <1,5>. Redukcja dwóch pierwszych pętli powoduje, iż należy wykonać je w sposób sekwencyjny. W każdej iteracji pętli o indeksie ‘j’ zostanie wykonanych 5 wątków zgodnie z kodem przedstawionym w tab. 4.5.1(b)

Tabela 4.5.1. Postać sekwencyjna i równoległa pętli BT_eror_2

```
for k=0 to N1 do
    for j=0 to N2 do
        for i=0 to N3 do
            for n=1 to 5 do
                u_exact(n) =  ce(n,1) + dble(i) * dnxm1*(ce(n,2) + dble(i) * dnxm1*(ce(n,5) +
dble(i) * dnxm1*(ce(n,8) + dble(i) * dnxm1*ce(n,11)))) + dble(j) * dnym1*(ce(n,3) + dble(j) *
dnym1*(ce(n,6) + dble(j) * dnym1*(ce(n,9) + dble(j) * dnym1*ce(n,12)))) + dble(k) *
dnzm1*(ce(n,4) + dble(k) * dnzm1*(ce(n,7) + dble(k) * dnzm1*(ce(n,10) +  dble(k) *
dnzm1*ce(n,13))))
                rms(n) = rms(n) + ( u(n,i,j,k) - u_exact(n) ) * ( u(n,i,j,k) - u_exact(n) )
            endfor
        endfor
    endfor
endfor
```

(a) Pętla poddana analizie

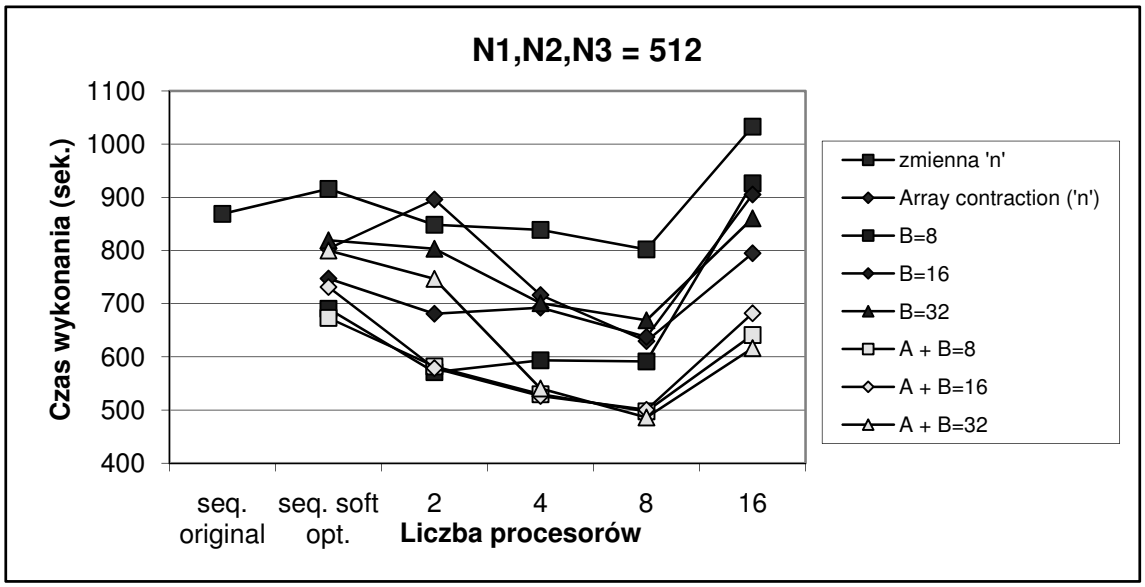
```
for(k=0;k≤N1;k++)     //sek
    for(j=0;j≤N2;j++)  //sek
    {
        for(i=0;i≤N3;i++)
        for(n=1;n≤5;n++)
            u_exact_1_(n,i) =  ce_1(n,1) + i * dnxm1*(ce_1(n,2) + i * dnxm1*(ce_1(n,5) + i *
dnxm1*(ce_1(n,8) + i * dnxm1*ce_1(n,11)))) + j * dnym1*(ce_1(n,3) + j * dnym1*(ce_1(n,6) +
j * dnym1*(ce_1(n,9) + j * dnym1*ce_1(n,12)))) + k * dnzm1*(ce_1(n,4) + k *
dnzm1*(ce_1(n,7) + k * dnzm1*(ce_1(n,10) +  k * dnzm1*ce_1(n,13))));

        for(n=1;n≤5;n++)          //par
            for(i=0;i≤N3;i++)
                rms_1(n) = rms_1(n) + ( u_1(n,i,j,k) - u_exact_1_(n,i) ) * ( u_1(n,i,j,k) -
u_exact_1_(n,i) );
    }
```

(b) Pętla gotowa do zrównoleglenia

Wyniki badań

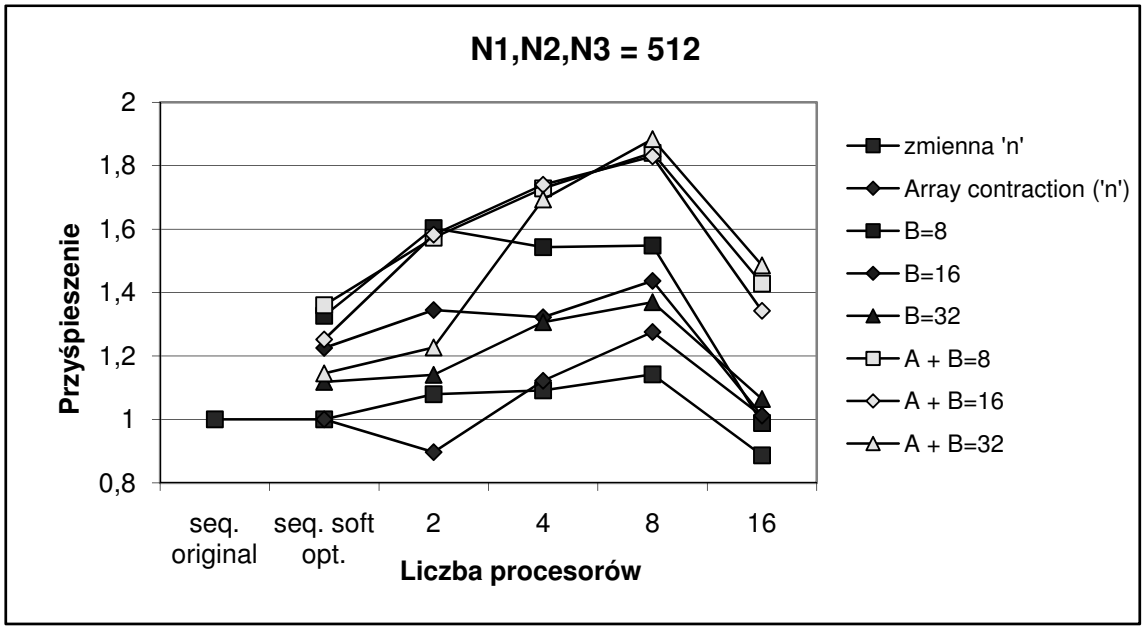
Testy przeprowadzono dla transformacji zgodnie z zaproponowanymi algorytmami jak również dla dwóch dodatkowych transformacji zwiększających lokalność kodu w obrębie pojedynczego wątku. Pierwsza z nich to skalaryzacja tablic, umożliwiająca wykorzystanie rejestrów procesora. Druga to kafelkowanie (ang. *blocking*) pozwalająca na zwiększenie liczby trafień do pamięci podręcznej procesora cache. Zbadano również czas wykonania pętli przy zastosowaniu dwóch powyższych transformacji jednocześnie. Wyniki pomiaru czasu przedstawiono na rys. 4.1.1. Najkrótszy czas wykonania uzyskano przy zastosowaniu ośmiu procesorów dla dwóch transformacji jednocześnie, gdzie dla kafalekowania wielkość bloku wynosi 32. Dla dwóch procesorów najkrótsze czasy uzyskano dla kafelkowania (B=8) oraz przy zastosowaniu dwóch transformacji jednocześnie (B=8,16). Dla czterech jak i szesnastu procesorów najkrótsze czasy uzyskano przy zastosowaniu dwóch transformacji jednocześnie, z tym że przy szesnastu procesorów czas wykonania pętli wydłużył się w stosunku do pomiarów dla mniejszej liczby procesorów. Wynik taki jest jak najbardziej wytłumaczalny gdyż wyznaczono pięć niezależnych wątków, w związku z tym wykonanie pętli na szesnastu procesorach nie mogło być opłacalne.



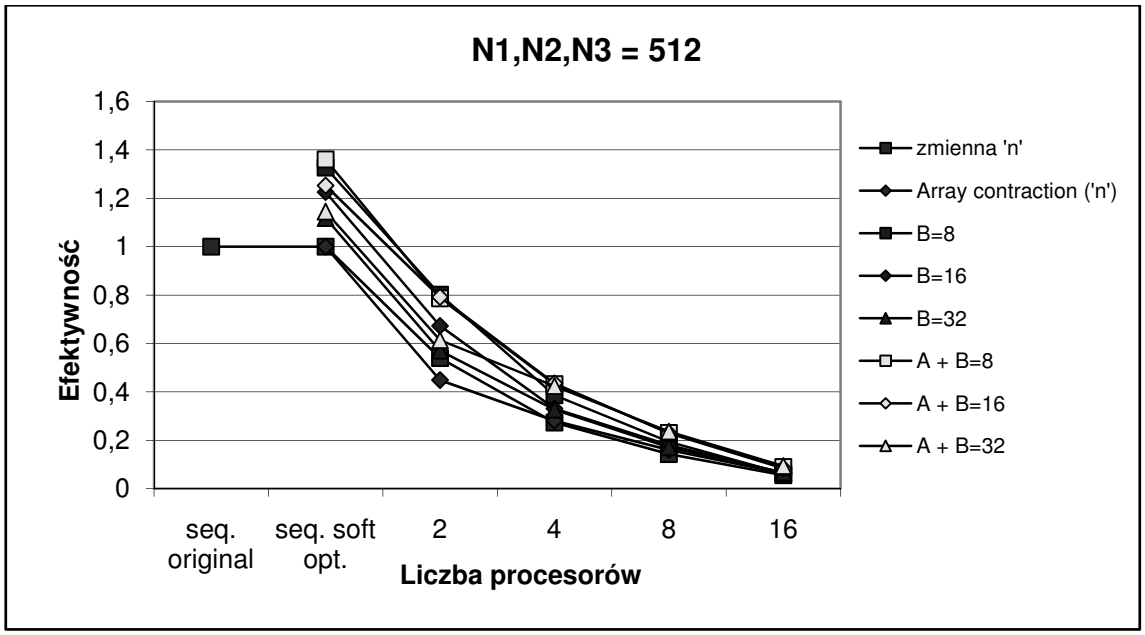
Rysunek 4.1.1. Czas wykonania pętli BT_error_2 [op. własne]

Jak można było przypuszczać na podstawie powyższych pomiarów największe przyśpieszenie uzyskano dla ośmiu procesorów przy zastosowaniu dwóch transformacji jednocześnie (patrz rys. 4.1.2). Największą efektywność wynoszącą około 1,4 uzyskano na jednym procesorze przy zastosowaniu sklaryzacji połączonej z kafelkowaniem (B=8). Wraz ze wzrostem liczby procesorów efektywność kodu spada. Najwyższą

efektywność przy założeniu, że liczba procsorów jest większa od jednego osiągnięto przy dwóch wątkach dla trzech transformacji: kafelkowanie (B=8), kafelkowanie (B=8 oraz B=16) wraz ze skalaryzacją zmiennych (patrz rys. 4.1.3).



Rysunek 4.1.2. Przyśpieszenie dla pętli BT_error_2 [op. własne]



Rysunek 4.1.3. Efektywność dla pętli pętla BT_error_2 [op. własne]

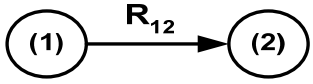
4.5.2. Pętla FT_auxfnct_2

Dane wejściowe do algorytmów stanowi oryginalny kod źródłowy zapisany w języku akceptowanym przez program Petit, tab. 4.6.1(a).

Przebieg analizy

Do analizy wykorzystane zostały algorytmy: 3.6, 3.7, 3.8, użycie algorytmu 3.9 nie było konieczne gdyż wyznaczono niezależne wątki bez konieczności redukcji pętli. Przy pomocy narzędzia Petit wyznaczono jedną zależność wewnątrz najbardziej wewnętrznej pętli miedzy instrukcjami (1) i (2):

flow $1 \rightarrow 2$: $\{[i,k,j] \rightarrow [i,k,j] : 1 \leq i \leq N1 \;\&\&\; 1 \leq k \leq N2 \;\&\&\; 1 \leq j \leq N3\}$



Na podstawie algorytmu 3.6 początki krańcowe dla niezależnych wątków znajdują w instrukcji (1), zbiór początków pokazano poniżej:

$SFF\_1 = \{[i,k,j,i,k,j]: 1 \leq i \leq N1 \;\&\&\; 1 \leq k \leq N2 \;\&\&\; 1 \leq j \leq N3\}$

Zgodnie z algorytmem 3.7 wyznaczono zbiory opisujące niezależne wątki o początku w instrukcji (1)

$[(1),(1)] : \{[i,k,j,i,k,j]: 1 \leq i \leq N1 \;\&\&\; 1 \leq k \leq N2 \;\&\&\; 1 \leq j \leq N3\}$

$[(1),(2)] : \{[i,k,j,i,k,j]: 1 \leq i \leq N1 \;\&\&\; 1 \leq k \leq N2 \;\&\&\; 1 \leq j \leq N3\}$

Zapis [(1),(1)] oznacza, iż zbiór opisuje instancje instrukcji (1) należące do wątku o początkach w (1), analogicznie zapis [(1),(2)] oznacza, że zbiór opisuje instancje instrukcji (2) należące do wątku o początkach w (1). Wygenerowanie kodu z zachowaniem porządku leksykograficznego dla powyższych zbiorów umożliwia stworzenie kodu przebierającego niezależne wątki, tab. 4.6.1(b). Do generacji kodu użyto funkcji ‘codegen’ Omega Calculator’a, gdzie dodatkowym parametrem dla każdego zbioru była transformacja tożsamościowa.

Liczba niezależnych wątków równa jest iloczynowi iteracji wszystkich pętli i wynosi:

$N3 \geq 1 \;\&\&\; N2 \geq 1 \Rightarrow N1*N2*N3.$

Tabela 4.6.1. Postać sekwencyjna i równoległa pętli FT_auxfnct_2

```
for i=1 to N1 do
    for k=1 to N2 do
        for j=1 to N3 do
            y(j,k,i)=y(j,k,i)*twiddle(j,k,i)  (1)
            x(j,k,i)=y(j,k,i)         (2)
        endfor
    endfor
endfor
```

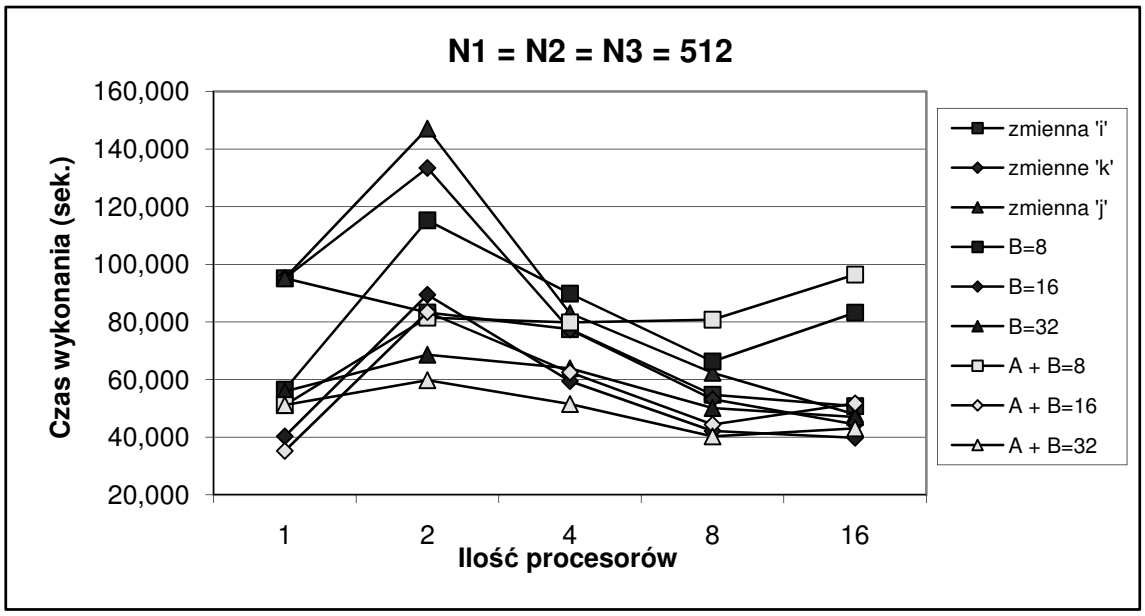
(a) Pętla poddana analizie

```
for(i=1;i≤N1;i++)       //par
    for(k=1;k≤N2;k++)           //par
        for(j=1;j≤N3;j++)          //par
        {
            y(j,k,i) = y(j,k,i) * twiddle(j,k,i);
            x(j,k,i) = y(j,k,i);
        }
```

(b) Pętla gotowa do zrównoleglenia

Wyniki badań

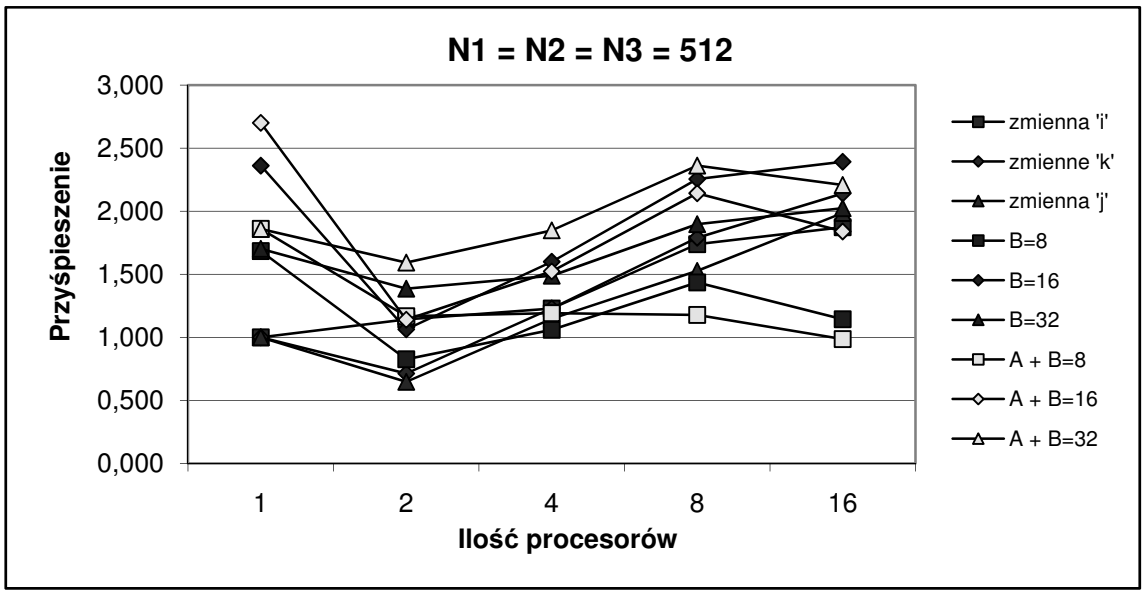
Analogicznie jak dla poprzedniego przykładu do zwiększenie lokalności kodu w obrębie pojedynczego wątku zastosowano skalaryzacje tablic oraz kafelkowanie. Pomiar czasu dokonano dla zrównoleglenia każdej z pętli (patrz tab. 4.6.1, pętle oznaczone //par) jak również względem pierwszej pętli (zmienna indeksowa 'i') przy zastosowaniu kafelkowania oraz skalaryzacji zmiennych (patrz rys. 4.2.1). Najkrótszy czas wykonania pętli uzyskano dla szesnastu procesorów wraz z kafelkowaniem, gdzie B=16. Najbardziej optymalną transformacją ze względu na czas wykonania pętli dla różnej liczby procesorów jest skalaryzacja tablic w połączeniu z kafelkowaniem (B=32). Jedynie dla szesnastu procesorów transformacja ta okazała się słabsza pod względem czasu od samego kafelkowania (B=16)



Rysunek 4.2.1. Czas wykonania pętli FT_auxfnct_2 [op. własne]

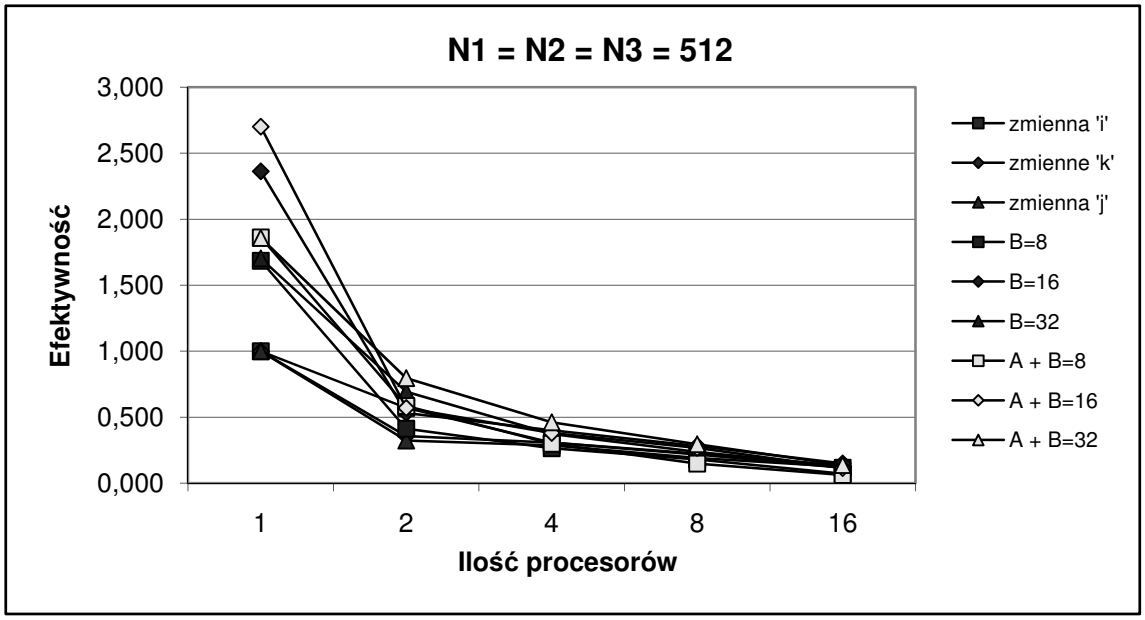
Uzyskane przyśpieszenie w zależności od zastosowanej transformacji przedstawione zostało na rys. 4.2.2. Najlepsze przyśpieszenie uzyskano dla skalaryzacji tablic wraz

z kafelkowaniem (B=32) przy ośmiu procesorach oraz dla kafelkowania (B=16) przy szesnastu procesorach. Ze względu na porównywalne czasy pomiarów a różną liczbę procesorów, efektywność dla ośmiu procesorów będzie wyższa niż dla szesnastu, co znajduje swoje potwierdzenie na rys. 4.2.3.



Rysunek 4.2.2. Przyśpieszenie dla pętli FT_auxfnct_2 [op. własne]

Najwyższą efektywność dla więcej niż jednego procesora uzyskano dla kafelkowania (B=32) połączonego ze skalaryzacją tablic oraz dla samego kafelkowania (B=32) (patrz rys. 4.2.3.). Wraz ze wzrostem liczby procesorów efektywność spada w sposób zbliżony do liniowego.



Rysunek 4.2.3. Efektywność dla pętli FT_auxfnct_2 [op. własne]

4.5.3. Pętla LU_HP_l2norm_2

Dane wejściowe do algorytmów stanowi oryginalny kod źródłowy zapisany w języku akceptowanym przez program Petit, tab. 4.7.1(a).

Przebieg analizy

Podczas analizy pętli wykorzystane zostały algorytmy 3.6, 3.7, 3.8, 3.9. Algorytm 3.9 został wykorzystany dwukrotnie w celu redukcji dwóch najbardziej zewnętrznych pętli. W pierwszym przebiegu przy pomocy narzędzia Petit wyznaczone zostały trzy zależności, co uniemożliwiło wyszukanie niezależnych wątków. W związku z powyższym zgodnie z algorytmem 3.9, przyjęto wykonanie najbardziej zewnętrznej pętli w sposób sekwencyjny (zwanej w dalszej części pracy redukcją pętli), dzięki czemu liczba zależności została zredukowana do dwóch. Zależności te utworzyły Zredukowany Graf Zależności (ang. *reduced dependency graph, RDG*) zawierający wspólne końce, przez co niemożliwe stało się wygenerowani niezależnych wątków na podstawie wyznaczonych zależności. W kolejnym kroku algorytmu 3.9 przestrzeń iteracji została zawężona do dwóch najbardziej wewnętrznych pętli, gdzie Petit wyznaczył jedną zależność:

flow $1 \rightarrow 1 : \{[i,m] \rightarrow [i+1,m] : N4 \leq i < N5 \;\&\&\; 1 \leq m \leq 5\}$.

Początki krańcowe dla niezależnych wątków opisuje następujący zbiór:

$SSF\_1 = \{[i,m] \rightarrow [i+1,m] : N4 \leq i < N5 \;\&\&\; 1 \leq m \leq 5\}$.

Zbiór opisujący niezależne wątki przedstawia się następująco:

$[(1),(1)] : \{[N4,m,N4,m]: 1 \leq m \leq 5 \;\&\&\; N4 < N5\} \text{ union } \{[N4,m,i,m]: N4 < i \leq N5 \;\&\&\; 1 \leq m \leq 5\}$.

Niezależne wątki przedstawione zostały w tab. 4.7.1(b). Ze względu na redukcje dwóch najbardziej zewnętrznych pętli, zgodnie z algorytmem 3.9, pętle te należy przebierać w sposób sekwencyjny . Wyznaczono pięć niezależnych wątków co jest równe górnej granicy pętli przebierającej po zmiennej indeksowej ‘m’.

Liczba niezależnych wątków równa jest ilości iteracji dla pętli o indeksie ‘m’ i wynosi:

$N4 \leq N5-1 \Rightarrow 5$

Tabela 4.7.1. Postać sekwencyjna i równoległa pętli LU_HP_l2norm_2

```
for k=2 to N1 do
    for j=N2 to N3 do
        for i=N4 to N5 do
            for m=1 to 5 do
                sum(m) = sum(m) + v(m,i,j,k) * v(m,i,j,k) (1)
            endfor
        endfor
    endfor
endfor
```

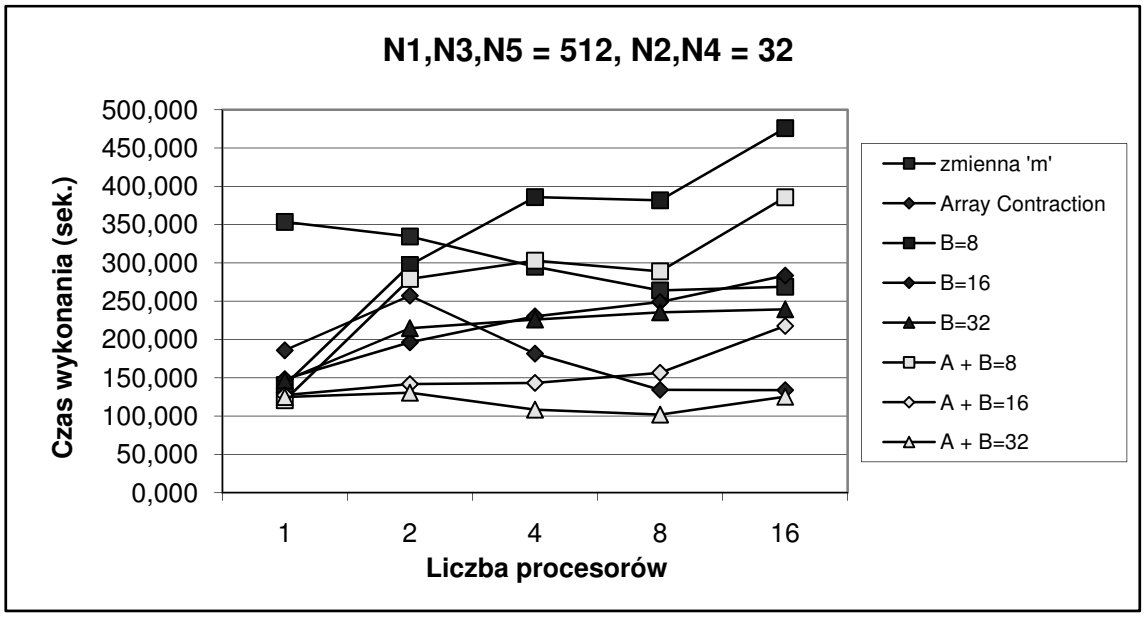
(a) Pętla poddana analizie

```
for(k=2;k≤N1;k++)    //seq
    for(j=N2;j≤N3;j++)          //seq
    {
        #pragma omp parallel for  private (m,i) shared(sum,v)
            for(m=1;m≤5;m++) //par
                for(i=N4;i≤N5;i++)         //seq
                    sum_1(m) = sum_1(m) + v_1(m,i,j,k) * v_1(m,i,j,k);
    }
```

(b) Pętla gotowa do zrównoleglenia

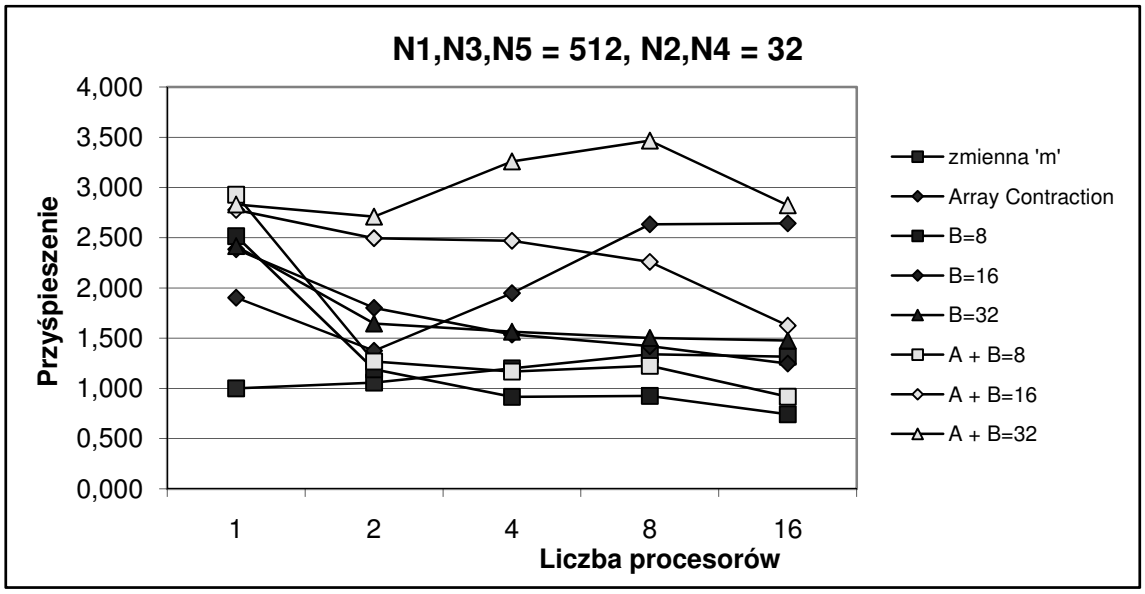
Wyniki badań

Podobnie jak dla pętli BT_error_2 tak i w tym przypadku liczba niezależnych wątków wynosi pięć, przy spełnieniu warunku N4 ≤ N5-1. Zwiększenie lokalności kodu w obrębie pojedynczego wątku pozwala na zredukowanie czasu wykonania pętli. Jednakże zgodnie z rys. 4.3.1, nie każda transformacja zwiększająca lokalność kodu powoduje redukcję czasu wykonania. Najlepsze wyniki zarówno pod względem czasu wykonania oraz przyśpieszenia uzyskano przy zastosowaniu skalaryzacji zmiennych wraz z kafelkowaniem (B=32). Wielkość bloku ma zasadnicze znaczenie, co potwierdzają uzyskane wyniki (patrz rys. 4.3.1). Dla B=8 czas wykonania pętli dla czterech, ośmiu oraz szesnastu procesorów jest najsłabszy ze wszystkich czasów, zarówno dla samego kafelkowania jak i kafelkowania wraz ze skalaryzacją tablic.

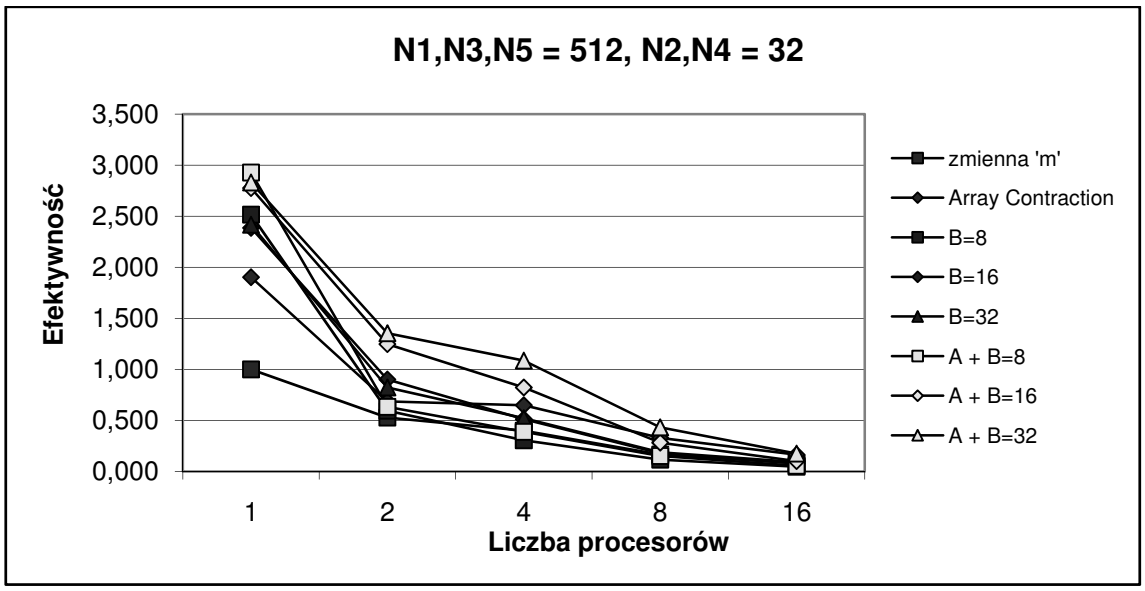


Rysunek 4.3.1. Czas wykonania pętli LU_HP_l2norm_2 [op. własne]

Maksymalne przyśpieszenie jaki uzyskano wynosi ~3,5 dla ośmiu procesorów przy zastosowaniu skalaryzacji tablic wraz z kafelkowaniem (B=32). Zwiększając lokalność danych poprzez zastosowanie jedynie skalaryzacji tablic uzyskujemy przyśpieszenie ~2,6 dla ośmiu procesorów. Świadczy to o tym, iż w przypadku powyższej pętli właśnie ta transformacja w obrębie pojedynczego wątku przynosi największą korzyść pod względem przyśpieszenia jak i efektywności kodu (patrz rys. 4.3.3). Okazuje się, iż wykonanie kodu na pojedynczym procesorze przy zwiększonej lokalności pozwala na osiągnięcie przyśpieszenia na poziomie 2,5 – 3,0 w stosunku do oryginalnego kodu. Wzrost przyśpieszenia jest tak duży gdyż sekwencyjne wykonanie kodu przy zwiększonej lokalności kodu (skalaryzacja zmiennych + kafelkowanie) odbywa się z zachowaniem wątków, z tym ze każdy wątek wykonywany jest na tym samym procesorze. Podobnie jak w poprzednich przykładach efektywność kodu wraz ze wzrostem liczby procesorów maleje. Ze względu na liczbę wątków wykonanie kodu na więcej niż pięciu procesorach praktycznie przestaje być opłacalne.



Rysunek 4.3.2. Przyśpieszenie dla pętli LU_HP_l2norm_2 [op. własne]



Rysunek 4.3.3. Efektywność dla pętli LU_HP_l2norm_2 [op. własne]

4.5.4. Pętla UA_adapt_1

Dane wejściowe do algorytmów stanowi oryginalny kod źródłowy zapisany w języku akceptowanym przez program Petit, tab. 4.8.1(a).

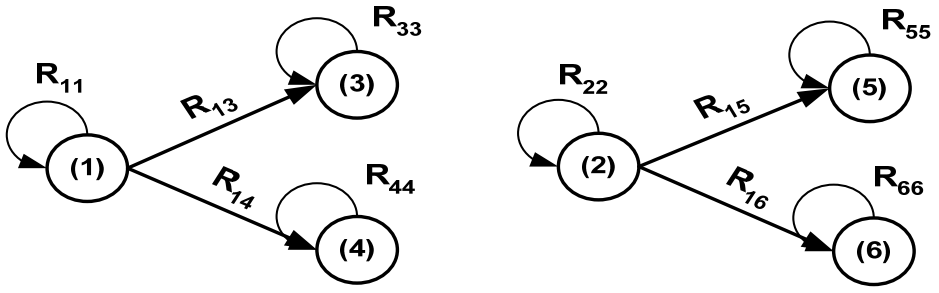
Przebieg analizy

W trakcie analizy wykorzystano algorytmy 3.6, 3.7, 3.8. W związku z tym, iż w pierwszym przebiegu wyznaczono niezalezne wątki, zawężanie przestrzeni iteracji

pętli w celu redukcji zależności nie było konieczne. Przy wykorzytsaniu narzędzia Petit wyznaczono dziesięć relacji opisujących zależności proste, lista relacji znajduje się ponizej.

flow 1 → 1 : {[i,kk,jj,ii] → [i,kk+1,jj,ii] : 1 ≤ i ≤ N1 && 1 ≤ kk < N2 && 1 ≤ jj ≤ N3 && 1 ≤ ii ≤ N4}
flow 1 → 3 : {[i,kk,jj,ii] → [i,jj,jj',ii] : 1 ≤ jj ≤ N3, N5 && 1 ≤ ii ≤ N4, N7 && 1 ≤ i ≤ N1 && 1 ≤ kk ≤ N2 && 1 ≤ jj' ≤ N6}
flow 1 → 4 : {[i,kk,jj,ii] → [i,jj,jj',ii] : 1 ≤ jj ≤ N3, N5 && 1 ≤ ii ≤ N4, N7 && 1 ≤ i ≤ N1 && 1 ≤ kk ≤ N2 && 1 ≤ jj' ≤ N6}
flow 2 → 2 : {[i,kk,jj,ii] → [i,kk+1,jj,ii] : 1 ≤ i ≤ N1 && 1 ≤ kk < N2 && 1 ≤ jj ≤ N3 && 1 ≤ ii ≤ N4}
flow 2 → 5 : {[i,kk,jj,ii] → [i,jj,jj',ii] : 1 ≤ jj ≤ N3, N5 && 1 ≤ ii ≤ N4, N7 && 1 ≤ i ≤ N1 && 1 ≤ kk ≤ N2 && 1 ≤ jj' ≤ N6}
flow 2 → 6 : {[i,kk,jj,ii] → [i,jj,jj',ii] : 1 ≤ jj ≤ N3, N5 && 1 ≤ ii ≤ N4, N7 && 1 ≤ i ≤ N1 && 1 ≤ kk ≤ N2 && 1 ≤ jj' ≤ N6}
flow 3 → 3 : {[i,kk,jj,ii] → [i,kk+1,jj,ii] : 1 ≤ i ≤ N1 && 1 ≤ kk < N5 && 1 ≤ jj ≤ N6 && 1 ≤ ii ≤ N7}
flow 4 → 4 : {[i,kk,jj,ii] → [i,kk+1,jj,ii] : 1 ≤ i ≤ N1 && 1 ≤ kk < N5 && 1 ≤ jj ≤ N6 && 1 ≤ ii ≤ N7}
flow 5 → 5 : {[i,kk,jj,ii] → [i,kk+1,jj,ii] : 1 ≤ i ≤ N1 && 1 ≤ kk < N5 && 1 ≤ jj ≤ N6 && 1 ≤ ii ≤ N7}
flow 6 → 6 : {[i,kk,jj,ii] → [i,kk+1,jj,ii] : 1 ≤ i ≤ N1 && 1 ≤ kk < N5 && 1 ≤ jj ≤ N6 && 1 ≤ ii ≤ N7}

Graf zależności wygenerowany na podstawie uzyskanych relacji przedstawiono poniżej. Każda instrukcja w pętli, zgodnie z algorytmem 3.8, tworzy cykliczny graf (ang. *strongly conncted component - SCC*). Niezależnie dla każdego cyklicznego grafu wyznaczone są początki wątków (algorytm 3.6) oraz kod przebierający wątki (algorytm 3.7). Zgodnie z poniższym grafem wykonanie wątków przebierających instancje instrukcji (3), (4), (5), (6) muszą być poprzedzone wykonaniem instancji instrukcji (1) i (2) tworzących niezależne wątki. Kolejność ta wynika z konieczności honorowania zależności R_{13}, R_{14} oraz R_{25}, R_{26}.



Początki krańcowe:

SSF_1 = {[i,1,jj,ii]: 1 ≤ i ≤ N1 && 1 ≤ ii ≤ N4 && 1 ≤ jj ≤ N3 && 2 ≤ N2}
SSF_2 = {[i,1,jj,ii]: 1 ≤ i ≤ N1 && 1 ≤ ii ≤ N4 && 1 ≤ jj ≤ N3 && 2 ≤ N2}
SSF_3 = {[i,1,jj,ii]: 1 ≤ i ≤ N1 && 1 ≤ ii ≤ N7 && 1 ≤ jj ≤ N6 && 2 ≤ N5}
SSF_4 = {[i,1,jj,ii]: 1 ≤ i ≤ N1 && 1 ≤ ii ≤ N7 && 1 ≤ jj ≤ N6 && 2 ≤ N5}
SSF_5 = {[i,1,jj,ii]: 1 ≤ i ≤ N1 && 1 ≤ ii ≤ N7 && 1 ≤ jj ≤ N6 && 2 ≤ N5}
SSF_6 = {[i,1,jj,ii]: 1 ≤ i ≤ N1 && 1 ≤ ii ≤ N7 && 1 ≤ jj ≤ N6 && 2 ≤ N5}

Niezależne wątki jako zbiory:

[(1),(1)] : {[i,1,jj,ii,i,1,jj,ii]: 1 ≤ i ≤ N1 && 1 ≤ ii ≤ N4 && 1 ≤ jj ≤ N3 && 2 ≤ N2} union
{[i,1,jj,ii,i,kk,jj,ii]: 1 ≤ i ≤ N1 && 2 ≤ kk ≤ N2 && 1 ≤ jj ≤ N3 && 1 ≤ ii ≤ N4}

[(2),(2)] : {[i,1,jj,ii,i,1,jj,ii]: 1 ≤ i ≤ N1 && 1 ≤ ii ≤ N4 && 1 ≤ jj ≤ N3 && 2 ≤ N2} union
{[i,1,jj,ii,i,kk,jj,ii]: 1 ≤ i ≤ N1 && 2 ≤ kk ≤ N2 && 1 ≤ jj ≤ N3 && 1 ≤ ii ≤ N4}

[(3),(3)] : {[i,1,jj,ii,i,1,jj,ii]: 1 ≤ i ≤ N1 && 1 ≤ ii ≤ N7 && 1 ≤ jj ≤ N6 && 2 ≤ N5} union
{[i,1,jj,ii,i,kk,jj,ii]: 1 ≤ i ≤ N1 && 2 ≤ kk ≤ N5 && 1 ≤ jj ≤ N6 && 1 ≤ ii ≤ N7}

[(4),(4)] : {[i,1,jj,ii,i,1,jj,ii]: 1 ≤ i ≤ N1 && 1 ≤ ii ≤ N7 && 1 ≤ jj ≤ N6 && 2 ≤ N5} union
{[i,1,jj,ii,i,kk,jj,ii]: 1 ≤ i ≤ N1 && 2 ≤ kk ≤ N5 && 1 ≤ jj ≤ N6 && 1 ≤ ii ≤ N7}

[(5),(5)] : {[i,1,jj,ii,i,1,jj,ii]: 1 ≤ i ≤ N1 && 1 ≤ ii ≤ N7 && 1 ≤ jj ≤ N6 && 2 ≤ N5} union
{[i,1,jj,ii,i,kk,jj,ii]: 1 ≤ i ≤ N1 && 2 ≤ kk ≤ N5 && 1 ≤ jj ≤ N6 && 1 ≤ ii ≤ N7}

[(6),(6)] : {[i,1,jj,ii,i,1,jj,ii]: 1 ≤ i ≤ N1 && 1 ≤ ii ≤ N7 && 1 ≤ jj ≤ N6 && 2 ≤ N5} union
{[i,1,jj,ii,i,kk,jj,ii]: 1 ≤ i ≤ N1 && 2 ≤ kk ≤ N5 && 1 ≤ jj ≤ N6 && 1 ≤ ii ≤ N7}

Zależności wyznaczone dla analizowanej pętli tworzą dwa niezależne grafy zależności, gdzie w każdym grafie występują po trzy cykliczne grafy. Wykonanie instrukcji pętli zgodnie z porządkiem wynikającym z grafów, czyli watki w każdym z cyklicznych grafów równolegle natomiast przejścia miedzy grafami sekwencyjnie, pozwoli zbadać przyspieszenie i efektywność drobno-ziarnistej równoległości. W celu zmniejszenia ziarnistości kodu możliwe jest połączenie grafów i wykonanie instrukcji (1) i (2) w jednej pętli, a pozostałych instrukcji w drugiej pętli, patrz tab. 4.8.1(b). Wyniki przedstawiono i omówiono w kolejnym podrozdziale.

Liczba niezależnych wątków dla instrukcji (1) i (2) jest równa iloczynowi iteracji pętli o indeksach ‘i’, ‘jj’, ‘ii’ i jest równa (w związku z tym iż rozpatrujemy każdą instrukcje oddzielnie, poniższą wartość należy podwoić):

N1 * N6 * N7.

Pozostałe instancje instrukcje (3 - 6) tworzą cztery zbiory niezależnych wątków, gdzie w każdym zbiorze liczba niezależnych wątków jest równa iloczynowi iteracji pętli o indeksach ‘i’, ‘jj’, ‘ii’ i wynosi:

N1 * N6 * N7.

Ogólna liczba niezależnych wątków wynosi:

2 * (N1 * N6 * N7) + 4 * (N1 * N6 * N7)

z tym że wątki dla instancji instrukcji (3) i (4) musza być wykonane po wątkach dla instrukcji (1), oraz wątki (5) i (6) po wykonaniu wątków utworzonych dla instancji instrukcji (2).

Tabela 4.8.1. Postać sekwencyjna i równoległa pętli UA_adapt_1

```
for i=1 to N1 do
    for kk=1 to N2 do
        for jj=1 to N3 do
            for ii=1 to N4 do
                yone(ii,jj,i,1) = yone(ii,jj,i,1) +ixmc1(ii,kk)*x(kk,jj,i)          (1)
                yone(ii,jj,i,2) = yone(ii,jj,i,2) +ixmc2(ii,kk)*x(kk,jj,i)          (2)
            endfor
        endfor
    endfor
    for kk=1 to N5 do
        for jj=1 to N6 do
            for ii=1 to N7 do
                ytwo(ii,i,jj,1) = ytwo(ii,i,jj,1) + yone(ii,kk,i,1)*ixtmc1(kk,jj)  (3)
                ytwo(ii,i,jj,2) = ytwo(ii,i,jj,2) + yone(ii,kk,i,1)*ixtmc2(kk,jj)  (4)
                ytwo(ii,i,jj,3) = ytwo(ii,i,jj,3) + yone(ii,kk,i,2)*ixtmc1(kk,jj)  (5)
                ytwo(ii,i,jj,4) = ytwo(ii,i,jj,4) + yone(ii,kk,i,2)*ixtmc2(kk,jj)  (6)
            endfor
        endfor
    endfor
endfor
```

(a) Pętla poddana analizie

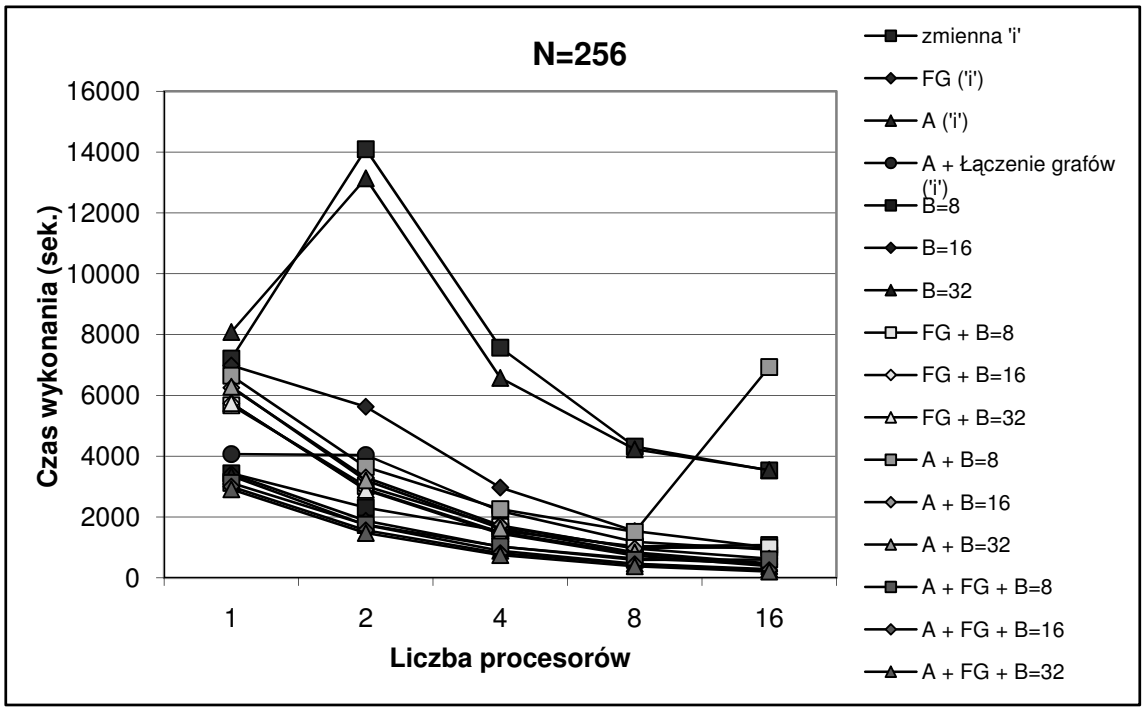
```
for(i=1;i≤N1;i++)       //par
    for(jj=1;jj≤N3;jj++)          //par
        for(ii=1;ii≤N4;ii++)      //par
            for(kk=1;kk≤N2;kk++)        //seq
            {
                yone(ii,jj,i,1) = yone(ii,jj,i,1) + ixmc1(ii,kk) * x(kk,jj,i);
                yone(ii,jj,i,2) = yone(ii,jj,i,2) + ixmc2(ii,kk) * x(kk,jj,i);
            }
for(i=1;i≤N1;i++)       //par
    for(jj=1;jj≤N6;jj++)          //par
        for(ii=1;ii≤N7;ii++)      //par
            for(kk=1;kk≤N5;kk++)        //seq
            {
                ytwo(ii,i,jj,1) = ytwo(ii,i,jj,1) + yone(ii,kk,i,1) * ixtmc1(kk,jj);
                ytwo(ii,i,jj,2) = ytwo(ii,i,jj,2) + yone(ii,kk,i,1) * ixtmc2(kk,jj);
                ytwo(ii,i,jj,3) = ytwo(ii,i,jj,3) + yone(ii,kk,i,2) * ixtmc1(kk,jj);
                ytwo(ii,i,jj,4) = ytwo(ii,i,jj,4) + yone(ii,kk,i,2) * ixtmc2(kk,jj);
            }
```

(b) Pętla gotowa do zrównoleglenia

Wyniki badań

Ze względu na skomplikowaną budowę grafu zależności (patrz rys. 4.4) dla powyższej pętli wykonano najwięcej testów ze wszystkich pętli. Badanie czasu wykonania pętli zmierzono dla analogicznego zestawu transformacji do wcześniejszych pętli. Dodatkowo wykonano testy dla nowej transformacji, łączenia grafów (ang. *fusing graphs – FG*). Zgodnie z rys. 4.4 wykonanie wątków zawierających instrukcje (3), (4) oraz (5), (6) możliwe jest po wykonaniu wątków zawierających instrukcje odpowiednio (1) oraz (2). Ze względu na brak zależności pomiędzy dwoma grafami SCC poprzez łączenie instrukcji w celu zmniejszenia granulacji kodu jest możliwe. W przeprowadzonych badaniach wykonanie instancji instrukcji (1) i (2) zawarto w jednym wątku, natomiast wykonanie instancji instrukcji (3), (4), (5) oraz (6) w oddzielnym wątku (patrz tab. 4.4b).

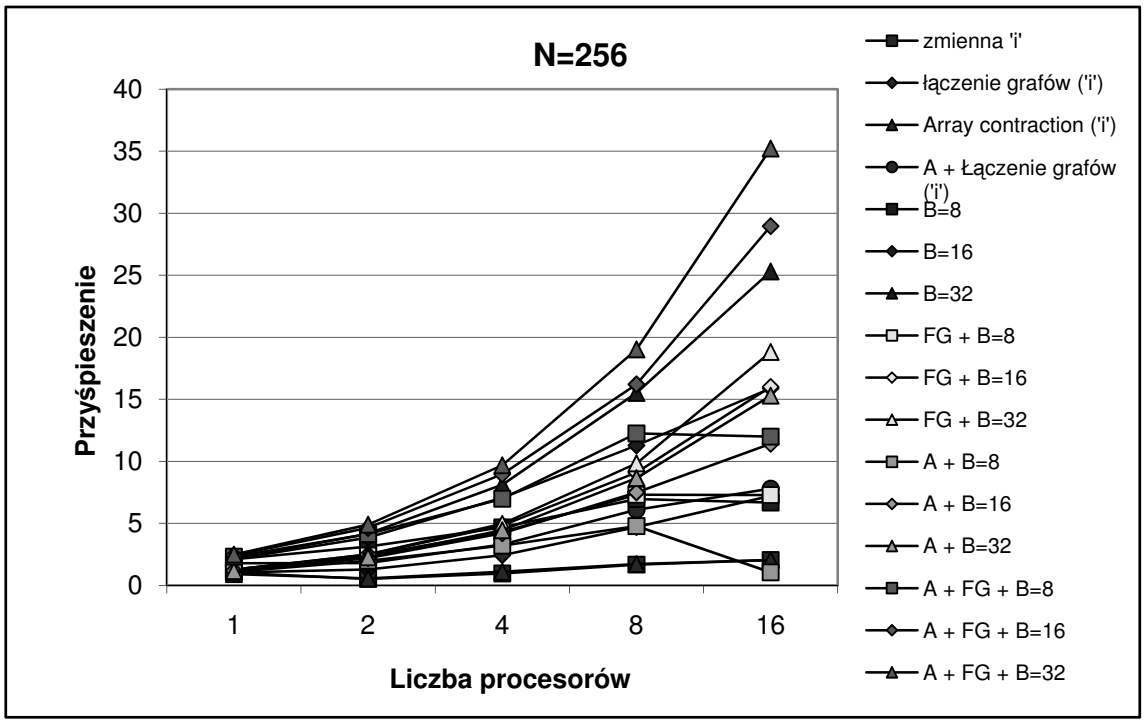
Zwiększenie lokalności kodu pozwoliło na znaczne zredukowanie czasu wykonania pętli w stosunku do oryginalnej wersji (rys. 4.4.1). Zmniejszenie granulacji kodu pozwoliło na dalszą redukcję czasu wykonania pętli. Biorąc pod uwagę czas wykonanie pętli na jednym procesorze przy zastosowaniu łączenia grafów porównywalne jest z wykonaniem kodu na dwóch procesorach bez łączenia grafów.



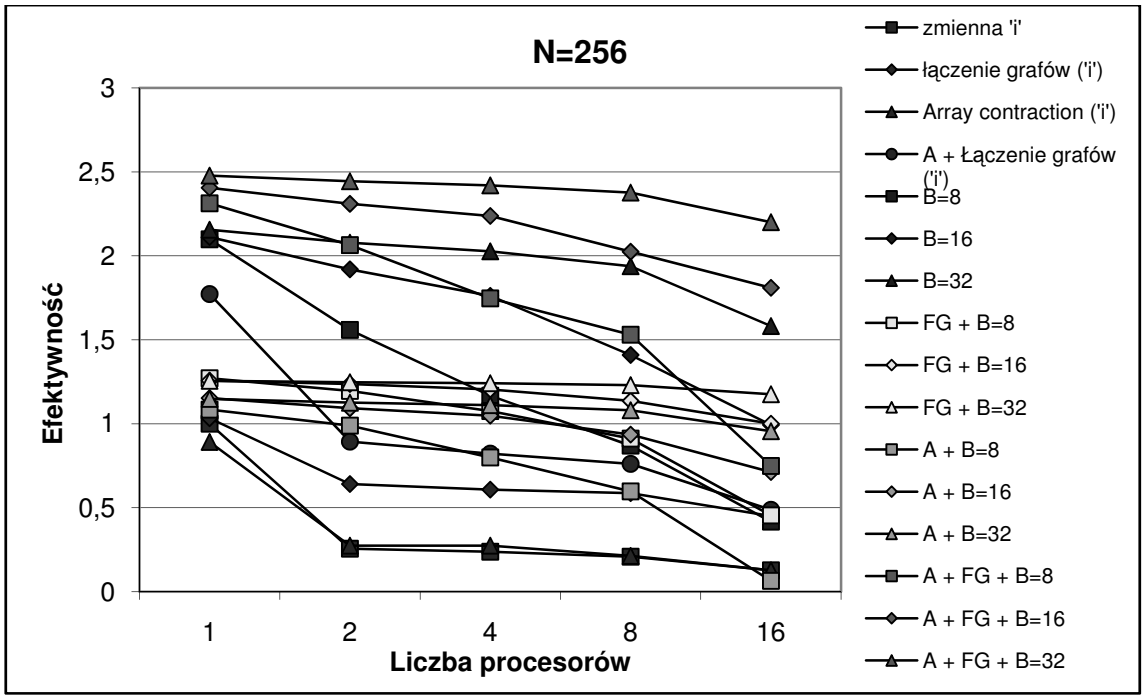
Rysunek 4.4.1. Czas wykonania pętli UA_adapt_1 [op. własne]

Dla dowolnej liczby procesorów z badanego zakresu możliwe było uzyskanie ponad-liniowego przyśpieszenia (rys. 4.4.2). Najwyższe przyśpieszenie uzyskano przy

zastosowaniu trzech transformacji jednocześnie tzn. skalaryzacji zmiennych, kafelkowania (B=32) oraz łączenia grafów (FG). W każdym z badanych przypadków przyśpieszenie było wieksze od liczby zastosowanych procesorów. Ma to swoje przełożenie w efektywności kodu, która za każdym razem przekroczyła wartość dwóch (rys. 4.4.3). W przypadku jednego, dwóch i czterech procesorów efektywność oscylowała około 2,5, natomiast dla ośmiu i szesnastu procesorów nieznacznie spadła odpowiednio do 2,3 oraz 2,2. Tak nieznaczny spadek efektywności kodu świadczy o dużej skalowalności zaproponowanego rozwiązania. Przypuszczalnie dla większej liczby procesorów spadek efektywności byłby większy, jednakże dla 32 procesorów efektywność kodu nie powinna spaść poniżej jedności. Ze względu na ograniczenia maszyny testowej, niemożliwe było wykonanie badań dla większej liczby procesorów.



Rysunek 4.4.2. Przyśpieszenie dla pętli UA_adapt_1 [op. własne]



Rysunek 4.4.3. Efektywność dla pętli UA_adapt_1 [op. własne]

4.5.5. Pętla UA_diffuse_3

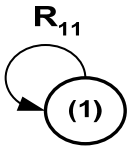
Dane wejściowe do algorytmów stanowi oryginalny kod źródłowy zapisany w języku akceptowanym przez program Petit, tab. 4.9.1(a).

Przebieg analizy

Podczas analizy wykorzystano algorytmy 3.6, 3.7, 3.8. Redukcja przestrzeni pętli nie była konieczna gdyż już w pierwszym przebiegu wyznaczono niezależne wątki. Przy pomocy narzędzia Petit wyznaczono jedną relację opisującą zależność prostą:

flow $1 \rightarrow 1 : \{[iz,k,j,i] \rightarrow [iz,k+1,j,i] : 1 \leq iz \leq N1 \;\&\&\; 1 \leq k < N2 \;\&\&\; 1 \leq j \leq N3 \;\&\&\; 1 \leq i \leq N4\}$.

Na podstawie relacji został wygenerowany graf zależności przedstawiony poniżej. Wyznaczony graf jest grafem cyklicznym, czyli początek i koniec relacji znajduje się w różnych instancjach tej samej instrukcji.



Na podstawie algorytmu 3.6 początki krańcowe dla niezależnych wątków znajdują w instrukcji (1), zbiór początków pokazano poniżej:

SSF_1 = $\{[iz,1,j,i]: 1 \leq iz \leq N1 \;\&\&\; 1 \leq i \leq N4 \;\&\&\; 1 \leq j \leq N3 \;\&\&\; 2 \leq N2\}$

Zgodnie z algorytmem 3.7 wyznaczono zbiór opisujący niezależne wątki, przedstawiony poniżej. Wygenerowanie kodu z zachowaniem porządku leksykograficznego dla poniższego zbioru umożliwia stworzenie kodu przebierającego niezależne wątki, tab. 4.9.1(b)

[(1),(1)] : {[iz,1,j,i,iz,1,j,i]: 1 ≤ iz ≤ N1 && 1 ≤ i ≤ N4 && 1 ≤ j ≤ N3 && 2 ≤ N2} union
{[iz,1,j,i,iz,k,j,i]: 1 ≤ iz ≤ N1 && 2 ≤ k ≤ N2 && 1 ≤ j ≤ N3 && 1 ≤ i ≤ N4}.

Na podstawie powyższego zbioru możemy oszacować liczbę niezależnych wątków i jest ona równa iloczynowi iteracji pętli o indeksach iz, j, i, czyli:

N4 ≥ 1 && N3 ≥ 1 && N2 ≥ 2 => N1*N3*N4.

Tabela 4.9.1. Postać sekwencyjna i równoległa pętli UA_diffuse_3

```
for iz=1 to N1 do
    for k=1 to N2 do
        for j=1 to N3 do
            for i=1 to N4 do
                tm1(i,j,iz) = tm1(i,j,iz)+wdtdr(i,k)*u(k,j,iz) (1)
            endfor
        endfor
    endfor
endfor
```

(a) Pętla poddana analizie

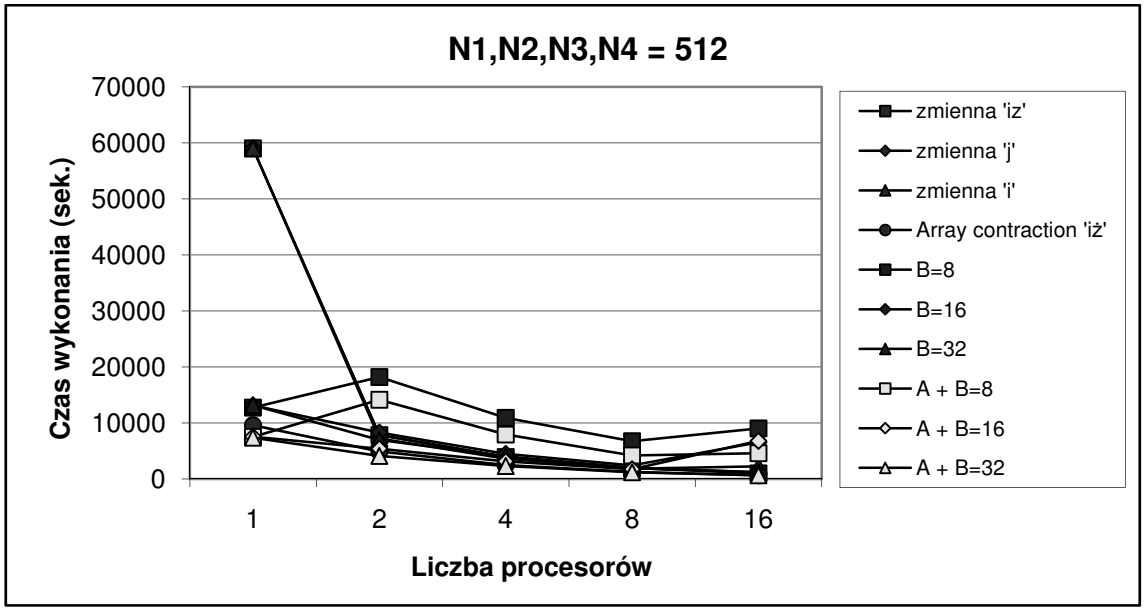
```
for(iz=1;iz≤N1;iz++)  //par
    for(j=1;j≤N3;j++)  //par
        for(i=1;i≤N4;i++)        //par
            for(k=1;k≤N2;k++)  //seq
                tm1_1(i,j,iz) = tm1_1(i,j,iz) + wdtdr_1(i,k) * u_1(k,j,iz);
```

(b) Pętla gotowa do zrównoleglenia

Wyniki badań

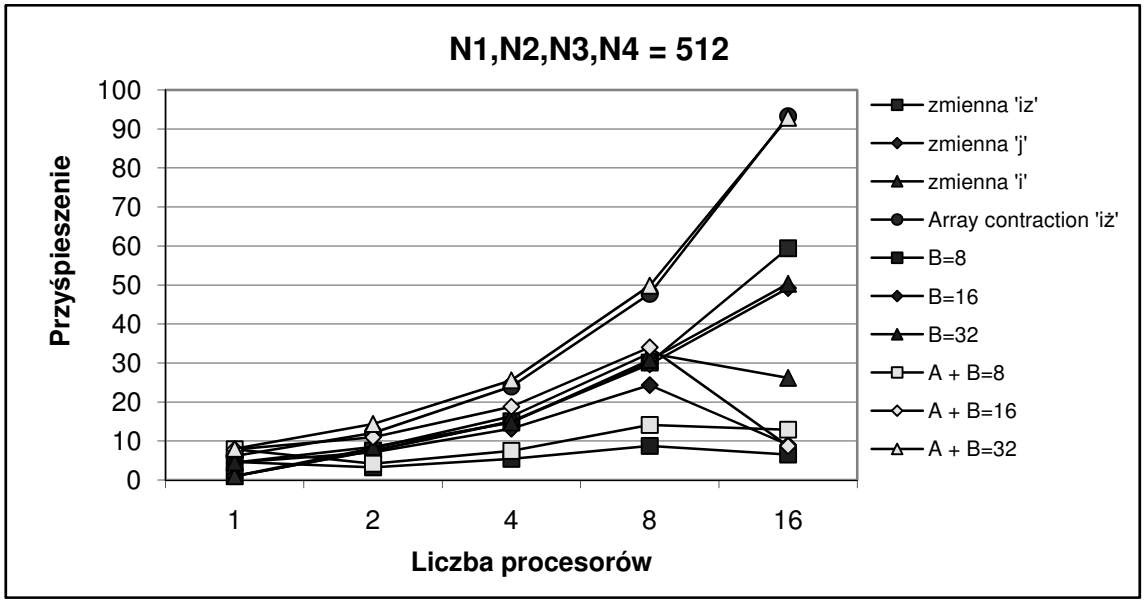
W celu zwiększenia czytelności wykresu (rys. 4.5.1) skala czasu została zredukowana do górnej granicy równej 2000, dzięki takiemu zabiegowi możliwe jest czytelne przedstawienie różnic w czasie wykonania poszczególnych transformacji. Najkrótszy czas wykonania uzyskano dla szesnastu wątków dla dwóch transformacji: skalaryzacji zmiennych jak również dla skalaryzacji zmiennych połączonej z kafelkowanie (B=32). Z badań jednoznacznie wynika, iż prawidłowo dobrana wielkość bloku (B) wpływa w znacznym stopniu na zmniejszenie czasu wykonania kodu. Dla dwóch wartości bloku (B=8,16) wzrost przyśpieszenia uzyskujemy jedynie

do ośmiu procesorów. Dalszy wzrost procesorów nie powoduje zmniejszenie czasu wykonania pętli - wręcz przeciwnie, czas zaczyna rosnąć.



Rysunek 4.5.1. Czas wykonania pętli UA_diffuse_3 [op. własne]

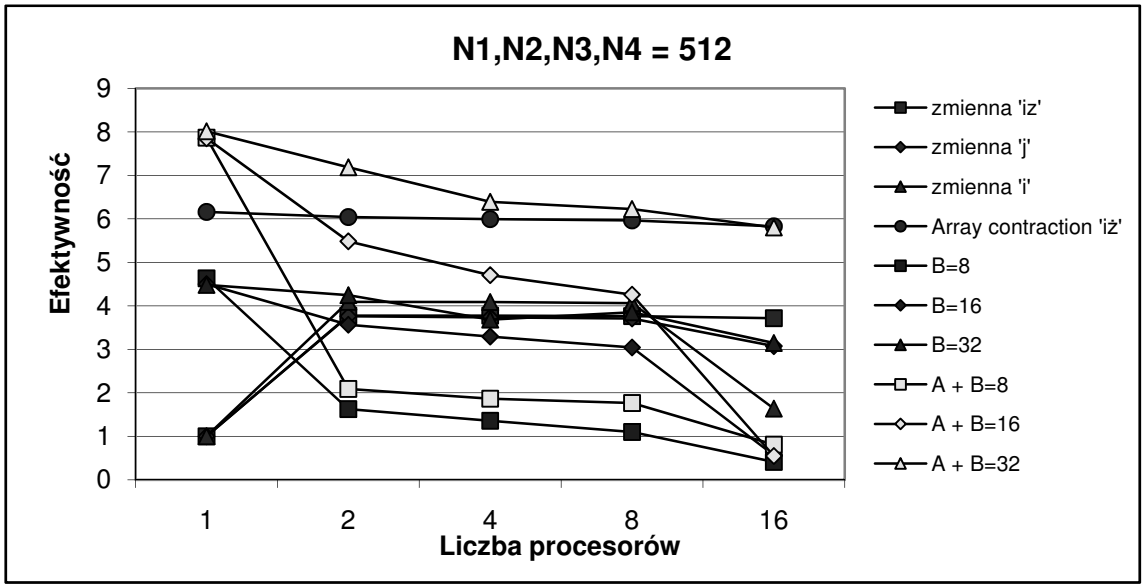
Maksymalne przyśpieszenie uzyskano na szesnastu procesorach i wyniosło powyżej 90 (patrz rys. 4.5.2). Podobnie jak dla pętli „UA_adapt_1" świadczy to o dużym potencjale kodu pod względem skalowalności.



Rysunek 4.5.2. Przyśpieszenie dla pętli UA_diffuse_3 [op. własne]

Efektywność kodu dla wspomnianych wcześniej dwóch najszybszych transformacji na 16 procesorach wynosi około sześciu. Biorąc pod uwagę najwyższą efektywność (czyli dla jednego procesora, patrz rys. 4.5.3), spadek efektywności dla szesnastu

procesorów jest dużo niższy w odniesieniu do poprzednich przykładów. Tak dobre pomiary pod względem czasu wykonania oraz efektywności wynikają z prawidłowego dobrania wielkości bloku (B) dla transformacji kafelkowania.



Rysunek 4.5.3. Efektywność dla pętli UA_diffuse_3 [op. własne]

4.5.6. Pętla UA_precond_4

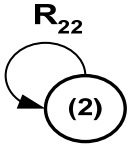
Dane wejściowe do algorytmów stanowi oryginalny kod źródłowy zapisany w języku akceptowanym przez program Petit, tab. 4.10.1(a).

Przebieg analizy

Podczas analizy wykorzystane zostały algorytmy 3.6, 3.7, 3.8. W pierwszym przebiegu wyznaczono niezależne wątki w związku, z czym nie było konieczności zawężania przestrzeni iteracji pętli. Podczas analizy pętli Petit wyznaczył jedną zależność prostą:

flow $2 \rightarrow 2 : \{[col,j,i] \rightarrow [col,j,i+1] : 1 \leq col \leq N1 \;\&\&\; 2 \leq j \leq N2 \;\&\&\; 1 \leq i < N3\}$

Graf zależności wygenerowany na podstawie uzyskanej relacji przedstawiono poniżej.



Początki krańcowe dla niezależnych wątków wyznaczone na podstawie algorytmu 3.6 znajdują się w instrukcji (2):

$SSF\_2 = \{[col,j,1]: 1 \leq col \leq N1 \;\&\&\; 2 \leq j \leq N2 \;\&\&\; 2 \leq N3\}$

Na podstawie algorytmu 3.7 uzyskano zbiór krotek opisujących niezależne wątki o początkach w instancjach instrukcji (2). Zgodnie z uzyskanym zbiorem [(2),(2)] możliwe jest zrównoleglenie pętli o dwóch zmiennych indeksowych ‘col’ oraz ‘j’.

[(2),(2)] : {[col,j,1,col,j,1]: 1 ≤ col ≤ N1 && 2 ≤ j ≤ N2 && 2 ≤ N3} union {[col,j,1,col,j,i]: 1 ≤ col ≤ N1 && 2 ≤ j ≤ N2 && 2 ≤ i ≤ N3}

Liczba niezależnych wątków równa jest iloczynowi iteracji dwóch pierwszych pętli i wynosi:

N3 ≥ 2 && N2 ≥ 2 => N1*(N2-1)

Tabela 4.10.1. Postać sekwencyjna i równoległa pętli UA_precond_4

```
for col=1 to N1 do
    tcpre(col,1)=tmp(col,1)     (1)
    for j=2 to N2 do
        for i=1 to N3 do
            tcpre(col,j) = tcpre(col,j) + qbnew(j-1,i,1)* tmp(col,i) (2)
        endfor
    endfor
endfor
```

(a) Pętla poddana analizie

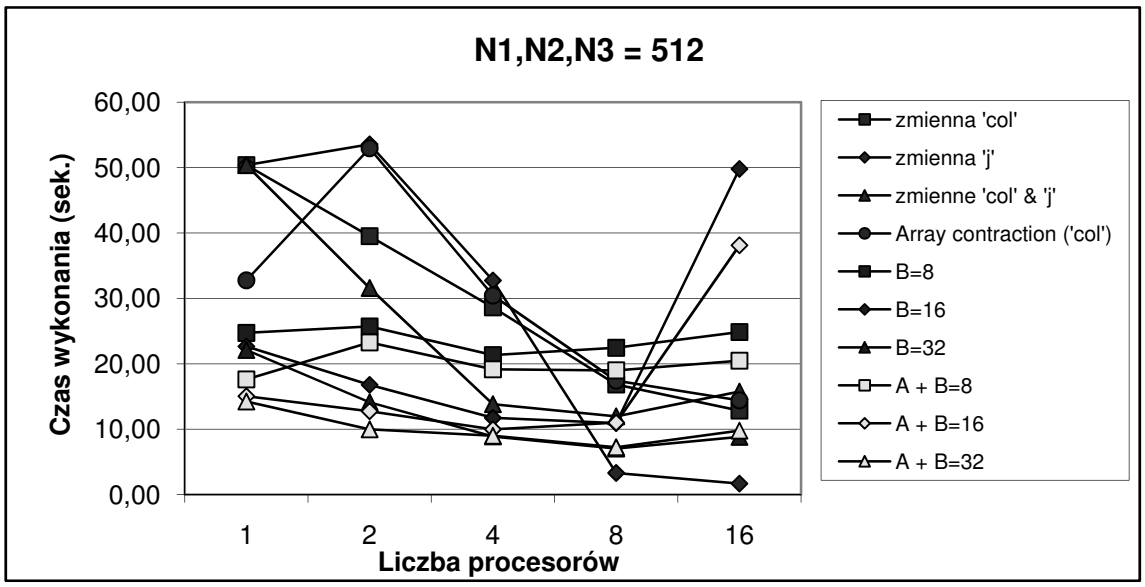
```
for(col=1;col≤N1;col++)        //par
{
    tcpre_1(col,1) = tmp_1(col,1);
    for(j=2;j≤N2;j++)  //par
        for(i=1;i≤N3;i++)
            tcpre_1(col,j) = tcpre_1(col,j) + qbnew_1(j-1,i,1) * tmp_1(col,i);
}
```

(b) Pętla gotowa do zrównoleglenia

Wyniki badań

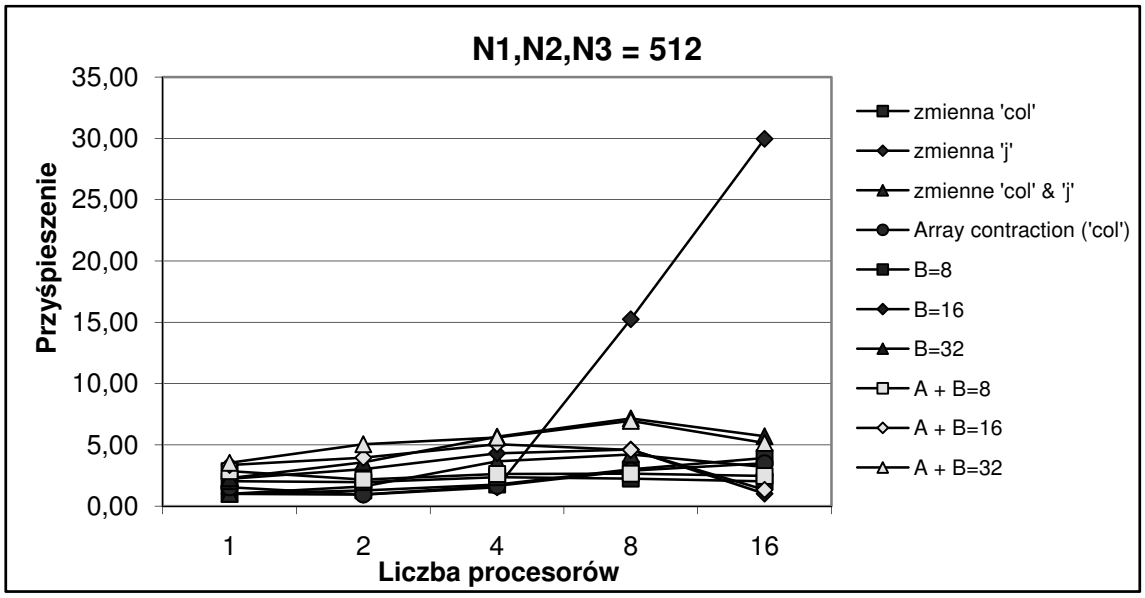
Pomiar czasu dla pętli UA_precond_4 przedstawiony został na rys. 4.6.1. Dla dwóch procesorów najbardziej wydajna transformacją jest skalaryzacja zmiennych wraz z kafelkowaniem (B=32). Dla czterech procesorów najkrótszy czas wykonania pętli uzyskujemy przy wykorzystaniu transformacji kafelkowania (B=32). Znaczącą redukcję czasu wykonania pętli w zależności od ilości procesorów otrzymano dla zrównoleglenia względem zmiennej indeksowej ‘j’ bez jakiejkolwiek transformacji pętli. Przy ośmiu i szesnastu procesorach jest najszybsze wykonanie pętli. Dla maksymalnej badanej liczby procesorów w przypadku transformacji zwiększających lokalność kodu uzyskano

zwiększenie czasu wykonania pętli. Bezpośrednie zrównoleglenie względem zmiennej indeksowej ‘j’ okazało się najbardziej wydajnym zrównolegleniem. Wynika to z faktu, iż instancje instrukcji (1) wykonywane były przed regionem równoległym, dzięki czemu pamięć podręczna jedynie jednego procesora wypełniana została wartościami tablicy ‘tcpre(col,1)’ natomiast dla pozostałych procesorów skopiowane zostały wartości ‘tcpre(col,j)’, gdzie ‘j’ jest różne dla każdego z procesorów. W związku z czym blok tablicy rozpoczynający się od indeksu 1 nie musi być zamieniany przez blok rozpoczynający się od indeksu ‘j’.

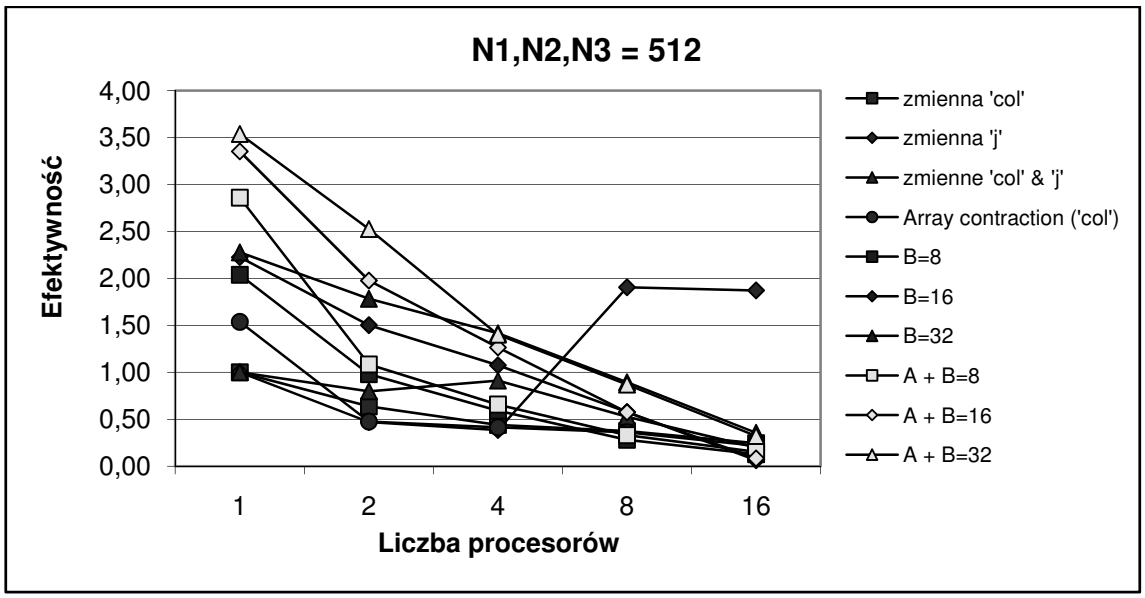


Rysunek 4.6.1. Czas wykonania pętli UA_precond_4 [op. własne]

W celu utrzymania czytelności wykresu zakres osi przyśpieszenia na rys. 4.6.2. został zredukowany od 0 do 9. W wielu przypadkach dla badanej pętli uzyskano ponad-liniowe przyspieszenie (patrz rys. 4.6.2). Największy wzrost przyśpieszenia, odbiegający najmocniej od liniowego, uzyskano dla bezpośredniego zrównoleglenia względem zmiennej indeksowej ‘j’ dla ośmiu (~15 s.) i szesnastu (~30 s.) procesorów. Efektywność dla tych dwóch przypadków wynosi odpowiednio 1,9 i 1,8 (rys. 4.6.3). Nie jest to najwyższa efektywność, gdyż dla dwóch posesorów sięga ~2,5, jednakże uzyskanie efektywności dla szesnastu procesorów na poziomie ~1,85 świadczy o dużej skalowalności i efektywnym wykorzystaniu lokalności kodu. W ogólnym przypadku uzyskanie efektywności przekraczającej wartość 1, świadczy o uzyskaniu ponad-liniowego przyspieszenia.



Rysunek 4.6.2. Przyśpieszenie dla pętli UA_precond_4 [op. własne]



Rysunek 4.6.3. Efektywność dla pętli UA_precond_4 [op. własne]

4.5.7. Pętla UA_setup_16

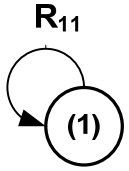
Dane wejściowe do algorytmów stanowi oryginalny kod źródłowy zapisany w języku akceptowanym przez program Petit, tab. 4.11.1(a).

Przebieg analizy

Do analizy pętli wykorzystane zostały algorytmy 3.6, 3.7, 3.8. Redukcja przestrzeni iteracji pętli nie była konieczna. Wyznaczono jedną relacje opisującą zależności proste:

flow $1 \rightarrow 1 : \{[i,j,ip] \rightarrow [i,j,ip+1] : 1 \leq i \leq N1 \;\&\&\; 1 \leq j \leq N2 \;\&\&\; 1 \leq ip < N3\}$.

Na podstawie powyższej zależności utworzono graf:



Zgodnie z algorytmem 3.6 początki krańcowe znajdują się w instancjach instrukcji (1) i opisane są przez poniższy zbiór krotek:

$SSF\_1 = \{[i,j,ip] \rightarrow [i,j,ip+1] : 1 \leq i \leq N1 \;\&\&\; 1 \leq j \leq N2 \;\&\&\; 1 \leq ip < N3\}$.

Zbiór opisujący niezależne wątki wyznaczony na podstawie algorytmu 3.7 przedstawiono poniżej.

$[(1),(1)] : \{[i,j,1,i,j,1]: 1 \leq i \leq N1 \;\&\&\; 1 \leq j \leq N2 \;\&\&\; 2 \leq N3\} \text{ union } \{[i,j,1,i,j,ip]: 1 \leq i \leq N1 \;\&\&\; 1 \leq j \leq N2 \;\&\&\; 2 \leq ip \leq N3\}$.

Na podstawie wyznaczone zbioru, możliwe jest zrównoleglenie pętli o indeksach ‘i’, ‘j’ czyli po dwóch pierwszych pętlach (tab. 4.11.1).

Liczba niezależnych wątków równa jest iloczynowi iteracji dwóch pierwszych pętli i wynosi N1*N2, przy spełnieniu następujących warunków $N3 \geq 2 \;\&\&\; N2 \geq 1$.

Tabela 4.11.1. Postać sekwencyjna i równoległa pętli UA_setup_16

```
for i=1 to N1 do
    for j=1 to N2 do
        for ip=1 to N3 do
            wdtdr(i,j) = wdtdr(i,j) + wxm1(ip)*dxm1(ip,i)*dxm1(ip,j)        (1)
        endfor
    endfor
endfor
```

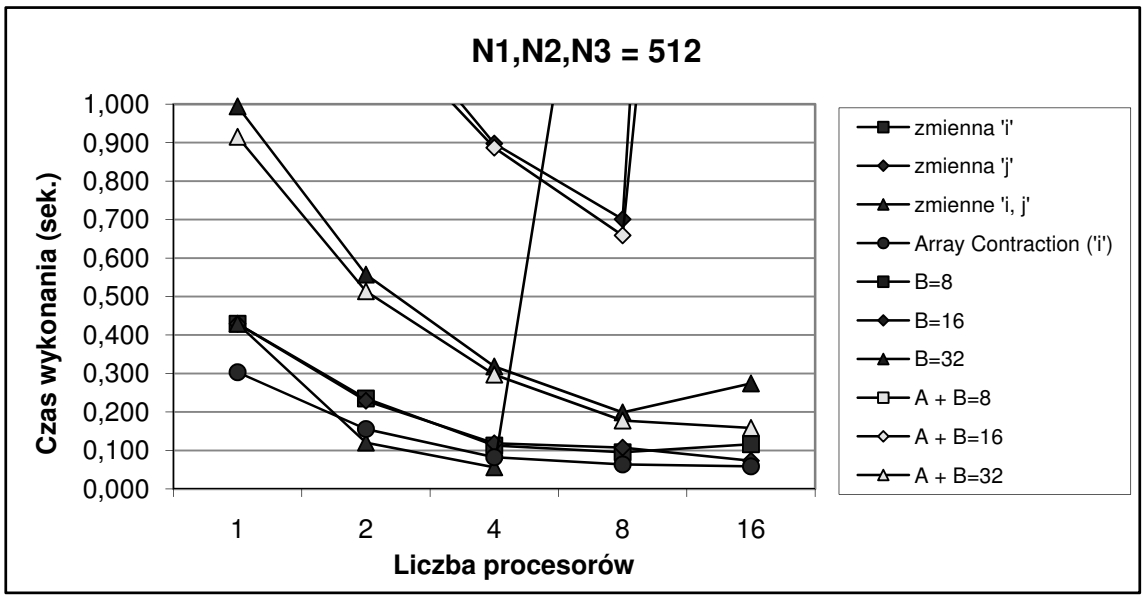
(a) Pętla poddana analizie

```
for(i=1;i≤N1;i++)       //par
    for(j=1;j≤N2;j++)  //par
        for(ip=1;ip≤N3;ip*=2)  //seq
            wdtdr_1(i,j) = wdtdr_1(i,j) + wxm1_1(ip) * dxm1_1(ip,i) * dxm1_1(ip,j);
```

(b) Pętla gotowa do zrównoleglenia

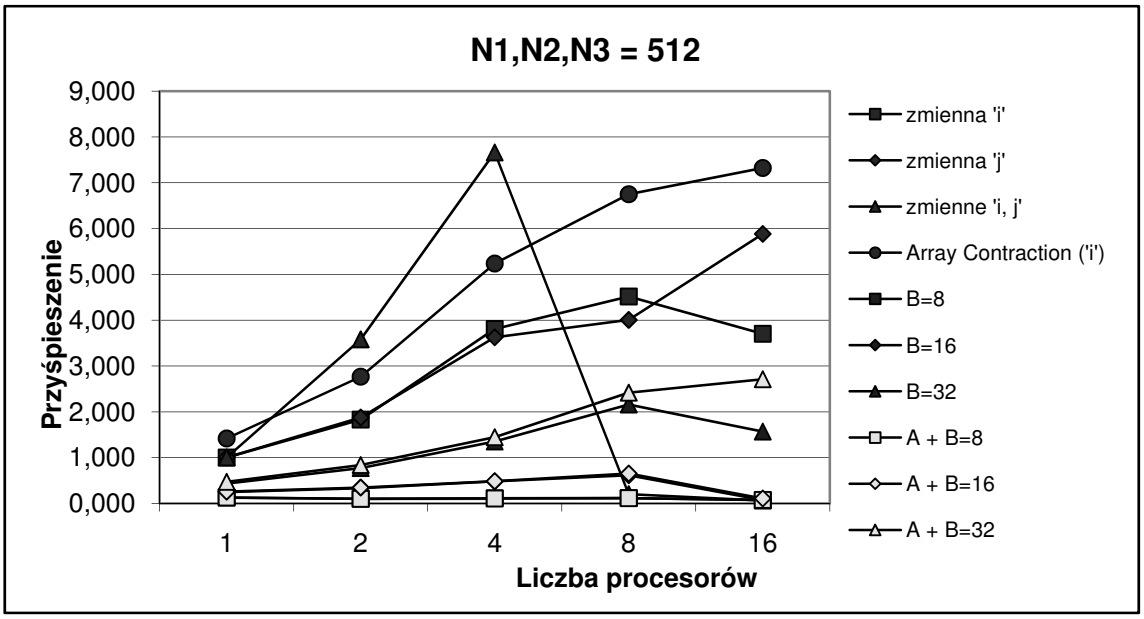
Wyniki badań

Pomiar czasu wykonania pętli przedstawiony został na rys. 4.7.1. Czas wykonania pętli w sposób sekwencyjny wyniósł 0,43 sek., w związku z tym zakres osi Y został zawężony do przedziału [0,1]. Dzięki takiemu zabiegowi wykres stał się czytelny, co umożliwiło odczytanie wartości bezpośrednio z wykresu. Minimalny czas uzyskano dla czterech procesorów przy zrównolegleniu jednocześnie po dwóch pętlach. W związku z tym regiony równoległe zostały zagnieżdżone, a liczba użytych procesorów wyniosła 16. Porównywalny czas do zagnieżdżonych regionów uzyskano przy wykorzystaniu rejestrów procesora (skalaryzacja zmiennych).

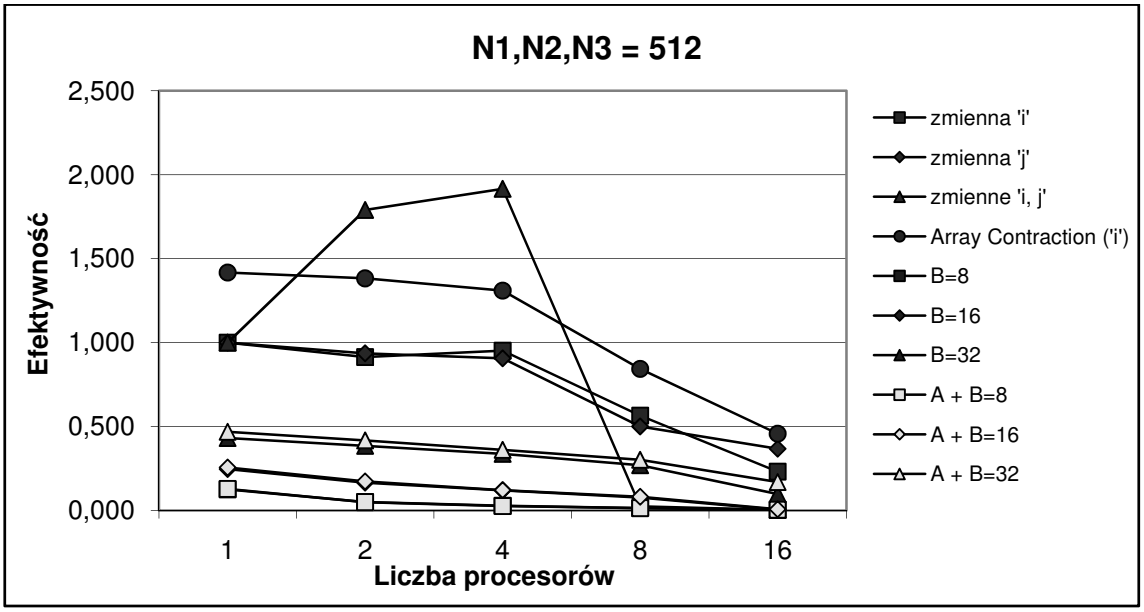


Rysunek 4.7.1. Czas wykonania pętli UA_setup_16 [op. własne]

Najwyższe przyśpieszenie uzyskano dla czterech procesorów przy zrównolegleniu dwóch pętli jednocześnie (zagnieżdżenie regionów równoległych), patrz rys. 4.7.2. Porównywalny czas uzyskano również przy wykorzystaniu szesnastu wątków dla skalaryzacji zmiennych. Najwyższa efektywność kodu przypada dla zrównoleglenia po obu pętlach jednocześnie, dla dwóch i czterech procesorów, rys. 4.7.3. Jednak należy pamiętać, że ze względu na zagnieżdżenie regionów równoległych liczba wykorzystywanych procesorów w praktyce jest wyższa. Kolejną transformacją o najwyższym wskaźniku efektywności jest skalaryzacja zmiennych dla dwóch i czterech procesorów. Warte podkreślenie jest to, iż efektywność jest powyżej jednego, co świadczy o superliniowym przyspieszeniu.



Rysunek 4.7.2. Przyśpieszenie dla pętli UA_setup_16 [op. własne]



Rysunek 4.7.3. Efektywność dla pętli UA_setup_16 [op. własne]

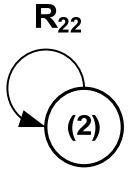
4.5.8. Pętla UA_transfer_11

Dane wejściowe do algorytmów stanowi oryginalny kod źródłowy zapisany w języku akceptowanym przez program Petit, tab. 4.12.1(a).

Przebieg analizy

W trakcie analizy pętli wykorzystano algorytmy 3.6, 3.7, 3.8. W pierwszym przebiegu analizy uzyskano jedną relację opisującą zależność prostą.

flow $2 \rightarrow 2 : \{[col,i,j] \rightarrow [col,i,j+1] : 1 \leq col \leq N1 \;\&\&\; 2 \leq i \leq N2 \;\&\&\; 1 \leq j < N3\}$



Zgodnie z relacją R_{22} początki krańcowe znajdują się w instrukcji (2), zgodnie z poniższym zbiorem:

$SSF\_2 = \{[col,i,1]: 1 \leq col \leq N1 \;\&\&\; 2 \leq i \leq N2 \;\&\&\; 2 \leq N3\}$.

Na podstawie algorytmu 3.7 oraz zbioru SSF_2 wyznaczono zbiór opisujący niezależne wątki.

$[(2),(2)] : \{[col,i,1,col,i,1]: 1 \leq col \leq N1 \;\&\&\; 2 \leq i \leq N2 \;\&\&\; 2 \leq N3\}$ union $\{[col,i,1,col,i,j]: 1 \leq col \leq N1 \;\&\&\; 2 \leq i \leq N2 \;\&\&\; 2 \leq j \leq N3\}$.

Liczba niezależnych wątków równa jest iloczynowi iteracji dwóch pierwszych pętli i wynosi N1*(N2-1) przy spełnieniu następujących warunków $N3 \geq 2$ && $N2 \geq 2$.

Tabela 4.12.1. Postać sekwencyjna i równoległa pętli UA_transfer_11

```
for col=1 to N1 do
    tmp(1,col)=tmor(1,col)     (1)
    for i=2 to N2 do
        for j=1 to N3 do
            tmp(i,col) = tmp(i,col) + qbnew(i-1,j,1)*tmor(j,col)     (2)
        endfor
    endfor
endfor
```

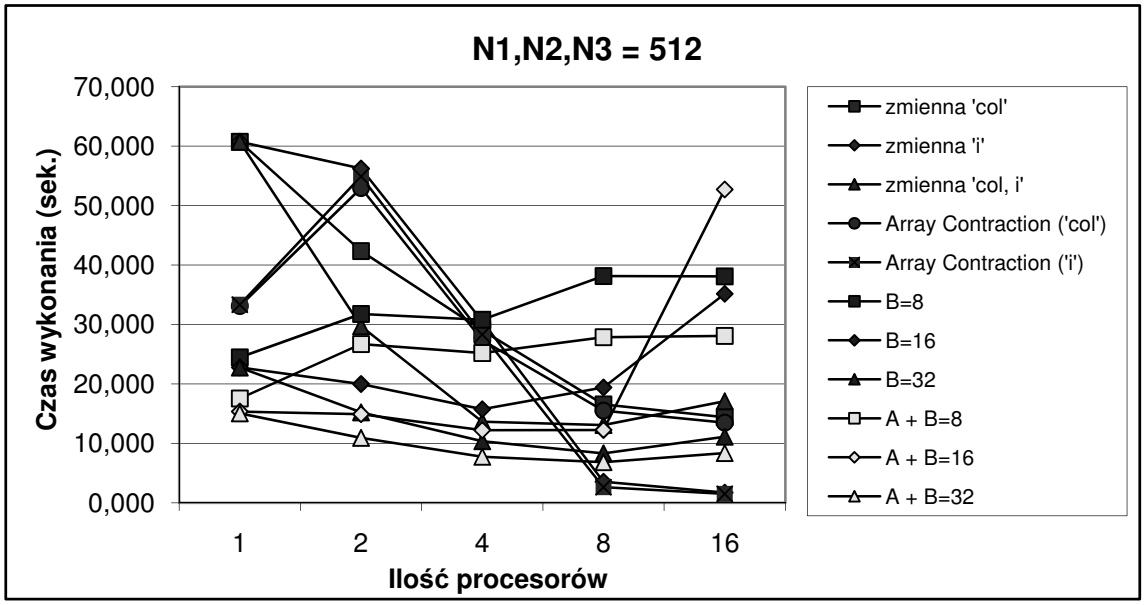
(a) Pętla poddana analizie

```
for(col=1;col≤N1;col++)        //par
    {
        tmp_1(1,col) = tmor_1(1,col);
        for(i=2;i≤N2;i++)        //par
            for(j=1;j≤N3;j++)
                tmp_1(i,col) = tmp_1(i,col) + qbnew_1(i-1,j,1) * tmor_1(j,col);
    }
```

(b) Pętla gotowa do zrównoleglenia

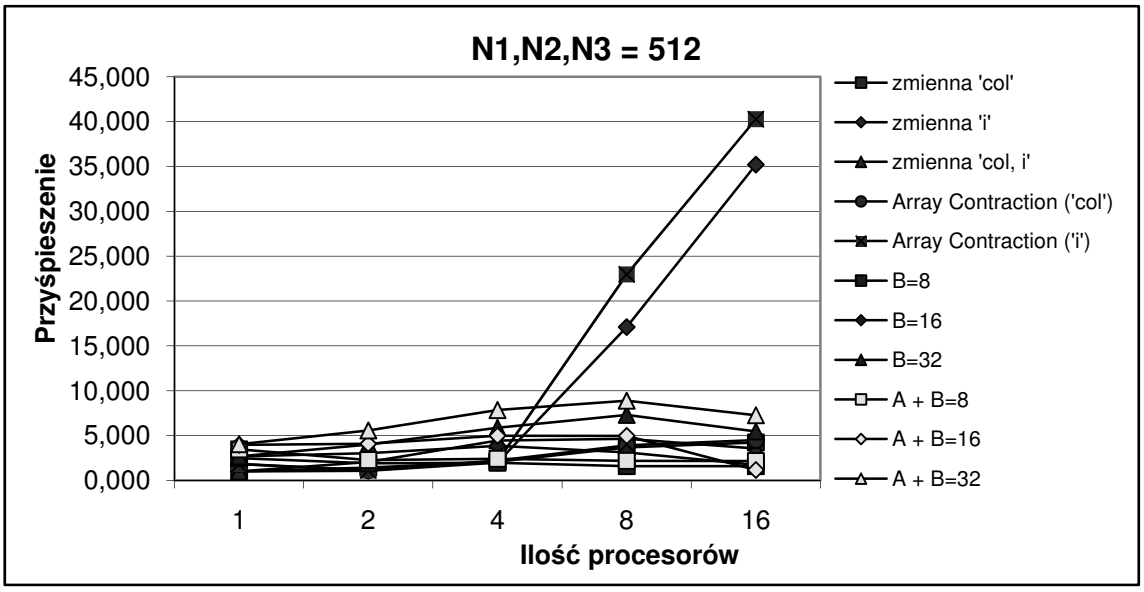
Wyniki badań

Zgodnie z poniższym rysunkiem najmniejsze czasy wykonania pętli uzyskano dla wątków wykonanych na szesnastu procesorach, przy zrównolegleniu względem zmiennej indeksowej ‘i’ oraz zastosowaniu skalaryzacji zmiennych (1,5 s.) lub przy bezpośrednim zrównolegleniu (1,7 s.). Warto zauważyć, że wszystkie zbadane przypadki pozwoliły na redukcje czasu w stosunku do czasu wykonania pętli sekwencyjne.

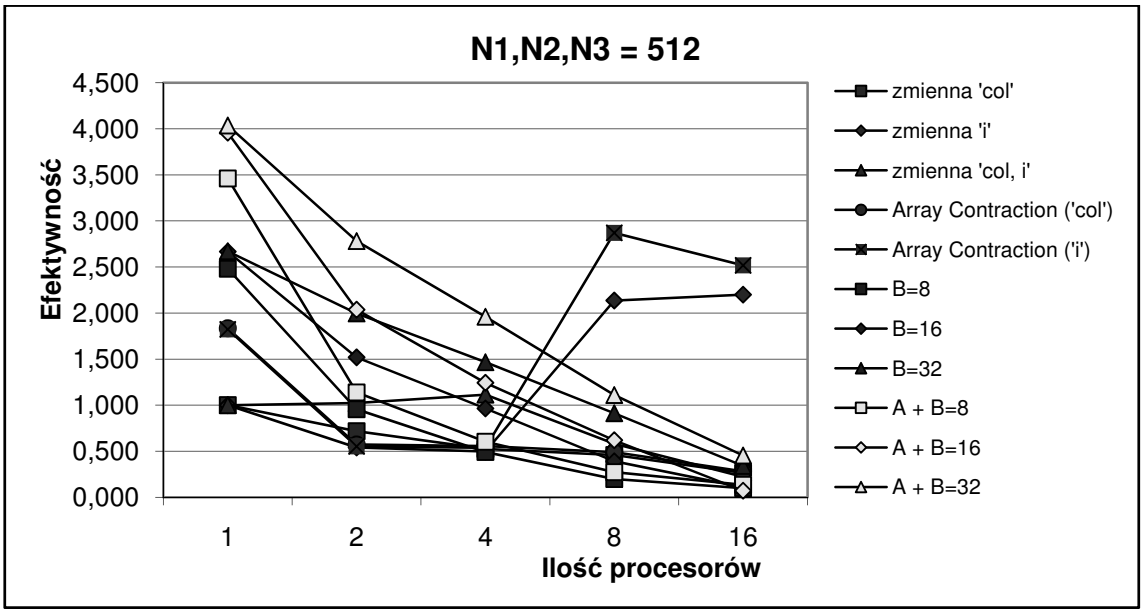


Rysunek 4.8.1. Czas wykonania pętli UA_transfer_11 [op. własne]

Jak i w poprzednich przypadkach tak i tutaj współczynnik efektywności kodu równoległego w wielu przypadkach wyniósł powyżej jedności (patrz rys. 4.8.3), co świadczy o przyśpieszeniu ponad-liniowym (patrz rys. 4.8.2). Szczególnie warto zwrócić uwagę na wyniki zrównoleglenia względem drugiej pętli (zmienna indeksowa ‘i’) z zastosowaniem skalaryzacji zmiennych jak i bez zastosowania jakiejkolwiek transformacji. W pierwszym przypadku dla szesnastu procesorów uzyskano ~40’krotne przyśpieszenie przy współczynniku efektywności ~2,5, natomiast w drugim przypadku uzyskano ~35’krotne przyśpieszenie dla efektywności ~2,2. Najwyższą efektywność ~2,85 uzyskano dla zrównoleglenia względem drugiej pętli ze skalaryzacją zmiennych dla ośmiu procesorów.



Rysunek 4.8.2. Przyśpieszenie dla pętli UA_transfer_11 [op. własne]



Rysunek 4.8.3. Efektywność dla pętli UA_transfer_11 [op. własne]

4.5.9. Pętla _BT_rhs_1

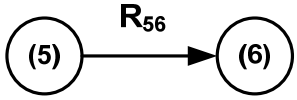
Dla oryginalnej wersji pętli (tab. 4.13.1(a)) uzyskujemy 34 relacje opisujące zależności proste, które uniemożliwiają wyznaczenie niezależnych wątków zgodnie z zaproponowanymi algorytmami. W związku z tym pętla źródłowa została zmodyfikowana do postaci przedstawionej w tab. 4.13.1(b), dla której otrzymujemy jedynie jedną zależność, między instrukcjami 5 i 6 (patrz tab. 4.13.1). Modyfikacja pętli polegała na zastąpieniu wszystkich wystąpień zmiennej skalarnej ‘rho_inv’ wyrażeniem ‘1.0/u(1,i,j,k)’ zawierającym bezpośrednie odwołanie do tablicy u(). Powyższe

przekształcenie pozwoliło na wyeliminowanie wszystkich zależności związanych ze zmienną ‘rho_inv’. Zależności wywoływane przez wspomnianą zmienną skalarną występowały wewnątrz pojedynczej iteracji jak również między iteracjami. Korzyści z powyższego przekształcenia biorąc pod uwagę redukcje zależności są oczywiste.

Przebieg analizy

Przy użyciu narzędzia Petit dla pętli z tab. 4.13.1(b) wyznaczono jedną relację opisująca zależności proste następującej postaci:

flow $5 \rightarrow 6 : \{[k,j,i] \rightarrow [k,j,i] : 0 \leq k \leq N1 \;\&\&\; 0 \leq j \leq N2 \;\&\&\; 0 \leq i \leq N3\}$.



Do dalszej analizy wykorzystane zostały algorytmy 3.6, 3.7, 3.8. Zgodnie z algorytmem 3.6 początki krańcowe znajdują się w instancjach instrukcji (5) i opisane są przez poniższy zbiór:

$SSF\_5 = \{[k,j,i]: 0 \leq k \leq N1 \;\&\&\; 0 \leq j \leq N2 \;\&\&\; 0 \leq i \leq N3\}$.

Niezależne wątki, o początkach w instancjach instrukcji (5) opisane zostały w następujący sposób:

$[(5),(5)] : \{[k,j,i,k,j,i]: 0 \leq k \leq N1 \;\&\&\; 0 \leq j \leq N2 \;\&\&\; 0 \leq i \leq N3\}$,

$[(5),(6)] : \{[k,j,i,k,j,i]: 0 \leq k \leq N1 \;\&\&\; 0 \leq j \leq N2 \;\&\&\; 0 \leq i \leq N3\}$.

Liczba niezależnych wątków:

$N3 \geq 0 \;\&\&\; N2 \geq 0 \Rightarrow (N1+1)*(N2+1)*(N3+1)$.

Tabela 4.13.1. Postać sekwencyjna i równoległa pętli _BT_rhs_1

```
for k=0 to N1 do
    for j=0 to N2 do
        for i=0 to N3 do
            rho_inv = 1.0/u(1,i,j,k)
            rho_i(i,j,k) = rho_inv
            us(i,j,k) = u(2,i,j,k) * rho_inv
            vs(i,j,k) = u(3,i,j,k) * rho_inv
            ws(i,j,k) = u(4,i,j,k) * rho_inv
            square(i,j,k) = 0.5* ( u(2,i,j,k)*u(2,i,j,k)+u(3,i,j,k)*u(3,i,j,k)+u(4,i,j,k)*u(4,i,j,k))*rho_inv
            qs(i,j,k) = square(i,j,k) * rho_inv
        endfor
    endfor
endfor
```

(a) Pętle oryginalna

```
for k=0 to N1 do
    for j=0 to N2 do
        for i=0 to N3 do
            rho_i(i,j,k) = 1.0 / u(1,i,j,k)   (1)
            us(i,j,k) = u(2,i,j,k) * 1.0 / u(1,i,j,k)    (2)
            vs(i,j,k) = u(3,i,j,k) * 1.0 / u(1,i,j,k)    (3)
            ws(i,j,k) = u(4,i,j,k) * 1.0 / u(1,i,j,k)   (4)
            square(i,j,k) = 0.5 * ( u(2,i,j,k)*u(2,i,j,k) + u(3,i,j,k)*u(3,i,j,k) + u(4,i,j,k)*u(4,i,j,k) ) *
1.0 / u(1,i,j,k)     (5)
            qs(i,j,k) = square(i,j,k) * 1.0 / u(1,i,j,k)          (6)
        endfor
    endfor
endfor
```

(b) Pętla zmodyfikowana poddana analizie

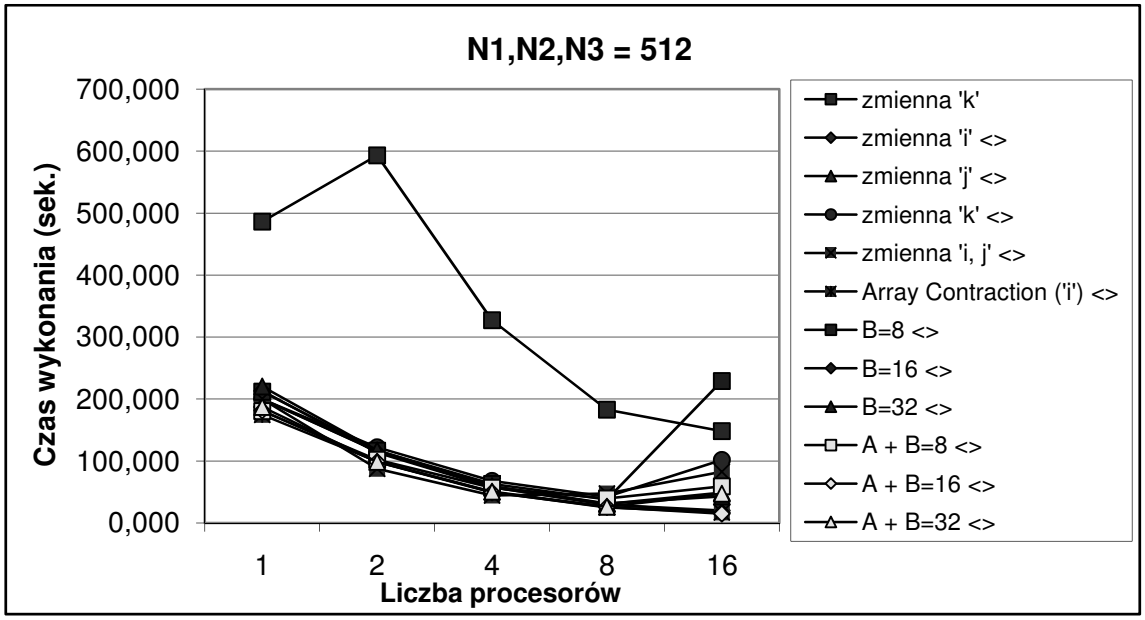
```
for(i=0;i≤N3;i++)       //par
    for(j=0;j≤N2;j++)  //par
        for(k=0;k≤N1;k++)       //par
        {
            rho_i_1(i,j,k) = 1.0 / u_1(1,i,j,k);
            us_1(i,j,k) = u_1(2,i,j,k) * 1.0 / u_1(1,i,j,k);
            vs_1(i,j,k) = u_1(3,i,j,k) * 1.0 / u_1(1,i,j,k);
            ws_1(i,j,k) = u_1(4,i,j,k) * 1.0 / u_1(1,i,j,k);
        }
for(i=0;i≤N3;i++)       //par
    for(j=0;j≤N2;j++)  //par
        for(k=0;k≤N1;k++)       //par
        {
            square_1(i,j,k) = 0.5 * ( u_1(2,i,j,k)*u_1(2,i,j,k) + u_1(3,i,j,k)*u_1(3,i,j,k) +
u_1(4,i,j,k)*u_1(4,i,j,k) ) * 1.0 / u_1(1,i,j,k);
            qs_1(i,j,k) = square_1(i,j,k) * 1.0 / u_1(1,i,j,k);
        }
```

(c) Pętla gotowa do zrównoleglenia

Wyniki badań

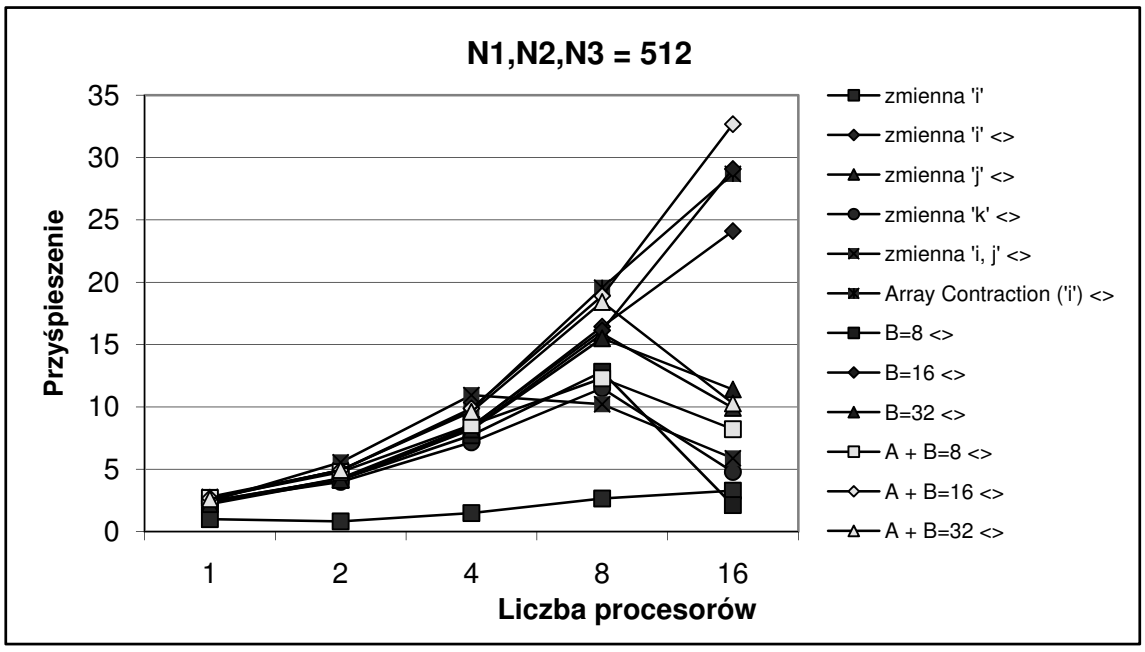
Ze względu na fakt, iż wszystkie pętle mogą zostać poddane zrównolegleniu w kodzie wynikowym zamieniono kolejność wykonywania pętli (ang. *loop intechange*). W wyniku takiej transformacji elementy tablicy przebierane są rzędami (ang. *row order*). Biorąc pod uwagę, iż w trakcie badań używano kompilatora języka C++, ma to duże znaczenie gdyż w tym języku tablice przechowywane są w pamięci rzędami.

Przyspieszenie dla wersji sekwencyjnej z odwrotną kolejnością pętli w stosunku do oryginalnego kodu wynosi 2,4.



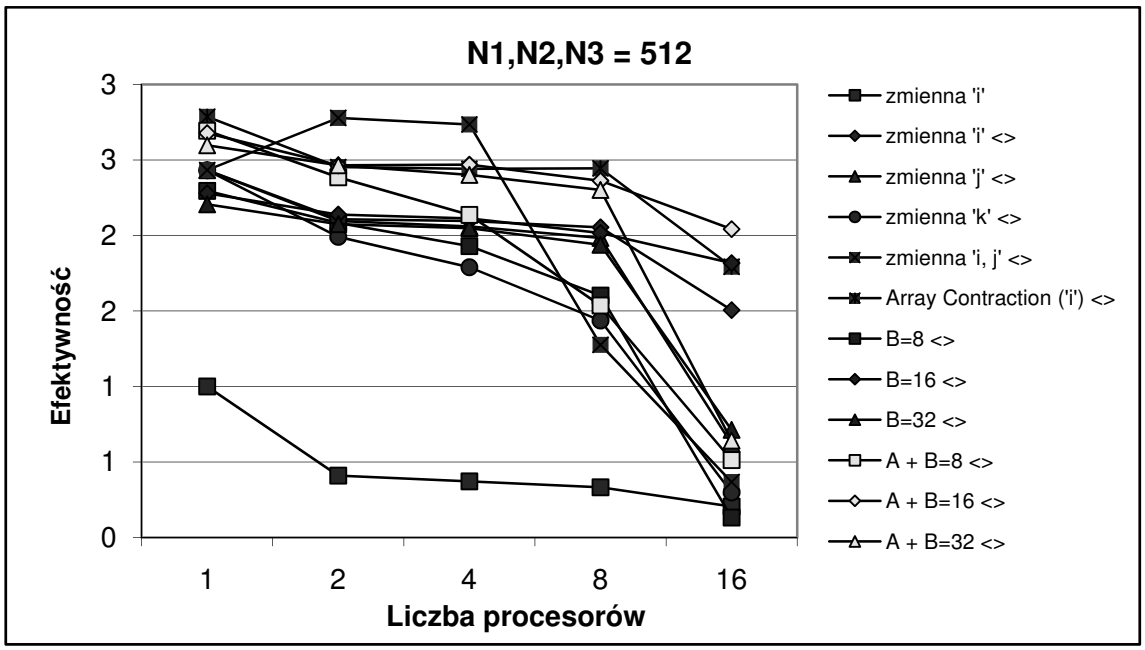
Rysunek 4.9.1. Czas wykonania pętli _BT_rhs_1 [op. własne]

Najwyższe przyśpieszenie ~32'krotne uzyskano dla skalaryzacji tablic połączonej z kafelkowaniem (B=16). Jest to przyśpieszenie ponad-liniowe, jednakże powyższa transformacja nie daje najbardziej efektywnego kodu, gdyż współczynnik efektywności w tym przypadku wynosi ~2.



Rysunek 4.9.2. Przyśpieszenie dla pętli _BT_rhs_1 [op. własne]

W przypadku zrównoleglenia dwóch pierwszych pętli jednocześnie (zagnieżdżenie regionów równoległych) dla dwóch i czterech procesorów współczynnik efektywności wynosi ~2,7. Jednakże należy pamiętać, iż w przypadku zagnieżdżenia regionów równoległych ilość używanych procesorów będzie większa od ustalonej wartości.



Rysunek 4.9.3. Efektywność dla pętli _BT_rhs_1 [op. własne]

4.5.10. Pętla _SP_ninvr_1

Podobnie jak poprzednio, relacje wyznaczone dla oryginalnej wersji pętli (tab. 4.14.1(a)) uniemożliwiły wyznaczenie niezależnych wątków pomimo dwukrotnego użycia algorytmu 3.9, w celu zawężenia przestrzeni iteracji pętli, co skutkowało redukcją liczby zależności. Zastosowanie rozszerzenia zmiennej skalarnej umożliwiło redukcję zależności z 44 przed transformacją do 10 po transformacji. Zmienne r1, r2, r3, r4, r5 zostały zastąpione odpowiednio przez zmienne tablicowe r1_1(), r2_1(), r3_1(), r4_1(), r5_1(). Odwołania do zmiennych skalarnych t1 i t2 zostały wyeliminowane poprzez bezpośrednie odwołanie do zmiennych tablicowych, analogicznie do poprzedniego przykładu.

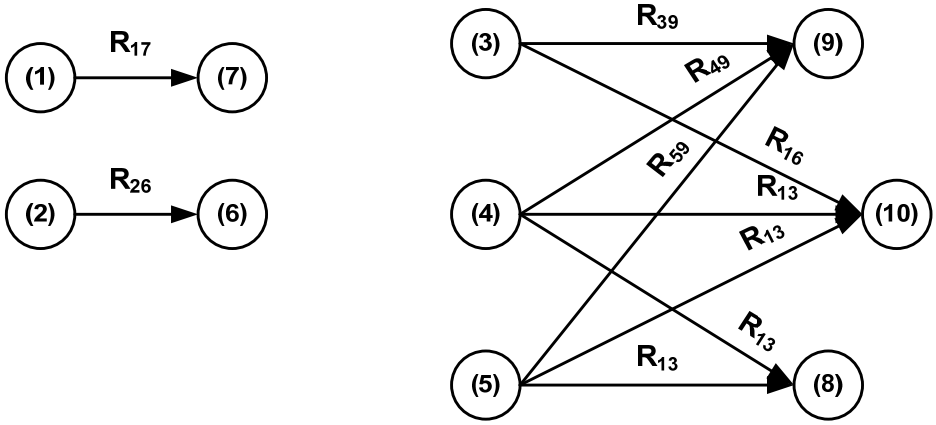
Przebieg analizy

Wyznaczone relacje dla analizowanej pętli przedstawiono poniżej:

flow $1 \rightarrow 7 : \{[k,j,i] \rightarrow [k,j,i] : 1 \leq k \leq N1 \;\&\&\; 1 \leq j \leq N2 \;\&\&\; 1 \leq i \leq N3\}$

flow $2 \rightarrow 6 : \{[k,j,i] \rightarrow [k,j,i] : 1 \leq k \leq N1 \;\&\&\; 1 \leq j \leq N2 \;\&\&\; 1 \leq i \leq N3\}$

flow 3 → 9 : {[k,j,i] → [k,j,i] : 1 ≤ k ≤ N1 && 1 ≤ j ≤ N2 && 1 ≤ i ≤ N3}
flow 3 → 10 : {[k,j,i] → [k,j,i] : 1 ≤ k ≤ N1 && 1 ≤ j ≤ N2 && 1 ≤ i ≤ N3}
flow 4 → 8 : {[k,j,i] → [k,j,i] : 1 ≤ k ≤ N1 && 1 ≤ j ≤ N2 && 1 ≤ i ≤ N3}
flow 4 → 9 : {[k,j,i] → [k,j,i] : 1 ≤ k ≤ N1 && 1 ≤ j ≤ N2 && 1 ≤ i ≤ N3}
flow 4 → 10 : {[k,j,i] → [k,j,i] : 1 ≤ k ≤ N1 && 1 ≤ j ≤ N2 && 1 ≤ i ≤ N3}
flow 5 → 8 : {[k,j,i] → [k,j,i] : 1 ≤ k ≤ N1 && 1 ≤ j ≤ N2 && 1 ≤ i ≤ N3}
flow 5 → 9 : {[k,j,i] → [k,j,i] : 1 ≤ k ≤ N1 && 1 ≤ j ≤ N2 && 1 ≤ i ≤ N3}
flow 5 → 10 : {[k,j,i] → [k,j,i] : 1 ≤ k ≤ N1 && 1 ≤ j ≤ N2 && 1 ≤ i ≤ N3}



Zgodnie z algorytmem 3.6 początki krańcowe otrzymano dla instrukcji (1) i (2):

SSF_1 = {[k,j,i]: 1 ≤ k ≤ N1 && 1 ≤ j ≤ N2 && 1 ≤ i ≤ N3}
SSF_2 = {[k,j,i]: 1 ≤ k ≤ N1 && 1 ≤ j ≤ N2 && 1 ≤ i ≤ N3}

W związku z tym, iż otrzymane początki krańcowe opisane są przez dwa różne zbiory, wyznaczenie niezależnych wątków odbywa się oddzielnie dla każdego zbioru. Dla zbioru SSF_1 otrzymujemy następujące zbiory instancji instrukcji opisujące wątki:

[(1),(1)] : {[k,j,i,k,j,i]: 1 ≤ k ≤ N1 && 1 ≤ j ≤ N2 && 1 ≤ i ≤ N3}
[(1),(7)] : {[k,j,i,k,j,i]: 1 ≤ k ≤ N1 && 1 ≤ j ≤ N2 && 1 ≤ i ≤ N3}

Dla zbioru SSF_2 uzyskujemy niezależne wątki opisywane są przez następujące zbiory:

[(2),(2)] : {[k,j,i,k,j,i]: 1 ≤ k ≤ N1 && 1 ≤ j ≤ N2 && 1 ≤ i ≤ N3}
[(2),(6)] : {[k,j,i,k,j,i]: 1 ≤ k ≤ N1 && 1 ≤ j ≤ N2 && 1 ≤ i ≤ N3}

Generowanie kodu odbywa się niezależnie dla każdego zestawu zbiorów. Zachowanie porządku leksykograficznego w obrębie zestawu zbiorów opisujących wątki o początkach w tej samej instrukcji oraz podwojenie wymiaru zbiorów pozwala na wygenerowanie zwięzłego kodu przebierającego wątki.

Dodatkowym problemem w analizowanej pętli jest występowanie zależności odwrotnej między instrukcjami (2)→(7) oraz (1)→(6). W celu wyeliminowania tej zależności utworzono kopię tablicy ‘rhs_1’. Nowo powstała tablica o nazwie ‘rhs_1_’ została zainicjalizowana wartościami tablicy ‘rhs_1’ przed wykonaniem ciała pętli.

Liczba niezależnych wątków dla każdej z pary instrukcji (1)→(7) oraz (2)→(6) równa jest iloczynowi iteracji pętli o indeksach ‘k’, ‘j’ oraz ‘i’ i wynosi:

N3 ≥ 1 && N2 ≥ 1 => N1*N2*N3.

Ogólna liczba niezależnych wątków dla całej pętli jest równa:

(N3 ≥ 1 && N2 ≥ 1) => 2*N1*N2*N3.

Tabela 4.14.1. Postać sekwencyjna i równoległa pętli _SP_ninvr_1

```
for k=1 to N1 do
    for j=1 to N2 do
        for i=1 to N3 do
            r1 = rhs(1,i,j,k)
            r2 = rhs(2,i,j,k)
            r3 = rhs(3,i,j,k)
            r4 = rhs(4,i,j,k)
            r5 = rhs(5,i,j,k)
            t1 = bt * r3
            t2 = 0.5 * ( r4 + r5 )
            rhs(1,i,j,k) = -r2
            rhs(2,i,j,k) =  r1
            rhs(3,i,j,k) = bt * ( r4 - r5 )
            rhs(4,i,j,k) = -t1 + t2
            rhs(5,i,j,k) =  t1 + t2
        endfor
    endfor
endfor
```

(a) Pętla oryginalna

```
for k=1 to N1 do
    for j=1 to N2 do
        for i=1 to N3 do
            r1_1(k,j,i) = rhs(1,i,j,k)        (1)
            r2_1(k,j,i) = rhs(2,i,j,k)        (2)
            r3_1(k,j,i) = rhs(3,i,j,k)        (3)
            r4_1(k,j,i) = rhs(4,i,j,k)        (4)
            r5_1(k,j,i) = rhs(5,i,j,k)        (5)
            rhs(1,i,j,k) = -r2_1(k,j,i)       (6)
            rhs(2,i,j,k) =  r1_1(k,j,i)       (7)
            rhs(3,i,j,k) = bt * ( r4_1(k,j,i) - r5_1(k,j,i) )     (8)
            rhs(4,i,j,k) = -( bt * r3_1(k,j,i) ) + 0.5 * ( r4_1(k,j,i) + r5_1(k,j,i) )          (9)
            rhs(5,i,j,k) =  bt * r3_1(k,j,i) + 0.5 * ( r4_1(k,j,i) + r5_1(k,j,i) ) (10)
        endfor
    endfor
endfor
```

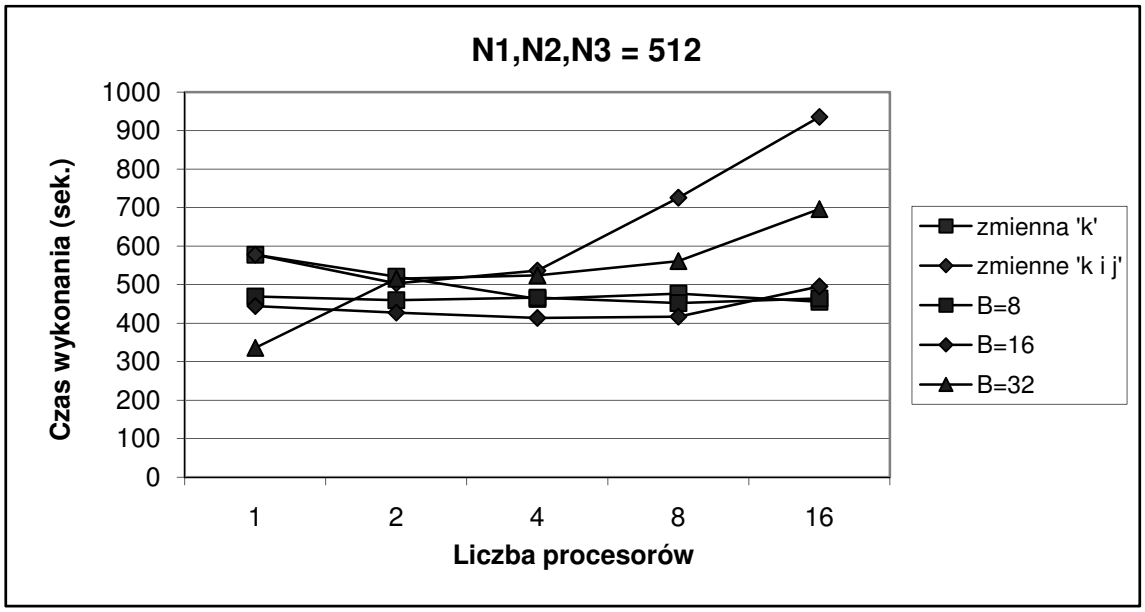
(b) Pętla zmodyfikowana poddana analizie

```
for(k=1;k≤N1;k++) //serial
    for(j=1;j≤N2;j++) //serial
        for(i=1;i≤N3;i++) //serial
        {
            r3_1(k,j,i) = rhs_1(3,i,j,k);
            r4_1(k,j,i) = rhs_1(4,i,j,k);
            r5_1(k,j,i) = rhs_1(5,i,j,k);
            rhs_1(3,i,j,k) = bt * ( r4_1(k,j,i) - r5_1(k,j,i) );
            rhs_1(4,i,j,k) = -( bt * r3_1(k,j,i) ) + 0.5 * ( r4_1(k,j,i) + r5_1(k,j,i) );
            rhs_1(5,i,j,k) =  bt * r3_1(k,j,i) + 0.5 * ( r4_1(k,j,i) + r5_1(k,j,i) );
        }
for(k=1;k≤N1;k++) //par
    for(j=1;j≤N2;j++) //par
        for(i=1;i≤N3;i++) //par
        {
            r2_1(k,j,i) = rhs_1_(2,i,j,k); //użycie tablicy rhs_1_ w celu eliminacji zależności
odwrotnej
            rhs_1(1,i,j,k) = -r2_1(k,j,i);
        }
for(k=1;k≤N1;k++) //par
    for(j=1;j≤N2;j++) //par
        for(i=1;i≤N3;i++) //par
        {
            r1_1(k,j,i) = rhs_1_(1,i,j,k); //użycie tablicy rhs_1_ w celu eliminacji zależności
odwrotnej
            rhs_1(2,i,j,k) =  r1_1(k,j,i);
        }
```

(c) Pętla gotowa do zrównoleglenia

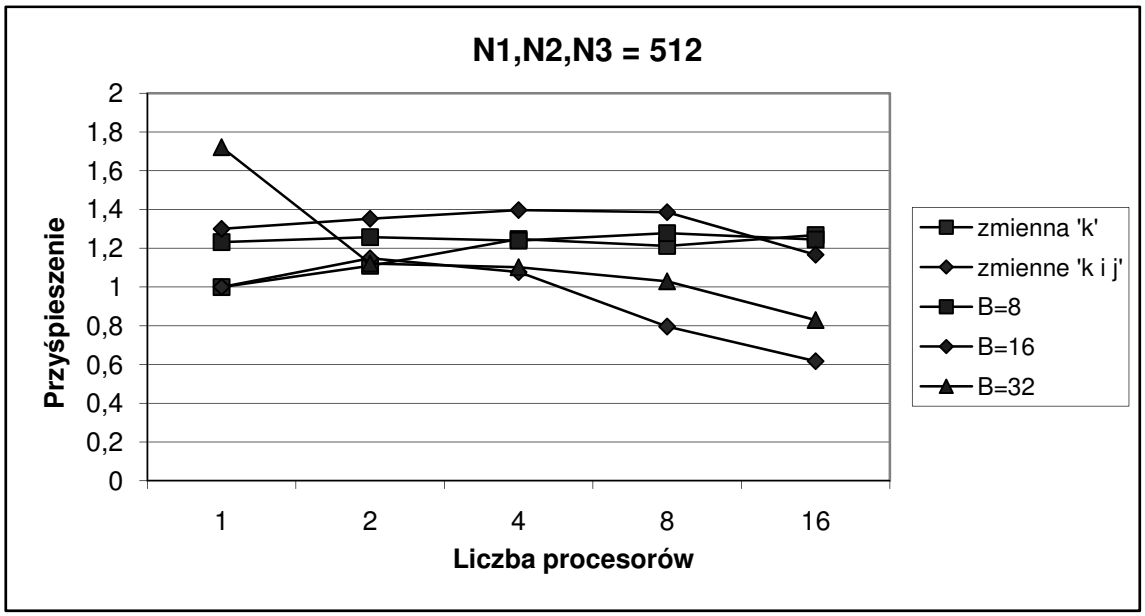
Wyniki badań

Przyśpieszenie dla pętli policzono względem wersji sekwencyjnej zmodyfikowanej (tab. 4.10(b), ponieważ tą wersję pętli poddano analizie. Minimalne czasy (różnica między pomiarami czasów w granicy błędu) uzyskano dla transformacji kafelkowania (B=16) przy czterech i ośmiu procesorach (patrz 4.10.1). Wraz ze wzrostem liczby procesorów dla większości pomiarów odnotowano wzrost czasu wykonania pętli, jedynym wyjątkiem jest zrównoleglenie dwóch ostatnich pętli względem zmiennej indeksowej ‘k’. Dla takiego podejścia uzyskano minimalny czas w grupie szesnastu procesorów, jednakże przy bardzo niskim współczynniku efektywności : ~0,079.

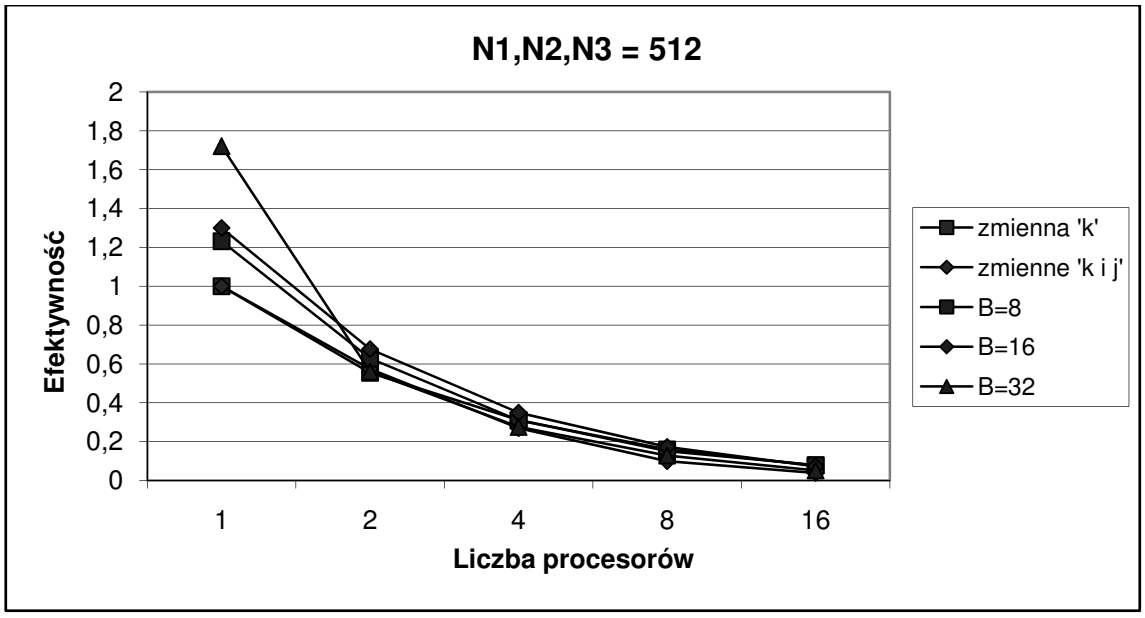


Rysunek 4.10.1. Czas wykonania pętli _SP_ninvr_1 [op. własne]

Maksymalne przyśpieszenie w żadnym przypadku nie wyniosło powyżej 1,4 (patrz rys. 4.10.2). Jedynie dla transformacji kafelkowania, gdzie B=16, dla czterech i ośmiu procesorów uzyskano przyśpieszenie około 1,4. Efektywność dla każdego z przypadków maleje w istotny sposób (patrz rys. 4.10.3), przypuszczalnie zwiększenie liczby procesorów do 32 lub więcej nie przyniesie już żadnych korzyści, wręcz przeciwnie.



Rysunek 4.10.2. Przyśpieszenie dla pętli _SP_ninvr_1 [op. własne]



Rysunek 4.10.3. Efektywność dla pętli _SP_ninvr_1 [op. własne]

4.6. Wnioski z przeprowadzonych badań

W tab. 4.15.1 przedstawiono zbiorcze zestawienie wyników dla pętli z rozdziału 4. Dla każdej pętli (pierwsza kolumna) przedstawiono następujące wskaźniki (zgodnie z porządkiem kolumn w tabeli):

- maksymalne przyśpieszenie (S_{max}),
- minimalne przyśpieszenie (S_{min}),
- maksymalna efektywność (E_{max}),
- minimalna efektywność (E_{min}),

w zależności od wielkości problemu (kolumna nr 6), ilości wątków (kolumna nr 7) oraz użycia dodatkowych transformacji (ostatnia kolumna). Zapis postaci ‘x/y/z/-’ w dwóch ostatnich kolumnach oznacza, że ilość wątków lub dodatkowe transformacje odnoszą się do kolumn:

- ‘x’ do $S_{max,}$
- ‘y’ do $S_{min,}$
- ‘z’ do $E_{max,}$
- ‘-‘ do E_{min},

przy czym oznaczenie ‘-‘ oznacza, że ilość wątków nie dotyczy tego pomiaru lub nie zastosowano żadnej dodatkowej transformacji.

Wnioski z przeprowadzonych badań są następujące:

- zaproponowane algorytmy w rozdziale 3 pozwalają na wyznaczenie niezależnych fragmentów kodu dostępnych w pętlach poddanych analizie,
- w większości przypadków uzyskano zadowalające wyniki tzn. uzyskano przyśpieszenie wykonania kodu na maszynie wieloprocesorowej w stosunku do jego sekwencyjnego odpowiednika,
- w celu osiągnięcie przyśpieszenia większego od 1 (S>1) niezbędne było zwiększenie ziarna kodu poprzez zastosowanie aglomeracji,
- zwiększenie lokalności kodu równoległego (w stosunku do zaproponowanych algorytmów) poprzez zastosowanie transformacji blokowania i skalaryzacji tablic, pozwoliło na dodatkową redukcje czasu wykonania pętli,
- w kilku przypadkach (pętle: 4.7, 4.8-4.10, 4.12, 4.14) dzięki znacznemu zwiększeniu lokalności danych możliwe było uzyskanie superliniowego przyśpieszenia,
- w przypadku pętli o małej liczbie iteracji (najczęściej górne mniejsze od 128) korzyści wynikające z wykonania niezależnych fragmentów kodu na wielu procesorach nie rekompensowały kosztów poniesionych na zarządzanie wątkami, czyli ziarnistość wątków była niewystarczająca,
- dla pętli o dużej liczbie iteracji (górne granice pętli równe 128, 256, 512) w większości przykładów uzyskano pozytywne przyśpieszenie w szczególności dla 2, 4 i 8 procesorów, świadczy to o wystarczającej ziarnistości kodu,
- zwiększenie liczby wątków (jeden wątek mapowany na jeden procesor) do wykonania tego samego zadania najczęściej prowadzi do spadku efektywności kodu, wiąże się to z kosztami, jakie należy ponieść na zarządzanie watkami,
- zastosowanie harmonogramowania statycznego [23] przy założeniu że:
    - liczba wątków nie jest większa od liczby jednostek przetwarzających,
    - brak jest wywłaszczeń wątków,

  dzięki zadowalającej lokalności kodu oraz mniejszych nakładach na harmonogramowanie, pozwala na osiągnięcie mniejszych czasów wykonania w odniesieniu do harmonogramowania dynamicznego [23],
- zastosowanie aglomeracji (zwiększenie ziarna kodu) poprzez wykonanie kilku niezależnych fragmentów kodu na jednym procesorze, pozwala na

zwiększenie ziarnistości kodu, prowadzi to do zredukowania czasu wykonania pętli,

- w uzyskanym kodzie, w przypadku sekwencji pętli możliwych do zrównoleglenia występujących bezpośrednio po sobie (oznaczone słowem ‘par’) zmiana kolejności pętli w taki sposób aby odwołanie do kolejnych komórek tablic występowało zgodnie ze sposobem ich alokowania w pamięci operacyjnej (zależnym od języka programowania, patrz podrozdział 4.3), pozwala na znaczne zwiększenie lokalności kodu a w konsekwencji na redukcję czasu wykonania pętli.

Przeprowadzone badania eksperymentalne pozwalają stwierdzić, że teza „Możliwe jest opracowanie algorytmów do automatycznego wyszukiwania drobno- i gruboziarnistej równoległości w pętlach programowych o złożoności pozwalającej na ich zastosowanie w akademickich oraz przemysłowych kompilatorach optymalizujących” została udowodniona. Opracowane algorytmy pozwalają na ich implementację w kompilatorach akademickich jak i przemysłowych. Zostało to udowodnione przez opracowanie własnych narzędzi i zastosowanie ich do automatyzacji procesu transformacji pętli sekwencyjnych na ich odpowiedniki równoległe. Czas wykonania transformacji i generacji kodu zgodnie z zaproponowanymi algorytmami jest akceptowalny do zastosowań w kompilatorach akademickich oraz przemysłowych. W trakcie badań eksperymentalnych, dla odpowiednio dużego problemu, w większości przypadków uzyskano pozytywne przyśpieszenie S ($S > 1$).

Tabela 4.15.1. Zestawienie wyników z przeprowadzonych badań

Pętla	S_{max}	S_{min}	E_{max}	E_{min}	Wielkość problemu	Ilość wątków	Dodatkowe transformacje[3]
BT_error_2	1,88	-	0,82	-	N1,N2,N3 = 512	8/-/2/-	B(32)+A/-/B(8)/-
	-	0,84	-	0,04	N1,N2,N3 = 64	16	-/B(16)/-/-
FT_auxfnct_2	21,7	-	1,54	-	N1,N2,N3 = 256	16/-/8/-	-
	-	0,59	-	0,06	N1,N2,N3 = 64	-/2/-/16	-/B(8)/-/-
LU_HP_l2norm_2	4,2	-	-	-	N1,N3,N5 = 128, N2,N4 = 32	8	A
	-	0,31	-	0,02	N1,N3,N5 = 64, N2,N4 = 32	16	B(8) + A
	-	-	1,35	-	N1,N3,N5 = 512, N2,N4 = 32	2	B(32) + A
UA_adapt_1	35,2	-	2,44	0,06	N = 256	16/-/2/16	B(32)+A+FG/-/ B(32)+A+FG/ B(8)+A
	-	1,74	-	-	N = 128	2	B(8) + A
UA_diffuse_3	93,2	-	7,18	-	N1,N2,N3,N4 = 512	16/-/2/-	A/-/ B(32)+A/-
	-	1,44	-	-	N1,N2,N3,N4 = 256	2	B(8)
	-	-	-	0,22	N1,N2,N3,N4 = 64	16	B(8)
UA_precond_4	29,96	-	-	-	N1,N2,N3 = 512	16	-
	-	0,03	-	~0	N1,N2,N3 = 64	16	-
	-	-	3,01	-	N1,N2,N3 = 256	2	-
UA_setup_16	10,64	~0	-	~0	N1,N2,N3 = 256	8/16/-/16	A/-/-/-
	-	-	1,91	-	N1,N2,N3 = 512	4	-
UA_transfer_11	55,64	-	7,7	-	N1,N2,N3 = 256	16/-/4/-	A/-/-/-
	-	0,03	-	~0	N1,N2,N3 = 64	16	-
_BT_rhs_1	32,67	-	2,77	-	N1,N2,N3 = 512	16/-/2/-	B(16)+A+LI/-/-/-
	-	0,26	-	0,01	N1,N2,N3 = 64	16	-
_SP_ninvr_1	1,39	-	-	-	N1,N2,N3 = 512	4	B(16)
	-	0,12	-	~0	N1,N2,N3 = 64	16	-
	-	-	0,72	-	N1,N2,N3 = 256	2	-

[3] B(x) – oznacza transformację blokowania (ang. *bloking*) o wielkości bloku x, A – oznacza skalaryzację zmiennych (ang. *array contr action*), FG – oznacza łączenie grafów w celu zwiększanie ziarna kodu (ang. *fusing graphs*), LI – zmiana kolejności pętli (ang. *loop interchange*)

5. PRACE POKREWNE

Uzyskane wyniki w niniejszej dysertacji stanowią rozwinięcie badań prowadzonych przez Pugha i Rossera w dziedzinie partycjonowania przestrzeni iteracji (ang. *iteration space slicing framework*) [80]. W swoich pracach przedstawili sposób zastosowania podziału przestrzeni w celu optymalizacji komunikacji między procesorowej, a w szczególności użycie partycjonowania do transformacji łączenia pętli (ang. *loop fusion*), tolerancji opóźnień w przesyłaniu komunikatów oraz umożliwienia łączenia komunikatów. W celu stworzenia fragmentu kodu (ang. *slice*), wykorzystywana jest operacja tranzytywnego domknięcia do obliczenia tranzytywnej zależności dla wszystkich iteracji. Dzięki temu, za pośrednictwem tranzytywnego domknięcia, możliwe jest wyznaczenie zbioru osiągalnych iteracji w kierunku zgodnym z zależnością jak i odwrotnym. Jednakże Pugh i Rosser nie przedstawili w swojej pracy [80] w jaki sposób można wyznaczyć niezależne fragmenty (ang. *synchronization-free slices*). W niniejszej dysertacji zaproponowane zostało rozwiązanie pozwalające na wyznaczenie zbiorów opisujących niezależne fragmenty kodu. Zgodnie z wiedzą autora, dotychczas nie przedstawiono rozwiązania pozwalającego na wyznaczenie niezależnych fragmentów kodu z wykorzystaniem partycjonowania przestrzeni pętli.

Zmiana kolejności wykonania pętli (ang. *loop interchanges*), odwrócenie pętli (ang. *loop reversal*) i przekoszenie pętli (ang. *loop skewing*) to transformacje umożliwiające wykrywanie gruboziarnistej równoległości w pętlach idealnie zagnieżdżonych umożliwiające przemieszczenie do najbardziej zewnętrznego gniazda pętli. Wykonanie powyższych transformacji oraz ich kombinacji umożliwiają transformacje unimodularne (ang. *unimodular transformations*) [7], [89], [90]. Znanych jest wiele algorytmów wykorzystujących unimodularne transformacje połączone z mozaikowaniem pętli (ang. *loop tiling*) w celu zwiększenia równoległości i lokalności

pętli, gdzie zależności reprezentowane są jako skierowane wektory dystansu [68]. Docelowa kombinacja transformacji ustalana jest poprzez wyszukanie odpowiedniej macierzy unimodularnej, umożliwiającej wygenerowanie kodu zgodnego z architekturą SPMD (ang. *Single Program Multiple Data*). Jednakże unimodularne transformacje mają kilka zasadniczych ograniczeń [68]:

- operacje należące do pojedynczej iteracji są niepodzielne – dlatego tez niemożliwe jest opisanie ważnych transformacji np. podział i dystrybucja pętli lub zmiana kolejności wykonania wyrażeń,
- niemożliwe jest zastosowanie transformacji unimodularnych bezpośrednio dla pętli nieidealnie zagnieżdżonych (istnieją rozwiązania pozwalające na transformację pętli nieidealnej do pętli idealnie zagnieżdżonej [91]),
- nie umożliwiają wykonanie transformacji takich jak: podział pętli (ang. *loop fission*), łączenie pętli (ang. *loop fusion*), skalowanie (ang. *scaling*), reindeksacja (ang. *reindexing*) lub (ang. *reordering*).

Metoda zaproponowana w publikacji [35] pozwala na wyznaczenie równoległości w pętlach pozbawionych komunikacji (ang. *synchronization-free parallelism*). Metoda ta bazuje na harmonogramowaniu opartym na rekurencji Hamiltona (ang. *Hamiltonian Recurrence*). W odróżnieniu od transformacji unimodalnych, w tym przypadku każda instrukcja rozpatrywana jest jako oddzielna jednostka, która może zostać przydzielona do procesora. Główne ograniczenia tej metody to:

- metoda może zostać zastosowana dla pętli o stałym wektorze dystansu
- pętla nie może być sparametryzowana
- dla pętli wielokrotnie zagnieżdżonych możliwe jest zrównoleglenie tylko jednej z nich.

W pracy J.M. Anderson i M.S. Lam [4] zaproponowano algorytm pozwalający na automatyczną dekompozycję danych oraz obliczeń. Algorytm ten umożliwia optymalizację zarówno równoległości jak i lokalności. Jednakże zaproponowane podejście ograniczone jest do sekwencji idealnie zagnieżdżonych pętli.

Metoda ATF (ang. *Affine Partitioning Framework*) opisana między innymi w [27], [31], [32], [68], [67], [69], łączy w sobie dużą liczbę wcześniej zaproponowanych transformacji. Zgodnie z założeniami tej metody przy pomocy przekształceń afinicznych możliwe jest wykonanie wielu transformacji [60] np.: zmiana kolejności wykonania pętli, dystrybucja pętli (ang. *loop distribution*), przekoszenie pętli (ang. *loop skewing*), mozaikowanie pętli (ang. *tiling*), podział przestrzeni pętli (ang. *index set splitting*), zmiana kolejności wykonania wyrażeń (ang. *statement reordering*). ATF

bazuje na założeniu, że transformacje mogą być reprezentowane, jako harmonogram mapujący oryginalną przestrzeń iteracji (wejściową) na nową przestrzeń (wyjściową, docelową). Transformacje afiniczne pozwalają w jednolity sposób reprezentować różne transformacje. W tab. 5.1 przedstawiono zestawienie transformacji możliwych do wykonania w zależności od rodzaju przyjętego podejścia.

Tabela 5.1. Możliwe transformacje w zależności od typu przekształceń [60]

Nazwa transformacja	**Przekształcenia**[4]	
	Transformacje unimodalne	**ATF**
Zmiana kolejności pętli (ang. *loop interchange*)	●	●
Odwrócenie wykonania pętli (ang. *loop reversal*)	●	●
Przekoszenie pętli (ang. *loop skewing*)	●	●
Zmiana kolejności wykonania instrukcji (ang. *statement reordering*)	○	●
Dystrybucja pętli (ang. *loop distribution*)	○	●
Łączenie pętli (ang. *loop fusion*)	○	●
Wyrównanie pętli (ang. *loop alignment*)	○	●
(ang. *loop interleaving*)	○	●
Podział pętli na bloki (ang. *loop blocking*)	○	●
Podział zmiennych indeksowych (ang. *index set-splitting*)	○	●
Redukcja wymiaru pętli (ang. *loop coalescing*)	○	●

Transformacje unimodularne [90] pozwalają na modyfikację pętli idealnie zagnieżdżonych, natomiast przekształcenia afiniczne (harmonogramowanie afiniczne) pozwalają na modyfikację zarówno pętli idealnie jak i nieidealnie zagnieżdżonych oraz sekwencji tych pętli. Zgodnie z modelem harmonogramowania afinicznego dla każdej instrukcji przypisane jest mapowanie afiniczne na podstawie, którego instancja instrukcji opisana przez współrzędne pętli mapowana jest na przestrzeń czasu. Operacje, których wykonanie ustalone zostało w tym samym czasie wykonywane są równolegle. Czas wykonania w modelu afinicznym może być wielowymiarowy, koordynaty w dziedzinie czasu (ang. *time domain*) wykonywane są sekwencyjne zgodnie z porządkiem leksykograficznym.

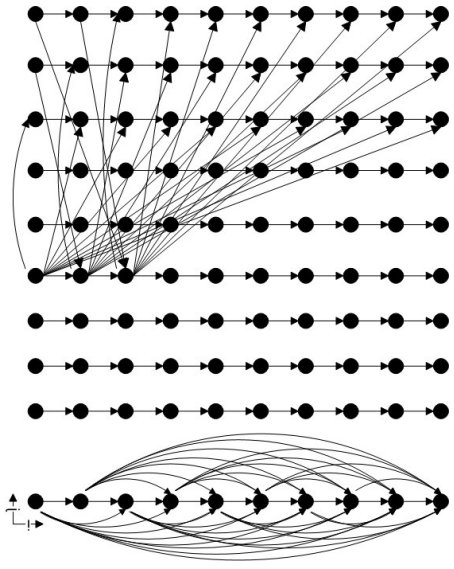
[4] ● – transformacja możliwa do wykonania, ○ – nie pozwala na uzyskanie transformacji

Feautrier przedstawił algorytm wyszukujący „picewise affine schedule" do minimalizacji czasu wykonania programu [31], [32]. Rozwiązanie oparte na algorytmie Feautriera pozwala ustalić moment wykonania instrukcji bez przypisania jej do jednostki wykonawczej. Komunikacja między równoległymi zadaniami nie jest modelowana i musi zostać wykonana jawnie poprzez zastosowanie dodatkowego algorytmu [33]. Transformacja „picewise affine schedule" na kod równoległy nie jest zadaniem prostym, a kod wynikowy może być skomplikowany [90]. Zgodnie z wiedzą autora metoda ATF, ze względu na dużą złożoność, nie została zaimplementowana w żadnym z obecnie dostępnych kompilatorów zarówno akademickich jak i przemysłowych.

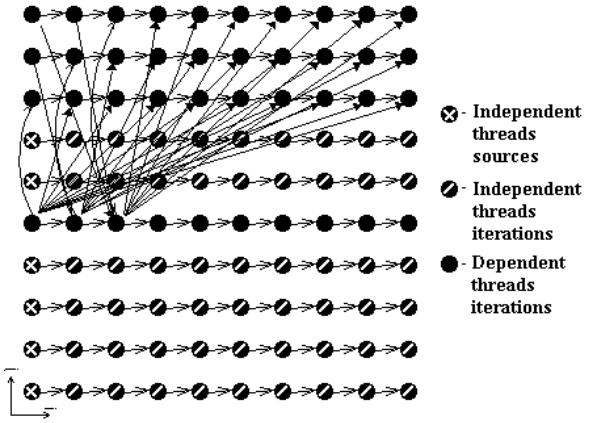
Należy zaznaczyć również, że metoda ATF nie wykrywa całkowitej równoległości zawartej w pętlach programowych. Przykład 5.1 przedstawia pętlę, dla której zgodnie z podejściem zaproponowanym w niniejszej pracy, w odróżnieniu od transformacji afinicznych, możliwe jest wyznaczanie niezależnych wątków.

```
for i=1 to 10 do
    for j=1 to 10 do
        a(j+4,j+1) = a(i+2*j+1,i+j+3)
    endfor
endfor
```

Przykład 5.1 Pętla z zależnościami niejednorodnymi



Rysunek 5.1. Zależności dla pętli z przykładu 5.1 [op. własne]



Rysunek 5.2. Zredukowany zbiór zależności dla przykładu 5.1 [op. własne]

Dla pętli z przykładu 5.1 otrzymujemy następujące relacje zależności (przy użyciu narzędzia Petit)

PR1 := {[i,5] → [i,i+7] : 1 ≤ i ≤ 3,
PR2 := {[i,5] → [i',i+7] : 1 ≤ i < i' ≤ 10 && i ≤ 3},
PR3 := {[1,9] → [2,5] } union {[2,10] → [3,5] },
PR4 := {[i,j] → [j-7,5] : 1 ≤ i ≤ j-8 && j ≤ 10},
PR5 := {[i,j] → [i',j] : 1 ≤ i < i' ≤ 10 && 1 ≤ j ≤ 10}.

Rysunek 5.1 przedstawia zależności opisywane przez powyższe relacje (zależności między iteracjami [i,j] : i=1 & 1≤j≤10 występują również dla 2≤i≤10). Należy zaznaczyć, że zależności występujące dla iteracji [i,j] : i=1 & 1≤j≤10, występują również dla iteracji [i,j]: 2≤i≤10 & 1≤j≤10. Po wykonaniu redukcji nadmiarowych zależności (patrz rozdział 3) otrzymujemy zredukowany zbiór relacji:

R1 := {[i,5] → [i,i+7] : 1 ≤ i ≤ 3},
R2 := {[i,5] → [i',i+7] : 1 ≤ i < i' ≤ 10 && i ≤ 3},
R3 := {[i,j] → [j-7,5] : 1 ≤ i ≤ j-8 && j ≤ 10},
R4 := {[i,j] → [i+1,j] : 1 ≤ i ≤ 9 && 1 ≤ j ≤ 10}.

Zależności opisywane przez powyższe relacje przedstawione zostały na rys. 5.2. Po wykonaniu algorytmu 3.3 otrzymujemy zbiory iteracji RSS oraz SFS zawierające początki krańcowe dla wątków odpowiednio zależnych oraz niezależnych:

RSS := {[1,j]: 8 ≤ j ≤ 10} union {[1,5]},
SFS := {[1,j]: 6 ≤ j ≤ 7} union {[1,j]: 1 ≤ j ≤ 4}.

W celu wygenerowania zbiorów zawierających iteracje dla niezależnych wątków zastosowano algorytm 3.5, gdzie wejście algorytmu stanowi zbiór SFS oraz relacje R1, R2, R3, R4. Uzyskany kod gotowy do zrównoleglenia przedstawiono poniżej.

```
par for(j = 1; j ≤ 4; j++) {
    for(i = 1; i ≤ 10; i++)
        a(j+4,j+1) = a(i+2*j+1,i+j+3)
}
par for(j = 6; j ≤ 7; j++) {
    for(i = 1; i ≤ 10; i++)
        a(j+4,j+1) = a(i+2*j+1,i+j+3)
}
```

Zgodnie z zaproponowanym podejściem dla przykładu 5.1 możliwe jest wyznaczenie niezależnych wątków w przestrzeni iteracji pętli. Dodatkowo należy zaznaczyć, iż zadaniem powyższego kodu jest jedynie przebieranie niezależnych wątków oraz iteracji należących do nich. W celu uzyskania kodu semantycznie zgodnego z oryginalną wersją pętli, konieczne jest dodanie kodu przebierającego zależne wątki i ich iteracje z zachowaniem porządku leksykograficznego.

```
j = 5
for(i = 1; i ≤ 10; i++)
    a(j+4,j+1) = a(i+2*j+1,i+j+3)
for(i = 1; i ≤ 10; i++)
    for(j=8; i ≤ 10; j++)
        a(j+4,j+1) = a(i+2*j+1,i+j+3)
```

Podsumowując można stwierdzić, że zaproponowane podejście jest prostsze od metody ATF. Autor zaimplementował zaproponowane algorytmy w postaci narzędzi akademickich generujących pseudokod, który w kolejnym kroku został ręcznie przekonwertowany na kod równoległy zgodny ze standardem OpenMP. Dużą zaletą zaproponowanych rozwiązań jest to, że pozwalają na wykrywanie równoległości w pętlach, dla których zgodnie z metodą ATF niemożliwe jest wyznaczenie równoległości.

6. PODSUMOWANIE

W niniejszej pracy zaproponowano zbiór algorytmów umożliwiających wyszukanie niezależnych fragmentów kodu w pętlach programowych dowolnie zagnieżdżonych (rozdział 3). Dla pętli idealnie zagnieżdżonych zaproponowano dodatkowe rozwiązanie charakteryzujące się zwiększoną ziarnistością oraz mniejszą złożonością. Przedstawione algorytmy pozwalają na wyznaczenie drobno- i gruboziarnistego kodu. W przypadku zbyt małej ziarnistości uzyskanego kodu możliwe jest zastosowanie aglomeracji w celu zwiększenia wielkości ziarna i redukcji liczby niezależnych wątków. W rozdziale 4 przedstawiono metodykę badań, charakterystykę systemów badawczych oraz wyniki z przeprowadzonych eksperymentów.

Rozdział 5 zawiera opis prac pokrewnych do zagadnienia poruszanego w niniejszej dysertacji. Przedstawiono ograniczenia istniejących rozwiązań oraz pokazano zalety zaproponowanego podejścia. Zaprezentowano przykład, dla którego jedna z najmocniejszych metod (ATF) zawodzi w wyznaczeniu istniejącej równoległości.

Do najważniejszych wyników pracy należą:

- algorytmy umożliwiające wyznaczenie niezależnych fragmentów kodu dla pętli dowolnie zagnieżdżonych z zależnościami afinicznymi,
- stworzenie narzędzia akademickiego będącego implementacją zaproponowanych algorytmów,

Główną zaletą opracowanych algorytmów jest to, że pozwalają na wyznaczenie równoległości w pętlach programowych, dla których najmocniejsze metody zawodzą. Dodatkowo złożoność zaproponowanych algorytmów pozwala na ich użycie zarówno w kompilatorach akademickich jak i przemysłowych do automatycznej transformacji pętli sekwencyjnych na ich odpowiedniki równoległe.

6.1. Wnioski końcowe

W oparciu o opracowane algorytmy oraz przeprowadzone badania eksperymentalne sformułowano następujące wnioski końcowe:

- Zastosowanie zbioru algorytmów 3.1 – 3.9 pozwala na wyznaczenie drobno- i grubo-ziarnistej równoległości w pętlach sparametryzowanych idealnie oraz dowolnie zagnieżdżonych z zależnościami afinicznymi.
- Algorytmy redukujące nadmiarowe zależności pozwalają na zwiększenie liczby pętli, w których możliwe jest wyznaczenie niezależnych fragmentów kodu.
- Niezależne fragmenty kodu pozwalają na zwiększenie i efektywne wykorzystanie lokalności kodu, w szczególności lokalności tymczasowej (ang. *temporal locality*).
- Zastosowanie aglomeracji (ang. *agglomeration*) dla niezależnych fragmentów kodu oraz transformacji blokowania (ang. *bloking*) pozwala na znaczną redukcje czasu wykonania pętli, co potwierdzono w trakcie badań eksperymentalnych
- W przypadku, gdy niemożliwe jest wyznaczenie niezależnych fragmentów kodu dla całej przestrzeni iteracji pętli, zawężenie przestrzeni do wybranych zmiennych indeksowych (zgodnie z wektorami dystansu) pozwala na wyznaczenie niezależnych fragmentów kodu w podprzestrzeniach pętli.
- W oparciu o zaproponowane algorytmy możliwe jest opracowanie akademickich oraz przemysłowych kompilatorów zrównoleglających dla pętli dowolnie zagnieżdżonych, pozwalających na zmniejszenie czasu wykonania programów równoległych w stosunku do ich sekwencyjnych odpowiedników.
- Zaproponowane w niniejszej pracy algorytmy umożliwiają wyznaczenie niezależnych fragmentów kodu dla większego spektrum pętli w porównaniu do znanych transformacji pętli programowych.
- Zaproponowane algorytmy pozwalają na udzielenie odpowiedzi na wiele pytań z dziedziny transformacji pętli programowych m.in.:
  - Ile niezależnych fragmentów kodu dostępne jest w badanej pętli ?
  - Jaki jest koszt związany z wyznaczeniem i generacją kodu niezależnych fragmentów kodu?
  - Czy wygenerowany kod jest efektywny?
  - Jak dużo równoległości zostało pominiętych przez znane transformacje?

6.3. Dalsze badania

W trakcie prowadzonych prac badawczych napotkano na wiele problemów, których rozwiązanie pozwoliłoby na zwiększenie - i tak już szerokiego - zakresu stosowalności zaproponowanych algorytmów:

- Niezbędnym jest opracowanie rozwiązania pozwalającego na wyznaczenie Tranzytywnego Domknięcia (ang. *Transitive Closure*) dla ogólnego przypadku lub zwiększenie spektrum przypadków dla których jest to możliwe, np. wyznaczanie dokładnego tranzytywnego domknięcia w przypadku gdy niemożliwe jest opisanie go przy pomocy afinicznych ograniczeń.
- Opracowanie algorytmów umożliwiających wyznaczenie niezależnych fragmentów kodu dla części przestrzeni iteracji pętli, w której relacje zależności opisują wspólne iteracje (wątek tworzą nie tylko łańcuchy zależnych instancji instrukcji, lecz zależne instancje instrukcji o dowolnej strukturze np. krata, sześcian, ...).
- Rozszerzenie zaproponowanego podejścia z pojedynczej pętli na sekwencję n ($n > 1$) niepowiązanych ze sobą pętli.

Powyższymi zagadnieniami autor pragnie zająć się w swoich przyszłych badaniach.

ZAŁACZNIK A

SPECYFIKACJA WYKORZYSTANEGO SPRZETU DO BADAŃ ESKPERYMENTALNYCH

Badania eksperymentalne przeprowadzono na dwóch maszynach równoległych. Pomiarów dla pętli idealnie zagnieżdżonych dokonano na maszynie Intel Quad-Core znajdującej się na Wydziale Informatyki Politechniki Szczecińskiej. Pętle nieidealnie zagnieżdżone zostały zbadane na maszynie SGI Altrix 3700 znajdującej się w Poznańskim Centrum Superkomputerowo-Sieciowym. Charakterystyka dwóch maszyn badawczych została zamieszczona poniżej.

SGI Altrix 3700

Tabela A.1. Charakterystyka systemu SGI Altrix 3700

Procesor:	Itanium2 1,5 GHz (NUMA)
Liczba procesorów:	128
Max. liczba proc. dla użytkownika:	16
Moc obliczeniowa:	768 GFLOPs
Pomięć operacyjna:	256 GB
Pojemność dyskowa:	1 TB
System operacyjny:	GNU/Linux (Red Hat 4.1.2-14)
Kompilator:	Intel C/C++ Compilers

Intel Quad-Core

Tabela A.2. Charakterystyka systemu Intel Quad-Core

Procesor:	Intel Xeon E5310
Liczba procesorów:	2
Max. liczba proc. dla użytkownika:	2
Pomięć operacyjna:	2 GB
Pojemność dyskowa:	400 GB
System operacyjny:	Linux
Kompilator:	GCC (ver. 4.2)

ZAŁĄCZNIK B

PŁYTA CD

Na dołączonej płycie CD w zależności od katalogu umieszczono następujące dodatki:

- src_LIVERMORE – kody źródłowe zbioru pętli LIVERMORE
- src_NPB3.2 – kody źródłowy zbioru pętli testowych NPB w wersji 3.2
- exp_LIVERMORE – kody źródłowe pętli wykorzystane w trakcie badań (rozdział 4), zapisane zgodnie ze standardem OpenMP, gotowe do kompilacji
- exp_NPB
    - Original – tłumaczenia oryginalnych pętli NPB na wersje zapisane zgodnie z językiem Petit
    - Modified – oryginalne wersje pętli poddane modyfikacją w celu redukcji liczby zależności
    - Parallel – kod równoległy wybranych pętli zapisany zgodnie ze standardem OpenMP
- stat_LIVERMORE – w plikach umieszczono wszystkie wykonane pomiary dla pętli Livermore, przedstawionych w rozdziale 4 wraz z wykresami obrazującymi uzyskane wyniki
- stat_NPB – w plikach umieszczono wszystkie wykonane pomiary dla pętli ze zbioru NPB, przedstawionych w rozdziale 4 wraz z wykresami obrazującymi uzyskane wyniki. Dodatkowo w pliku „summery.pdf" przedstawiono statystyki dla wszystkich przeanalizowanych petli.

PUBLIKACJE WŁASNE[5]

1. W. Bielecki, K. Siedlecki. Wyznaczanie niezależnych wątków w pętlach idealnie zagnieżdżonych. PS, M VII SNI, 2002.
2. W. Bielecki, K. Siedlecki. Finding Free Schedules for Non-uniform Loops. Klagenfurt, EuroPar 2003.
3. W. Bielecki, K. Siedlecki. Wyszukiwanie początków niezależnych wątków w dowolnie zagnieżdżonych pętlach programowych. PS, M IX SNI, 2004.
4. W. Bielecki, K. Siedlecki. Finding Free Schedules for Non-Uniform Loops. Electronic Modeling 2004.
5. W. Bielecki, K. Siedlecki. Extracting synchronization-free threads in perfectly nested loops using the Omega project software. WSEAS SEPADS, SALZBURG 2005.
6. W. Bielecki, M. Pałkowski, K. Siedlecki. Badania efektywności metod wyszukiwania gruboziarnistej równoległości w pętlach programowych. PS, M XI SNI, 2006.
7. W. Bielecki, K. Siedlecki. Wyszukiwanie równoległości niewymagającej synchronizacji w pętlach idealnie zagnieżdżonych. PS, M XI SNI, 2006.
8. W. Bielecki, K. Kraska, K. Siedlecki. Increasing Program Locality by Extracting Synchronization Free Slices in Arbitrarily Nested Loops. ACS 2007.
9. W. Bielecki, K. Siedlecki. Finding sources of synchronization-free slices in perfectly nested loops. Electronic Modeling 2007.
10. W. Bielecki, K. Siedlecki. Extracting synchronization-free slices in perfectly nested loops. Electronic Modeling 2007.

[5] sortowanie zgodnie z datą publikacji

BIBLIOGRAFIA

1. Allen R., Kennedy K. Optimizing compilers for modern architectures: A Dependence- based Approach. Morgan Kaufmann Publishers, Inc., 2001.
2. Amdahl G. On the Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities. Proceedings of the AFIPS Conference. 1967.
3. Ancourt C., Irigoin F. Scanning polyhedra with do loops. in Proceedings of the Third ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming. 1991, 39-50.
4. Anderson J. M., Lam M. S. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. Computer Systems Laboratory, Stanford University,. CA 94305.
5. Bacon D. F., Graham S. L., Sharp O. J. Compiler Transformations for High-Performance Computing. California : Computer Science Division, University of California, Berkeley.
6. Bailey D., Barszcz E., Barton J., Browning D., Carter R., Dagum L., Fatoohi R., Fineberg S., Frederickson P., Lasinski T., Schreiber R., Simon H., Venkatakrishnan V., Weeratunga S. THE NAS PARALLEL BENCHMARKS. March 1994. RNR Technical Report RNR-94-007.
7. Banerjee U. Unimodular transformations of double loops. Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing. 1990, 192-219.
8. Bastou C. Code Generation in the Polyhedral Model Is Easier Than You Think. Proceedings of the PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques. 2004, strony 7-16.
9. Bielecki W. Esentials of Parallel and distributed Computing. Informa. 2002.
10. Bielecki W., Kraska K., Siedlecki K. Increasing Program Locality by Extracting Synchronization Free Slices in Arbitrarily Nested Loops. ACS. 2007.
11. Bielecki W., Pałkowski M., Siedlecki K. Badania efektywności metod wyszukiwania gruboziarnistej równloegłości w pętlach programowych. PS, M XI SNI. 2006.
12. Bielecki W., Siedlecki K. Extracting synchronization-free slices in perfectly nested loops. Electronic Modeling. 2007.

13. Bielecki W., Siedlecki K. Extracting synchronization-free threads in perfectly nested loops using the Omega project software. WSEAS SEPADS. SALZBURG 2005.
14. Bielecki W., Siedlecki K. Finding Free Schedules for Non-Uniform Loops. Electronic Modeling. 2004.
15. Bielecki W., Siedlecki K. Finding Free Schedules for Non-uniform Loops. EuroPar. 2003.
16. Bielecki W., Siedlecki K. Finding sources of synchronization-free slices in perfectly nested loops. Electronic Modeling. 2007.
17. Bielecki W., Siedlecki K. Wyszukiwanie początków niezależnych wątków w dowolnie zagnieżdżonych pętlach programowych. PS, M IX SNI. 2004.
18. Bielecki W., Siedlecki K. Wyszukiwanie równoległości nie wymagającej synchronizacji w pętlach idealnie zagnieżdżonych. PS, M XI SNI. 2006.
19. Bielecki W., Siedlecki K. Wyznaczanie niezależnych wątków w pętlach idealnie zagnieżdżonych. PS, M VII SNI. 2002.
20. Blume W., et al. Effective Automatic Parallelization with Polaris. International Journal of Parallel Programming. 1995.
21. Boulet P., Darte A., Silber G. A., Vivien F. Loop parallelization algorithms: from parallelism extraction to code generation. Parallel Computing. 1998, Tomy 24, 3-4, 421-444.
22. Calland P. Y., Darte A., Robert Y., Vivien F. On the removal of anti and output dependences.July 1997.
23. Chandra R., Menon R., Dagum L., Kohr D., Maydan D., McDonald J. Parallel Programming in OpenMP. Morgan Kaufmann, 2000.
24. Chapman B., Jost G., Pas R. Using OpenMP, Portable Shared Memory Parallel Programming. MIT Press, 2008.
25. Collard J. F., Feautrier P., Risset T. Construction of do loops from systems of affine constraints. Parallel Processing Letters. 1995, Tom 5, 421-436.
26. Dagum L., Menon R. OpenMP: An Industry-Standard API for Shared-Memory Programming. SILICON GRAPHICS INC.
27. Darte A., Robert Y., Vivien F. Scheduling and Automatic Parallelization. Birkhäuser Boston, 2000.

28. Diaconescu R. Object Based Concurrency for Data Parallel Applications: Programmability and Effectiveness. Trondheim, Norway : Department of Computer and Information Science, Norwegian University of Science and Technology.
29. El-Rewini H., Abd-El-Barr M. ADVANCED COMPUTER ARCHITECTURE AND PARALLEL PROCESSING., A JOHN WILEY & SONS, INC PUBLICATION, 2005.
30. Faigin K., et al. The polaris internal representation. International Journal of Parallel Programming. 1994.
31. Feautrier P. Some efficient solutions to the affine scheduling problem, part i, one dimensional time. International Journal of Parallel Programming 21. 1992, 313-348.
32. Feautrier P. Some efficient solutions to the affine scheduling problem, part ii, multidimensional time. International Journal of Parallel Programming 21. 1992, 389- 420.
33. Feautrier P. Towards automatic distribution, Technical Report. Institiut Blaise Pascal/Laboratoire MASI, December 1992.
34. Foster I. Designing and Building Parallel Programs. http://www-unix.mcs.anl.gov/dbpp/.
35. Gavalda R., Ayguade E., Torres J. Obtaining Synchronization-Free Code with Maximum Parallelism. 1996.
36. Geist G. A., Kohl J. A., Papadopoulos P. M. PVM and MPI: a Comparison of Features. 1996.
37. Grama A., Gupta A., Karypis G., Kumar V. An Introduction to Parallel Computing.Addison Wesley, 2003.
38. Grama A., Gupta A., Kumar. V. Isoefficiency Function: A Scalability Metric for Parallel Algorithms and Architectures. IEEE Parallel and Distributed Technology, Special Issue on Parallel and Distributed Systems: From Theory to Practice. August 1993, Tomy Volume 1, Number 3.
39. Gropp W., Lusk E., Skjellum A. Using MPI: Portable Parallel Programming with the Message-Passing Interface, second edition. MIT Press, 1999.
40. Gustafson J. Reevaluation Amdahl's law. Communications of the ACM. 1988, Tom 31, 5.
41. HENNESSY J. L., PATTERSON D. A. Computer Architecture A Quantitative Approach. San Mateo : Morgan-Kauffman Publisher, 2007.

42. High Performance Fortran - Language Specication, Version 2.0. 1997.
43. http://suif.stanford.edu/
44. http://www.csm.ornl.gov/pvm/
45. http://www.ece.northwestern.edu/cpdc/Paradigm/Paradigm.html
46. http://www.man.poznan.pl/zasoby/komputery_obliczeniowe/guarana/index.html
47. http://www.nas.nasa.gov/Resources/Software/npb.html
48. http://www.netlib.org/benchmark/, Netlib Repository at UTK and ORNL.
49. http://www.openmp.org/
50. http://www-unix.mcs.anl.gov/mpi/
51. https://computing.llnl.gov/tutorials/openMP/
52. Intel® C++ Compiler Documentation. Intel Corporation.
53. Intel® C++ Optimizing Applications. Intel Corporation.
54. Jin H., Frumkin M., Yan J. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. October 1999. NAS Technical Report NAS-99-011.
55. Juhasz Z., Kacsuk P., Kranzlmuller D. Distributed and Parallel Systems - cluster and grid computing. Springer Science, Business Media, Inc. 2005.
56. Kazuhisa I., Motoki O., Hironori K. Cache Optimization for Coarse Grain Task Parallel Processing using Inter-Array Padding. Department of Computer Science, Waseda University. Tokyo, Japan.
57. Kelly W., Maslov V., Pugh W., Rosser E., Shpeisman T., Wonnacott D. New User Interface for Petit and Other Extensions. User Guide. December 5, 1996.
58. Kelly W., Maslov V., Pugh W., Rosser E., Shpeisman T., Wonnacott D. The omega library interface guide. University of Maryland : brak nazwiska, 1995. Technical Report CS-TR-3445.
59. Kelly W., Maslov V., Pugh W., Rosser E., Shpeisman T., Wonnacott D. The Omega Library, Version 1.1.0, Interface Guide. November 18, 1996.
60. Kelly W., Pugh W. A Framework for Unifying Reordering Transformations. Univ. Of Maryland, 1993.
61. Kelly W., Pugh W., Rosser E., Shpeisman T. Transitive Closure of Infinite Graphs and its Applications. International Journal of Parallel Programming. 1996, Tomy 24, n. 6, 579-598.
62. Kozielski S., Szczerbiński Z. Komputery równoległe. Architektura. Elementy programowania. WNT. 1993.

63. Kunz Robert C. PERFORMANCE BOTTLENECKS ON LARGE-SCALE SHARED-MEMORY MULTIPROCESSORS. December 2004.
64. Lewis B., Berg D. J. Multithreaded Programming with Pthreads. Sun Microsystem Press, 1998.
65. Lewis B., Berg D. J. PThreads Primer, A Guide to Multithreaded Programming. Sun Microsystem Press, 1996.
66. Lilja D. J. A Multiprocessor Architecture Combining Fine-Grained and Coarse-Grained Parallelism Strategies. pp. 729-751, Parallel Computing, May 1994, Tomy Vol. 20, No. 5.
67. Lim A. W., Cheong G. I., Lam M. S. An affine partitioning algorithm to maximize parallelism and minimize communication. Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing. 1999.
68. Lim A. W., Lam M. S. Communication-free parallelization via affine transformations. proceedings of the Seventh workshop on languages and compilers for parallel computing. 1994, strony 92-106.
69. Lim A. W., Lam M. S. Maximizing parallelism and minimizing synchronization with affine transforms. Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1997.
70. Lim W., Lam M. S. Maximizing parallelism and minimizing synchronization with affine transforms. Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1997.
71. Moldovan D. Parallel Processing: From Applications to Systems. Morgan Kaufmann Publishers, Inc., 1993.
72. Morrison R. S. Cluster Computing: Architectures, Operating Systems, Parallel Processing & Programming Languages. 1998 – 2003.
73. Muchnick S. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997.
74. Nichols B., Buttlar D., Farrell J. P. Pthreads Programming. O'Reilly, 1998.
75. OpenMP Application Program Interface, Version 2.5. May 2005.
76. Pacheco P. S. Parallel Programming with MPI. Morgan Kaufmann, 1997.
77. Padua D., et al. Polaris: A new-generation parallelizing compiler for MPPs. University of Illionois. 1993. CSRD-1306.

78. Park I. PARALLEL PROGRAMMING METHODOLOGY AND ENVIRONMENT FOR THE SHARED MEMORY PROGRAMMING MODEL. Purdue University, 2000. Thesis.
79. Polychronopoulos C., Girkar M., et al. The Structure of Parafrase-2: An Advanced Parallelizing Compiler for C and Fortran. Languages and Compilers for Parallel, MIT Press, 1990.
80. Pugh W., Rosser E. Iteration Space Slicing and Its Application to Communication Optimization. Proceedings of the International Conference on Supercomputing. 1997, 221-228.
81. Pugh W., Wonnacott D. An exact method for analysis of value-based array data dependences. Workshop on Languages and Compilers for Parallel Computing. 1993.
82. Quillere F., Rajopadhye S., Wilde D. Generation of efficient nested loops from polyhedra. International Journal of Parallel Programming. 2000.
83. Sahnin S., Thanvantri V. Parallel Computing: Performance Metrics and Models. Computer & Information Sciences Department University of Florida.
84. Snir M., Otto S., Huss-Lederman S., Walker D., Dongarra J. MPI - The Complete Reference, Volume 1, The MPI Core, Second Edition. Massachusetts Institute of Technology, 1996.
85. Song Y., Xu R., Wang C., Li Z. Improving Data Locality by Array Contraction.
86. Ted Lewis G. Foundations of Parallel Programming, A Machine-Independent Approach. Los Alamitos, California : IEEE Computer Society Press, 1994.
87. Waheed A., Yan J. Parallelization of NAS Benchmarks for Shared Memory Multiprocessors. March 1998. NAS Technical Report NAS-98-010.
88. Wilson R., et al. The SUIF compiler system: A parallelizing and optimizing research compiler. SIGPLAN, 1994. CSL-TR-94-620.
89. Wolf M. E. Improving locality and parallelism in nested loops. 1992. Ph.D. Dissertation CSL-TR-92-538.
90. Wolf M. E., Lam M. S. A loop transformation theory and an algorithm to maximize parallelism. Transactions on Parallel and Distributed Systems. October 1991, Tom 2(4), strony 452-470.
91. Xue J. Unimodular Transformations of Non-Perfectly Nested Loops. Departmet of Mathematics, Statistics and Computing Science, University of New England