POLITECHNIKA SZCZECIŃSKA

WYDZIAŁ INFORMATYKI

KATEDRA INŻYNIERII OPROGRAMOWANIA

mgr inż. Marek Pałkowski

Algorytmy zwiększające ekstrakcję równoległości w pętlach programowych

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 Małgorzacie oraz Rodzicom za wszelką pomoc w trakcie pisania pracy, cierpliwość i motywację.

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 instrukcji
δ^{-1}	symbol oznacza wystąpienie antyzależności (ang. *antidependence*) między dwoma instancjami instrukcji
δ^{0}	symbol oznacza wystąpienie zależności 'po wyjściu' (ang. *output dependence*) między dwoma instancjami instrukcji
$\exists$	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
$\Leftrightarrow$	wtedy, i tylko wtedy gdy – oznaczenie równoważności
$\mid$	zawężenie (obcięcie), w notacji Omega Calculator oznaczana dwukropkiem *:*,
$\rightarrow$	odwzorowanie; np. $X \rightarrow Y$ oznacza przyporządkowanie elementowi ze zbioru X elementu ze zbioru Y
&&	iloczyn (część wspólna) dwóch zbiorów lub relacji, zapis według notacji Omega Calculator
||	unia (suma) dwóch zbiorów lub relacji, zapis według notacji Omega Calculator, oznaczany również jako *OR* lub *union*.

Słownik skrótów

ATF	ang. *Affine Transformation Framework*, zbiór przekształceń afinicznych pozwalających na wykonanie wielu transformacji w sposób jednolity
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
UTDSP	ang. *University of Toronto Digital Signal Processor* - zbiór programów cyfrowego przetwarzania sygnałów pozwalający na zbadanie wydajności systemów jedno- i wieloprocesorowych

1. WSTĘP

Rozdział niniejszy stanowi wprowadzenie do problematyki poruszanej w rozprawie doktorskiej. W podrozdziale 1.1. przedstawiono uzasadnienie podjęcia tematu, 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. Określono również zakres prowadzonych badań oraz przedstawiono narzędzia programowe niezbędne do ich realizacji. Opisano metodykę prowadzonych badań oraz sposób weryfikacji zaproponowanych algorytmów. Podrozdział 1.3 obejmuje strukturę pracy wraz z krótką charakterystyką poszczególnych rozdziałów.

1.1. Stan problemu

Od kilku dziesięcioleci w świecie nauki i techniki coraz większą rolę odgrywają obliczenia i symulacje komputerowe. Wykorzystywane są one przede wszystkim w najszybciej rozwijających się gałęziach przemysłu i nauki, takich jak: przemysł kosmiczny, lotniczy, motoryzacyjny, elektroniczny, fizyka jądrowa i kwantowa, elektronika, medycyna, chemia i biologia. Zapotrzebowanie na maszyny o dużej mocy obliczeniowej nieustannie wzrasta, również w przypadku komputerów osobistych. Tendencja ta ma charakter stały i ciągle ulega nasileniu. Nieustanny postęp technologiczny w dziedzinie sprzętu komputerowego oraz rozwój oprogramowania w zakresie przetwarzania równoległego skutkują popularyzacją tej dziedziny w środowiskach akademickich i biznesowych.

Początki prac nad wykorzystaniem wielu procesorów do współbieżnego wykonywania obliczeń sięgają lat 60. ubiegłego wieku. Obecnie problemy technologiczne nie pozwalają przyspieszać taktowania procesorów w takim tempie,

jakie zakłada prawo Moore'a. Rozwiązaniem są systemy wieloprocesorowe lub procesory wielordzeniowe [15].

Programy dedykowane na maszyny wieloprocesorowe znacząco różnią się od programów sekwencyjnych. Zagadnienia takie jak: zarządzanie wątkami czy wyszukiwanie zależności pomiędzy zadaniami nie należą do łatwych. W związku z tym zwykle nie jest wykorzystywana w pełni moc tkwiąca w maszynach wieloprocesorowych. Dla przeciętnego użytkownika bardziej naturalne i o wiele mniej skomplikowane wydaje się tworzenie rozwiązań sekwencyjnych w przeciwieństwie do rozwiązań równoległych. Koszt napisania programu efektywnie korzystającego z wielu wątków jest zazwyczaj dużo wyższy, niż koszt napisania takiego samego programu pracującego sekwencyjnie. Zatem konwersja programów sekwencyjnych na ich odpowiedniki równoległe powinna być w jak największym stopniu zautomatyzowana.

Proces tworzenia wydajnego oprogramowania jest czynnością czasochłonną i skomplikowaną, wymagającą od twórców gruntownego poznania zagadnień z zakresu przetwarzania równoległego. Tworzenie i projektowanie aplikacji równoległej wymusza uwzględnienie wielu, często wzajemnie wykluczających się czynników:

- docelowa architektura systemu,
- podział problemu (dekompozycja / partycjonowanie),
- komunikacja, koszt i liczba komunikatów, opóźnienia i przepustowość,
- synchronizacja – semafory, zamki, bariera, sekcja krytyczna, monitory,
- zależności danych i sterowania,
- ziarnistość obliczeń,
- zarządzanie wątkami i obciążeniem obliczeń,
- zbilansowanie obciążenia procesorów.

Powszechnie wiadomo, że statystycznie najwięcej czasu, jaki procesor potrzebuje do realizacji zadania zajmuje wykonanie instrukcji zawartych w pętlach programowych, składających się z wielu iteracji [15]. Transformacja tych obszarów programu może przynieść największe korzyści uwzględniając czas ich wykonania. Przekształcenie pętli poprzedza analiza zależności. Dzięki niej wiadomo, w jaki sposób iteracje bądź instrukcje pętli zależą od siebie w trakcie ich wykonywania. Wykonanie jednej lub całej sekwencji odpowiednich transformacji pozwala na wyeliminowanie zależności. Następnie należy dokonać podziału przestrzeni iteracji pętli. Proces analizy zależności oraz etap generowania kodu jest silnie złożony. Z tego powodu niezbędne jest opracowanie algorytmów umożliwiających automatyzację procesu zrównoleglenia pętli.

Zaproponowano wiele rozwiązań umożliwiających wykonanie automatycznej transformacji pętli [9], [18], [35], [38], [39], [41], [65], [66], [67], [85], [92], [93], jednak żadne z nich nie pozwala na wyszukanie pełnej równoległości zawartej w pętli programowej.

W niniejszej dysertacji zaprezentowano autorskie algorytmy pozwalające na wyznaczenie równoległych fragmentów kodu dla pętli programowych, w przypadkach gdy inne techniki zawodzą. Algorytmy te mogą być zastosowane dla pętli dowolnie zagnieżdżonych w przypadku, w których granice pętli podane są jako parametr.

Wyszukanie pełnej równoległości pętli programowej zazwyczaj wiąże się z określonym kosztem wykonania kodu równoległego. Są to narzuty obliczeniowe i pamięciowe, mechanizmy synchronizacji oraz komunikacji zadań. Ważne jest zatem zbadanie, czy korzyści czasowe wynikające ze zrównoleglenia pętli przewyższają koszty wykonania aplikacji równoległej.

Główną ideą proponowanego rozwiązania jest podział przestrzeni iteracji pętli na zbiór niezależnych fragmentów kodu [80]. **Fragment kodu** to zbiór wszystkich iteracji powiązanych zależnościami między iteracjami lub instrukcjami pętli. Fragment kodu nazywamy niezależnym, jeżeli nie istnieje żadna zależność pomiędzy jego iteracjami i iteracjami należącymi do innego fragmentu kodu. W celu znalezienia niezależnych fragmentów kodu wyznaczany jest zbiór ich początków. Następnie wyszukiwane są iteracje należące do tych fragmentów, w celu uzyskania pełnego kodu równoległego. Zaproponowano kilka rozwiązań przebierania ich iteracji dla dokładnego wyznaczenia fragmentów kodu. Jeżeli znaleziono tylko jeden początek, wyszukiwana jest równoległość w przestrzeni iteracji pojedynczego fragmentu kodu. Określany jest nowy zbiór zależnych fragmentów kodu, wymagających synchronizacji pomiędzy nimi.

Prezentowane algorytmy bazują na dokładnej analizie zależności opisanych przy pomocy relacji i zbiorów krotek [55].

W niniejszej pracy główny nacisk położony został na podział przestrzeni iteracji pętli umożliwiający wyszukanie jak największej równoległości w porównaniu z innymi, znanymi autorowi metodami. Wiele istniejących transformacji dedykowanych jest tylko dla określonego typu pętli programowych. Dla przykładu transformacje unimodularne [9], [92], [93] – przeznaczone są dla pętli idealnie zagnieżdżonych, czy też transformacje oparte na rekurencji Hamiltona (ang. *Hamiltonian Recurrence*) [41] - pętle o stałym wektorze dystansu. Dokonano próby zrównoleglenia pętli, dla których uważane aktualnie za najmocniejsze przekształcenia oparte na mapowaniu przestrzeni iteracji przy pomocy transformacji afinicznych (ang. *Affine Transformation*

Framework) [35], [38], [39], [65], [66], [67] oraz na podziale przestrzeni iteracji [80], [85], zawodzą.

Celem prowadzonych badań było wypełnienie luki poznawczej, poprzez opracowanie algorytmów umożliwiających automatyczne zrównoleglenie pętli programowych.

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

Wyniki badań zostały przedstawione w autorskich publikacjach w czasopismach i na konferencjach polskich [14], [28], [70], [76], [77], oraz międzynarodowych [16], [17], [19], [24], [27]. Opracowane algorytmy stanowią rozwinięcie zagadnień związanych z automatycznym zrównolegleniem pętli sekwencyjnych dla kompilatorów optymalizujących.

1.2. Cel i teza badawcza pracy

Istnieje wiele algorytmów umożliwiających przekształcenie pętli programowych, pozwalających na wyznaczenie równoległości w ograniczonym stopniu i zakresie. Brak jest natomiast metody uniwersalnej, efektywnej dla pętli o dowolnej postaci. Zwykle ograniczenia te dotyczą struktury pętli, typu zależności oraz poziomu równoległości. Zakres stosowalności rozwiązań w odniesieniu do struktury uzależniony jest od rodzaju zagnieżdżenia pętli, występowania granic w postaci sparametryzowanej i instrukcji sterujących. Skuteczne zastosowanie algorytmów uzależnione jest od możliwości akceptacji zależności jednorodnych (o stałym wektorze dystansu) i niejednorodnych (o zmiennym wektorze dystansu) oraz zależności o danym typie (proste, odwrotne lub po wyjściu). Ostatni rodzaj ograniczeń dotyczy ekstrakcji równoległości na poziomie pojedynczej iteracji. W licznych transformacjach np. unimodularnych instancje (instrukcje) pętli nie są rozpatrywane osobno.

Ponadto możliwość zastosowania algorytmów dla danej klasy pętli nie gwarantuje uzyskania pełnej równoległości. Istotny jest rozmiar części przestrzeni iteracji pętli, która musi zostać wykonana sekwencyjnie.

Zrównoleglenie pętli programowych może zostać zrealizowane na dwóch poziomach:

- równoległość na poziome iteracji – wszystkie instrukcje w ciele pętli traktowane są jako atomowa, niepodzielna i jednokrotnie wykonywana instancja bloku instrukcji. Takie podejście stosowane jest dla pętli idealnie

zagnieżdżonych. Zaletą tego rozwiązania jest dodatkowa możliwość eliminacji zależności,

- równoległość na poziomie instrukcji – jest to równoległość pomiędzy instancjami poszczególnych instrukcji pętli. Niepodzielnym i atomowym elementem w tym przypadku jest instancja pojedynczej instrukcji. Wyszukanie równoległości na poziomie instrukcji stosowane jest dla pętli idealnie i nieidealnie zagnieżdżonych. W tym podejściu niezbędna jest analiza zależności pomiędzy poszczególnymi instrukcjami pętli.

Zdaniem autora, wyzwaniem jest opracowanie uniwersalnego rozwiązania pozwalającego na transformację pętli o dowolnej strukturze zarówno na poziomie iteracji jak i instrukcji. Jest to niezbędny element efektywnego kompilatora, umożliwiającego automatyczną transformację sekwencyjnego kodu do równoległej postaci.

Celem niniejszej pracy jest opracowanie algorytmów pozwalających na zwiększenie ekstrakcji równoległości gruboziarnistej w pętlach programowych.

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

Możliwe jest opracowanie algorytmów automatycznego wyszukiwania równoległości gruboziarnistej w pętlach programowych, skuteczniejszych ze względu na stopień uzyskiwanej równoległości w porównaniu z rozwiązaniami opartymi na transformacjach afinicznych i metodach znajdowania niezależnych fragmentów kodu.

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 [35], [38], [39], [65], [66], [67]. Transformacje oparte na podziale przestrzeni iteracji pętli na zbiór niezależnych fragmentów kodu [80], [85] pozwalają na wyznaczenie kodu równoległego w sposób automatyczny.

Metody te nie umożliwiają jednak uzyskania pełnej równoległości dla określonych klasy pętli. Celem opracowania algorytmów zawartych w niniejszej pracy jest możliwość zwiększenia automatycznej ekstrakcji równoległości dla takich pętli.

Zwiększanie ekstrakcji równoległości rozumiane jest jako:

- zrównoleglenie pętli programowej, dla której inne znane autorowi rozwiązania zawodzą, tj. nie pozwalają na uzyskanie kodu równoległego,
- uzyskanie większej liczby równoległych fragmentów kodu.

Uzyskanie kodu równoległego dla pętli, w której inne metody zawodzą i odnotowanie przyspieszenia obliczeń pozwalają stwierdzić jednoznacznie, że zostało dokonane zwiększenie ekstrakcji równoległości. Zwiększenie ekstrakcji równoległości to również możliwość uzyskania większej liczby fragmentów kodu. Uzyskany kod równoległy cechuje się wówczas lepszym wskaźnikiem skalowalności. Możliwość wykonania większej ilości iteracji pętli równolegle wiąże się ze zmniejszeniem części sekwencyjnej. Zgodnie z prawem Amdahla [4] i mając na uwadze koszty czasowe zrównoleglenia, zmniejszenie części sekwencyjnej pozwala na uzyskanie większego przyspieszenia obliczeń. Potwierdza to istotę zwiększenia ekstrakcji równoległości w pętlach programowych.

Zagadnienia poruszane w niniejszej rozprawie doktorskiej stanowią kontynuację badań zawartych w pracy [85], w której zaproponowano algorytmy wyszukiwania drobno- i gruboziarnistej równoległości w pętlach programowych z zależnościami afinicznymi. Przeanalizowano ograniczenia tych rozwiązań (bazujących na podziale przestrzeni iteracji pętli) oraz opracowano algorytmy o większym zakresie zastosowania, umożliwiające zwiększenie ekstrakcji równoległości.

Proponowane algorytmy pozwalają na ekstrakcję równoległości gruboziarnistej. Ziarnistość obliczeń (ang. *granularity*) [63] wyznacza ilość obliczeń aplikacji między zdarzeniami synchronizacji lub komunikacji. Drobnoziarnista równoległość obliczeń (ang. *fine-grained parallelism*) odnotowywana jest w przypadku, gdy rozmiar zadań wykonywanych pomiędzy zdarzeniami synchronizacji lub komunikacji jest stosunkowo niewielki. Drobnoziarnistość umożliwia redukcję czasu przestojów oraz efektywne zarządzanie obciążeniem systemu wieloprocesorowego. Jednak koszt zarządzania wieloma wątkami może okazać się 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.

Gruboziarnista równoległość obliczeń (ang. *coarse-grained parallelism*) oznacza przypadek, gdy czas obliczeń wykonanych przez wątki jest dużo większy od czasu potrzebnego na zarządzanie wątkami. Zaletą takiego podziału jest możliwość zwiększenia wydajności algorytmu poprzez: redukcję czasu potrzebnego na synchronizację i komunikację, zwiększenie lokalności kodu oraz zmniejszenie zapotrzebowania na pamięć. Wadą gruboziarnistości jest ryzyko wystąpienia dużych czasów przestojów i różnej wielkości ziaren obliczeń, utrudniających zbalansowanie obciążenia na poszczególnych jednostkach przetwarzających. Gruboziarnisty podział preferowany jest w systemach, w których znaczące są koszty: tworzenia i zarządzania

zadaniami (wątkami) oraz komunikacji między nimi. Podział na drobno- i gruboziarnistość jest uzależniony od środowiska sprzętowo-programowego. Kod tego samego programu może w jednym środowisku cechować się drobnoziarnistością, a w innym środowisku zostać uznany jako gruboziarnisty.

Z opracowaniem algorytmów automatycznego zrównoleglenia pętli, umożliwiających zwiększenie ekstrakcji równoległości, wiążą się liczne korzyści, z których główne to:

- zwiększenie liczby zrównoleglonych pętli programowych w aplikacji,
- zwiększenie skalowalności kodu,
- przyspieszenie obliczeń,
- redukcja kosztów i błędów, związanych z ręczną transformacją kodu do równoległej postaci.

Za podjęciem zagadnienia będącego tematem niniejszej pracy przemawiają przesłanki naukowe i ekonomiczne. Przesłanki naukowe to próba odpowiedzi na następujące pytania:

- jaka jest liczba niezależnych fragmentów kodu w zadanej pętli programowej?
- jak zwiększyć stopień ekstrakcji równoległości w zadanej pętli programowej i jakie są koszty wykonania takiego kodu równoległego?

Fragment kodu to zbiór instrukcji powiązanych ze sobą poprzez zależności. Korzystanie z wspólnych komórek pamięci przez poszczególne instrukcje pojedynczego fragmentu, 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.

W pracy poruszono problem przebierania iteracji fragmentów kodu w czasie wykonania programu. Dokonano próby odpowiedzi, czy wyznaczenie następnej iteracji w czasie wykonania umożliwia zwiększenie stopnia równoległości oraz jakim kosztem. Powodem zbadania tego zagadnienia są ograniczenia narzędzi programowych do automatycznego wyznaczenia zbioru iteracji fragmentu kodu.

Fragment kodu jest niezależny, gdy nie istnieją zależności pomiędzy jego instrukcjami i instrukcjami należących do innych fragmentów kodu. Ekstrakcja równoległości zaczyna się od znalezienia zbioru niezależnych fragmentów kodu w celu wyznaczenia całej równoległości na poziomie fragmentów. Jeżeli jest to niemożliwe, pozostaje możliwość wykonania równoległego kodu złożonego ze zbioru zależnych fragmentów kodu. W tym celu należy dokonać synchronizacji pomiędzy nimi oraz

oszacować jej koszt. W pracy zbadano problem ekstrakcji równoległości z synchronizacją.

Przesłanki ekonomiczne poruszonego zagadnienia to możliwość automatyzacji całego procesu transformacji kodu sekwencyjnego na odpowiednik równoległy. Wiąże się z tym zmniejszenie kosztów oraz redukcja błędów ludzkich związanych z ręcznym przekształcaniem kodu. Poza tym minimalny udział użytkownika w procesie zrównoleglenia kodu przyczynia się do wzrostu popularności systemów wieloprocesorowych w rozwiązaniach biznesowych. W ostatnich latach producenci oprogramowania ogólnodostępnego i komercyjnego coraz częściej wykorzystują w swoich aplikacjach kod równoległy w celu przyspieszenia obliczeń. Przetwarzanie równoległe znajduje zastosowanie w popularnych aplikacjach graficznych, programach do przetwarzania dokumentów czy pakietach biurowych. Niewątpliwie wiąże to się z faktem wzrostu popularności procesorów wielordzeniowych, montowanych w komputerach osobistych i przenośnych.

Opracowane algorytmy mają szeroki zakres stosowalności. Umożliwiają zrównoleglenie dowolnie zagnieżdżonych pętli programowych. Granice pętli mogą być określone w postaci parametrów. Możliwe jest zrównoleglenie pętli z zależnościami jednorodnymi i niejednorodnymi. Dopuszczalna jest obecność iteracji, należących do różnych zależności. Możliwa jest ekstrakcja równoległości kodu, w przypadku gdy w przestrzeni iteracji pętli zawiera się tylko jeden niezależny fragment kodu.

Dane wejściowe algorytmów wyszukiwania gruboziarnistej równoległości stanowi kod źródłowy zapisany w języku Petit [54]. Za pomocą narzędzia Petit wyznaczono także zbiór relacji zależności pętli wejściowej. Modyfikacje na relacjach przeprowadzono przy pomocy pakietu Omega Calculator [56]. Powyższe narzędzia stanowią integralną część projektu Omega opracowanego na Uniwersytecie Maryland (ang. *University of Maryland*). W celu wygenerowania kodu równoległego zaimplementowany został autorski generator kodu [75], przetwarzający wyniki z narzędzia Omega Calculator. Kod równoległy zapisany został zgodnie ze standardem programowania równoległego OpenMP [31]. Wygenerowana pętla równoległa może zostać zaimplementowana także w innym środowisku, umożliwiającym obliczenia równoległe, np. PThreads [30]. Badania przeprowadzono na maszynie wieloprocesorowej z pamięcią dzieloną Workstation Board Intel Xeon S5000XVN (8 jednostek przetwarzających - dwa procesory czterordzeniowe Intel Xeon 1.6 GHz Quad-Core), z wykorzystaniem kompilatora GNU Compiler Collection [42].

Zweryfikowano poprawność i jakość otrzymanego kodu równoległego. Dokonano porównania pętli równoległych z ich wersją sekwencyjną dla 1, 2 ,4 i 8 wątków oraz

różnych wartości parametrów granic pętli. Pozwoliło to na ustalenie poprawności oraz określenie jakości uzyskanych wyników (przyspieszenie, efektywność, skalowalność).

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. Omówione zostały podstawowe pojęcia związane z analizą zależności i ziarnistością obliczeń. Przedstawiono wskaźniki wykorzystywane do określania jakości kodu oraz model projektowania algorytmów równoległych. Wyjaśniono podstawy arytmetyki Presburgera, która jest niezbędna do zrozumienia reprezentacji zależności oraz mechanizmu działania przedstawionych algorytmów.

Kolejny rozdział zawiera opis zaproponowanych algorytmów wraz z przykładami obrazującymi ich działanie. Dodatkowo opis ten poprzedzono niezbędną wiedzą umożliwiającą zrozumienie proponowanych rozwiązań.

W rozdziale 4 zaprezentowano badania eksperymentalne przeprowadzone przy pomocy autorskiego narzędzia, opracowanego na potrzeby badań zgodnie z 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. W rozdziale tym przedstawiono zestaw pętli, dla której transformacje afiniczne (jedne z najmocniejszych obecnie transformacji) zawodzą, natomiast zaproponowane algorytmy powalają na uzyskanie kodu równoległego. Dokonano także porównania zakresu stosowalności rozwiązań z algorytmami automatycznego podziału przestrzeni iteracji, przedstawionych w pracy [85], z wykorzystaniem wybranego zestawu pętli testowych.

W ostatnim rozdziale zawarto wnioski końcowe dotyczące realizacji podjętych w pracy zagadnień.

2. PRZEGLĄD ZAGADNIEŃ Z ZAKRESU PRZETWARZANIA RÓWNOLEGŁEGO

Przetwarzanie równoległe staje się coraz bardziej istotne w dziedzinie technik tworzenia oprogramowania. Nieustanny wzrost ilości danych koniecznych do przetworzenia wiąże się nierozerwalnie z ciągłym wzrostem zapotrzebowania na moc obliczeniową systemów informatycznych. Ograniczenia technologiczne (niemożliwość dalszej miniaturyzacji czy emisja ciepła) nie pozwalają na przyspieszanie taktowania procesorów w nieskończoność. Rozwiązaniem są systemy wieloprocesorowe w różnej postaci: sieci komputerowych, klastrów, maszyn wieloprocesorowych czy też komputera osobistego wyposażonego w procesor wielordzeniowy. System równoległy to także specjalne oprogramowanie i kompilatory, które umożliwiają podział problemu obliczeniowego i wykonanie kodu równolegle przez wiele jednostek przetwarzających [71]. Najwięcej obliczeń aplikacji zwykle znajduje się w pętlach programowych [15]. Stąd ważnym zagadnieniem jest zrównoleglenie pętli, które wiąże się z koniecznością uwzględnienia wielu dodatkowych czynników takich jak:

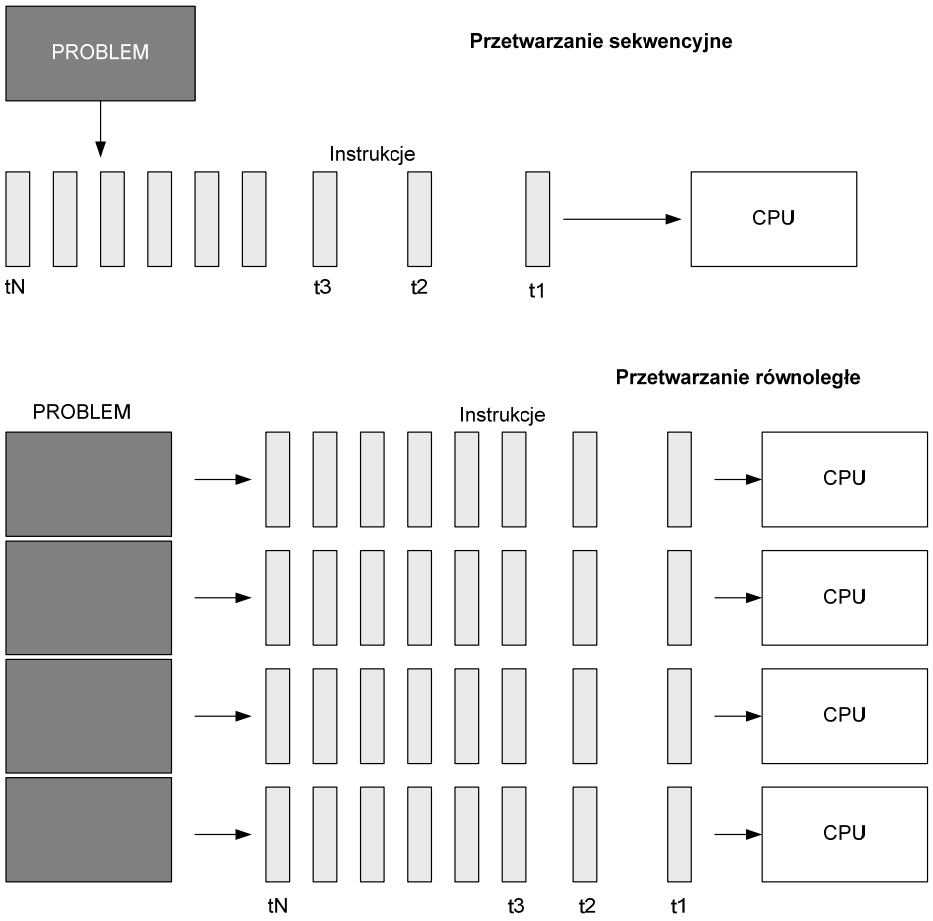
- zależności między danymi,
- automatyzacja procesu zrównoleglenia,
- ziarnistości kodu,
- docelowa architektura,
- aglomeracja i mapowanie.

Niniejszy rozdział przedstawia zagadnienia związane z przetwarzaniem równoległym i zrównolegleniem pętli programowych. Omówione zostanie rodzaje pętli programowych, zależności w pętlach, ziarnistość kodu, wskaźniki jakości i model projektowania algorytmów równoległych. W celu zrozumienia proponowanych w niniejszej dysertacji rozwiązań przybliżona zostanie arytmetyka Presburgera, za pomocą

której reprezentowane są zależności w pętlach. Operacje arytmetyki Presburgera są podstawą aparatu matematycznego proponowanych algorytmów automatycznego zrównoleglenia pętli.

2.1. Definicja przetwarzanie równoległego

Przetwarzaniem równoległym (ang. *parallel computing*) określa się współdziałanie wielu autonomicznych jednostek przetwarzających tak, aby współpracowały w rozwiązywaniu wspólnego problemu [15]. Operacje należące do pewnego rodzaju obliczeń są przeprowadzane w tym samym czasie (tzn. przeprowadzanie jednej operacji rozpoczyna się równocześnie z drugą operacją lub przed jej zakończeniem). Na rysunku 2.1 przedstawiono rozwiązywanie problemu sekwencyjnie (za pomocą jednego procesora) i równolegle (za pomocą wielu procesorów). W celu rozwiązania problemu w sposób równoległy należy dokonać jego podziału na mniejsze zadania.



Rys. 2.1. Przetwarzanie sekwencyjne i równoległe [10].

Przetwarzanie równoległe znajduje zastosowanie w wielu dziedzinach: grafika, kryptografia, różnego rodzaju symulacje z zakresu meteorologii, astronomii, medycyny, sejsmografii czy budownictwa [2], [91].

Równoległe przeprowadzanie operacji może w ogólności prowadzić do znacznego **przyspieszenia obliczeń**, lecz jednocześnie powoduje powstawanie dodatkowych problemów, wymagających rozwiązania. Jednym z podstawowych ograniczeń związanych z równoległym przetwarzaniem danych jest to, że dane potrzebne do przeprowadzenia jednej operacji mogą być uzyskiwane w wyniku przeprowadzenia innej operacji i dlatego tych dwóch operacji nie można przeprowadzić w tym samym czasie [36]. Dostęp do wspólnych danych i ich modyfikacja przez wiele równoległych wątków prowadzi w ogólnym przypadku do występowania **zależności danych**. Ich uwzględnienie w trakcie procesu zrównoleglenia jest niezbędne dla uzyskania poprawności kodu równoległego [78]. Kod równoległy jest poprawny jeżeli wynik jego wykonania z wykorzystaniem dowolnej liczby procesorów jest równy wynikowi wykonania kodu sekwencyjnego. Jedną z cech poprawności jest determinizm. 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.

Większość obliczeń zawarta jest w pętlach programowych aplikacji. Zrównoleglenie pętli z uwzględnieniem zależności danych jest kluczowe dla przyspieszenia wykonania obliczeń [60].

2.2. Zrównoleglenie pętli programowej

W celu zrównoleglenia pętli programowej należy dokonać analizy zależności. Cenna jest także wiedza o właściwościach pętli dla zastosowania dalej odpowiednich transformacji pozwalających na uzyskanie kodu równoległego. W punkcie 2.2.1 przybliżono definicje i rodzaje zależności, w 2.2.2 przedstawiono natomiast rodzaje pętli programowych.

2.2.1. Zależności w pętlach programowych

W przypadku występowania zależności niemożliwe jest bezpośrednie zrównoleglenie wybranego fragmentu kodu. Ze względu na rodzaj występujących

zależności wyróżniamy dwa typy ograniczeń uniemożliwiających bezpośrednie zrównoleglenie fragmentu kodu:

- zależność sterowania,
- zależność danych.

Zależność sterowania (ang. *control dependence*) pomiędzy instrukcją S1 i instrukcją S2 ma miejsce wówczas, jeżeli instrukcja S1 decyduje, czy instrukcja S2 będzie wykonywana [68], tak jak to ilustruje poniższy przykład:

```
S1: if(x==0)
S2:  y = 0;
```

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], [68]:

1. Zależność przepływu danych zwana również **zależnością prostą** (ang. *Data - Flow Dependence, True Dependence*) [1], [68] - występuje między instrukcjami S1 i S2 ($S1 \prec S2$), jeżeli zapis danych następuje w instrukcji poprzedzającej ich odczyt.

   ```
   S1:  X = ...
   S2 : ... = X.
   ```

   Zależność ta oznacza, że instrukcja S2 pobiera wartość obliczoną za pomocą instrukcji S1 i oznaczana jest: $S1\ \delta\ S2$ (odczyt S2 uzależniony jest od S1).
2. **Zależność odwrotna** (ang. *Antidependece*) [1], [68] - występuje między instrukcjami S1 i S2 ($S1 \prec S2$), jeżeli odczyt danych następuje w instrukcji poprzedzającej ich zapis.

   ```
   S1:  ... = X
   S2:  X = ....
   ```

   Zależność ta zapobiega zamianie S1 z S2, która to mogłaby doprowadzić do wadliwego użycia wartości obliczonej za pomocą instrukcji S2. Zależność odwrotna (zwana też antyzależnością) oznaczana jest: $S1\ \delta^{-1}\ S2$ (lub $S1\ \delta^{-}S2$).
3. **Zależność po wyjściu** (ang. *Output Dependence*) [1], [68] - występuje między instrukcjami S1 i S2 ($S1 \prec S2$), jeżeli zapis danych wykonywany jest w obydwu instrukcjach.

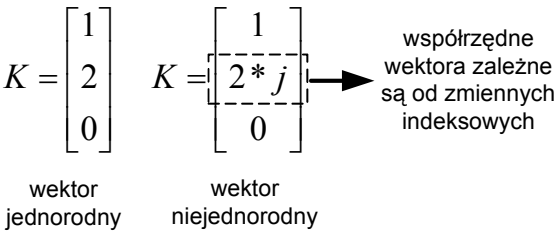
S1: X = ...

S2: X = ...

Zależność ta zapobiega zamianie, w wyniku której kolejna instrukcja mogłaby odczytywać błędną wartość i oznaczana jest: S1 δ^0 S2.

Rozpatrywane powyżej zależności danych mogą występować w ramach pojedynczej iteracji pętli, albo pomiędzy kolejnymi iteracjami pętli. W pierwszym przypadku mamy do czynienia z **zależnością niezależną od pętli** (ang. *loop-independent dependence*), natomiast w drugim z **zależnością przenoszoną pętlą** (ang. *loop-carried dependence*) [68].

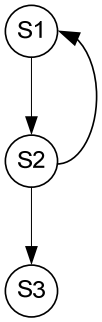
Zależność jednorodna *(*ang. *uniform depence)* **–** jest to zależność, którą można przedstawić za pomocą wektora dystansu o stałych elementach. Jeśli współrzędne wektora dystansu są zależne od zmiennych indeksowych pętli to mówimy o **zależności niejednorodnej** *(*ang. *non-uniform depence)* – rys.2.2.



Rys. 2.2. Rodzaje wektorów dystansu [1].

Ze względu na występowanie zależności, w wielu przypadkach niemożliwe jest wykonanie kodu w sposób równoległy bez uprzedniej analizy, a następnie redukcji zależności.

W celu ułatwienia uzyskania kodu równoległego, po otrzymaniu zbioru zależności pętli programowej tworzony jest tzw. **zredukowany graf zależności** (ang. *reduced dependence graph - RDG*) [85]. Jest to graf skierowany G=(V, E). Jego krawędzie zdefiniowane są przez zależności. Wierzchołki natomiast oznaczają instrukcje ciała pętli, pomiędzy którymi te zależności zachodzą. Następnie dla danego grafu wyszukiwane są wszystkie silnie spójnie składowe (ang. *strongly connected components – SCCs*). Silnie spójnia składowa grafu skierowanego G to taki maksymalny podgraf S=(V,E), w którym dla każdego wierzchołka $u \in V$ oraz $u \notin V'$, nie istnieje wierzchołek $v \in V'$, dla którego $(u,v) \in E$. Innymi słowy, pomiędzy każdymi dwoma wierzchołkami podgrafu S istnieje skierowane połączenie. Na rysunku 2.3 przedstawiono przykład grafu złożonego z dwóch silnie spójni składowych, {S1,S2} i {S3}.



Rys. 2.3. Przykład SCCs [13].

Z przeprowadzonych obserwacji wynika, iż 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. Podział grafu na SCCs pozwala na zwiększenie zakresu stosowalności proponowanych algorytmów w niniejszej pracy.

2.2.2. Klasyfikacje pętli programowych

Algorytmy zrównoleglenia pętli programowych dedykowane są do pewnych klas pętli programowych. Oznacza to, że mogą istnieć rodzaje pętli, dla których dany algorytm nie dokona zrównoleglenia. Klasyfikacja pętli pozwala na porównanie rozwiązań i ocenę ich zakresu stosowalności. Charakterystyka pętli określona jest przez stopień i typ jej zagnieżdżenia, obecność sparametryzowanych granic oraz rodzaje zależności.

Pętla idealnie zagnieżdżona (ang. *perfectly nested loop*) 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 loop*) [32]. Poniżej pokazano przykłady takich pętli.

for i=1 to M
  for j=1 to N {
  instrukcja 1
  instrukcja 2
}
	for i=1 to M {
	  instrukcja 1
	  for j=1 to N {
	  instrukcja 2
	  }
	}

Pętla idealnie zagnieżdżona	*Pętla nieidealnie zagnieżdżona*

Pętla sparametryzowana *(*ang. *parameterizied loop*) jest to pętla, której granice określone są przez parametr. Innymi słowy wartość granic w trakcie kompilacji jest nie znana. Jeśli wszystkie granice określone są za pomocą liczb jest to pętla **niesparametryzowana** (ang. *unparameterizied loop*). Poniżej pokazano przykłady takich pętli.

for i=1 to 10
for j=1 to 10 {
instrukcja 1
}
	for i=1 to M
	for j=1 to N {
	instrukcja 1
	}

Pętla niesparametryzowana	*Pętla sparametryzowana*

Pętlę zawierającą wyłącznie jednorodne zależności nazywamy **pętlą jednorodną** *(ang. uniform loop)*; jeśli występują zależności niejednorodne to jest to przypadek **pętli niejednorodnej** *(ang. non-uniform loop)*. W poniższych przykładach dla pierwszej pętli istnieje jednorodny wektor dystansu [0,1]. W drugim przypadku zależność danych posiada niejednorodny wektor dystansu [0,i], gdzie $1\text{-}N \leq i < N$.

for i=1 to N
for j=1 to N {
a[i][j] = a[i][j+1]
}
	for i=1 to N
	for j=1 to N {
	a[i][j] = a[i][j+i]
	}

Pętla jednorodna	*Pętla niejednorodna*

Łatwiejszym zadaniem w ogólnym przypadku jest zrównoleglenie pętli idealnie zagnieżdżonej niż nieidealnie zagnieżdżonej. Wszystkie instrukcje pętli tej samej iteracji można potraktować jako jedną i dokonać próby dodatkowej redukcji zależności [64]. Uzyskanie kodu równoległego pętli z zależnościami jednorodnymi jest zazwyczaj mniej skomplikowane, niż w przypadku obecności zależności niejednorodnych. Zrównoleglenie pętli sparametryzowanej jest trudniejsze niż pętli o stałych, znanych granicach.

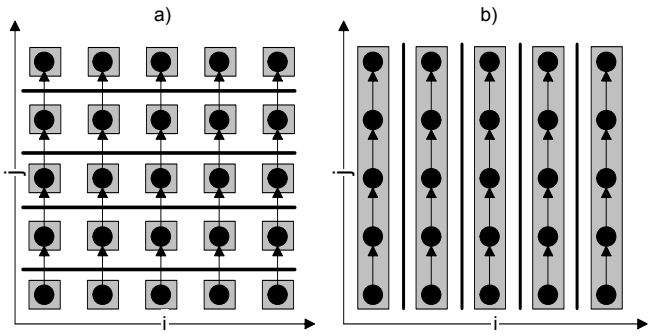
Zaproponowane rozwiązania w niniejszej dysertacji pozwalają na zrównoleglenie pętli jednorodnych i niejednorodnych, dowolnie zagnieżdżonych z granicami stałymi i sparametryzowanymi.

2.3. Ziarnistość kodu

W przetwarzaniu równoległym **ziarnistość kodu** (ang. *granularity*) wyznacza ilość obliczeń programu pomiędzy zdarzeniami synchronizacji lub komunikacji. Ziarnistość obliczeń jest ważnym wskaźnikiem w przetwarzaniu równoległym, mającym zasadnicze znaczenie w określaniu, w jakim stopniu obciążenie programu może być zbalansowane na maszynie równoległej. 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 [45]. Liczba oraz wielkość zadań, na jakie problem został podzielony określają **ziarnistość dekompozycji**. Ze względu na wielkość ziarna obliczeń w przetwarzaniu równoległym wyróżnia się podział grubo- i drobnoziarnisty [15].

W przypadku gdy stosunek czasu obliczeń aplikacji do czasu potrzebnego na synchronizację i komunikację jest mniejszy oraz ilość obliczeń pomiędzy punktami synchronizacji jest stosunkowo niewielka, to mówimy o **drobnoziarnistej równoległości** (ang. *fine-grained parallelism*) – rys. 2.4.a. Drobnoziarnistość cechuje możliwość maksymalnej redukcji czasów przestojów i efektywnego planowania obciążenia. Jednak narzut czasowy potrzebny do tworzenia i unicestwiania, komunikacji i synchronizacji wątków może być zbyt kosztowny i całkowicie obniżyć wydajność przetwarzania. W związku z tym podział kodu na drobne ziarna preferowany jest w systemach, gdzie koszt komunikacji między jednostkami przetwarzającymi jest relatywnie mały oraz planowanie obciążenia jest bardzo ważne [33].

Gruboziarnista równoległość pozwala na redukcję kosztów zarządzania wieloma zadaniami oraz zmniejszenie zapotrzebowania na pamięć poprzez zwiększenie lokalności kodu [83]. 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).



Rys 2.4. Drobnoziarnista (a) i gruboziarnista (b) równoległość – rys. poglądowy [28].

Dobranie właściwego stopnia ziarnistości kodu równoległego uzależnione jest architekturą programową i sprzętową. Warunkiem koniecznym jest uzyskanie krótszego czasu wykonania kodu równoległego niż czas wykonania jego sekwencyjnego odpowiednika:

$$t_{równ} < t_{sekw},$$

gdzie: t_{sekw} to czas wykonania kodu sekwencyjnego, $t_{równ}$ – czas wykonania kodu równoległego.

Uwzględniając dodatkowy czas niezbędny do obsługi kodu równoległego t_{dod} tj. czas synchronizacji, komunikacji czy zarządzania wątkami oraz wykonanie kodu równoległego na P procesorach, uzyskuje się następującą nierówność:

$$\frac{t_{sekw}}{P} + t_{dod} < t_{sekw}.$$

Na podstawie powyższego wzoru można stwierdzić, że przy ustalaniu ziarnistości kodu ważne jest oszacowanie czasu dodatkowego na obsługę równoległości i sprawdzenie czy jest prawdziwa poniższa nierówność:

$$t_{dod} < (P-1) * t_{sekw}.$$

Spełnienie powyższego warunku gwarantuje uzyskanie wartości przyspieszenia obliczeń większej niż 1.

W zależności od typu użytej transformacji pętli możliwe jest uzyskanie pożądanej ziarnistości kodu dla wybranej architektury [1], [7], [52]. Do transformacji pozwalających na uzyskanie drobnoziarnistego kodu zalicza się: rozszerzenie skalaru (ang. *scalar expansion*), zmianę nazw zmienny skalarnych (ang. *scalar renaming*), zmianę nazw zmiennych tablicowych (ang. *array renaming*), podział węzłów (ang. *node splitting*), redukcję (ang. *reduction*), podział zmiennych indeksowych (ang. *index-set splitting*). Z kolei jako transformacje pozwalające uzyskać kod gruboziarnisty wyróżnia się: prywatyzację (ang. *privatization*), podział pętli (ang. *loop distribution*), wyrównanie (ang. *alignment*), replikację kodu (ang. *code replication*), łączenie pętli

(ang. *loop fusion*), odwrócenie wykonania pętli (ang. *loop reversal*), wielowymiarowe łączenie pętli (ang. *multilevel loop fusion*). Wynikiem transformacji zmiany kolejności wykonani pętli (ang. *loop interchange*) i przekoszenia pętli (ang. *loop skewing*) może być zarówno kod drobno- jak i gruboziarnisty [1].

W ogólnym przypadku ziarnistość kodu jest pojęciem względnym, ściśle związanym z określona architekturą systemu wieloprocesorowego [64]. Kod wykonany na systemie z pamięcią dzieloną sklasyfikowany jako gruboziarnisty, może zostać uznany za drobnoziarnisty w systemie z pamięcią rozproszoną.

2.4. Wskaźniki jakości algorytmów równoległych

Jednym z najbardziej istotnych celów przetwarzania równoległego jest maksymalne wykorzystanie możliwości obliczeniowych komputerów wieloprocesorowych i wynikające z tego zwiększenie szybkości wykonywania aplikacji [49]. Analiza jakości zrównoleglenia to odpowiedź na następujące pytania:

- czy czas wykonania aplikacji równoległej jest krótszy od sekwencyjnego odpowiednika?
- jakie korzyści czasowe wynikają z użycia większej liczby procesorów? Czy rozwiązanie jest skalowalne?
- jak efektywnie jest wykorzystana dostępna liczba procesorów?
- jakie są ograniczenia korzyści czasowych wynikających z zrównoleglenia aplikacji i jaka jest ich zależność od rozmiaru sekwencyjnej części aplikacji?

W przypadku oceny efektywności programu sekwencyjnego bierze się z reguły pod uwagę czas wykonania obliczeń w odniesieniu do rozmiaru danych wejściowych zadania. W przypadku aplikacji równoległych dodatkowym czynnikiem wpływającym w zasadniczy sposób na czas wykonywania obliczeń równoległych jest architektura komputera równoległego. Z tego właśnie względu wprowadzono pojęcie **systemu równoległego,** jako kombinacji programu równoległego i architektury równoległej.

Czas wykonania obliczeń równoległych (T_P) dla systemu P procesorowego to czas, jaki upływa od rozpoczęcia wykonywania programu na pierwszym procesorze do momentu, kiedy ostatni z procesorów zakończy jego wykonanie [4].

Przyśpieszenie (ang. *speedup*) jest to współczynnik przyśpieszenia obliczeń równoległych (S_P) wykonanych na P procesorach określony poniższym wzorem [69], [84]:

$$S_P = \frac{T_1}{T_P},$$

gdzie T_1 oznacza czas wykonania obliczeń na pojedynczym procesorze, natomiast T_P to czas potrzebny do wykonania tych samych obliczeń na P procesorach. Przyspieszenie jest miarą przydatną głównie do oceny stopnia zrównoleglenia aplikacji.

Ze względu na czas niezbędny do tworzenia, synchronizacji, komunikacji zachodzącej między wątkami jak i innymi czynnikami opóźniającymi, **przyśpieszenie liniowe** jest zazwyczaj mniejsze niż liczba procesorów: $S_P = a*P$, gdzie $a < 1$. W przypadku gdy współczynnik przyśpieszenia jest równy liczbie procesorów ($S_P = P$) mówimy o **przyspieszeniu doskonale liniowym**. Jeżeli przyspieszenie jest większe od liczby użytych procesorów do wykonania obliczeń ($S_P > P$), mamy do czynienia z tzw. **przyśpieszeniem superliniowym** (ang. *superlinear speedup*) [69].

Efektywność (ang. efficiency) E_P obliczeń równoległych jest to stosunek współczynnika przyspieszenia (S_P) do liczby użytych procesorów P:

$$E_P = \frac{S_P}{P} = \frac{T_1}{P * T_P}.$$

Efektywność jest doskonałą miarą wyrażania stopnia wykorzystania systemu równoległego, gdyż ilość procesorów P jest równa przyspieszeniu idealnemu S_P. Niska wartość efektywności odzwierciedla zatem niewykorzystanie mocy obliczeniowej procesorów systemu równoległego. Z uwagi na to, że przyspieszenie jest ograniczone liczbą procesorów P (poza superliniowym), $S_P \leq P$, wartość efektywności ograniczona jest w przedziale $0 \leq E_p \leq 1$.

W przypadku programu równoległego składającego się z dwóch składników-części równoległej oraz części sekwencyjnej stosuje się **prawo Amdahla**, które mówi, że rozmiar części sekwencyjnej algorytmu równoległego ogranicza przyspieszenie obliczeń równoległych, niezależnie od liczby procesorów użytych do przetwarzania. Zatem nie jest możliwe przyspieszanie wykonywania programu w nieskończoność, a nadmierne zwiększenie liczby procesorów powoduje jedynie spadek efektywności systemu, zgodnie z poniższym wzorem [4]:

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

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 $\frac{1}{f_s}$.

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. Wówczas przyspieszenie określone jest następującym wzorem [47]:

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

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

Istotnym, kryterium oceny jakości i przydatności aplikacji równoległej jest jej **skalowalność**. Przez skalowalność systemu równoległego rozumie zdolność systemu do efektywnego użycia rosnącej liczby procesorów [46], [59]. Analiza skalowalności pozwala na dobranie najlepszego dla danego problemu systemu równoległego z uwzględnieniem ograniczeń na rozmiar problemu i ilości dostępnych procesorów.

Do oceny właściwości systemów równoległych używanych jest wiele metryk. Dobry ich przegląd można znaleźć w pracy [84]. Dla praktycznego zobrazowania jakości stosuje się jednak jedynie wybrane wskaźniki. Zdaniem autora, dominującą metryką pozostaje czas wykonania obliczenia, a wszystkie inne metryki mogą być rozważane tylko w powiązaniu z tą pierwszą. W niniejszej rozprawie, autor ogranicza się do podania rzeczywistego czas wykonania programu, przedstawia przyspieszenie i efektywność oraz bada skalowalność uzyskanego kodu równoległego.

2.5 Arytmetyka Presburgera

Do opisu zależności oraz implementacji algorytmów wybrana została analiza zależności zaproponowana przez Pugha oraz Wonncotta [81]. Zależności reprezentowane są przez relacje zależności zbudowane z formuł Presburgera. Formuły te, które są podstawą **arytmetyki Presburgera** składają się z liniowych ograniczeń nad

zmiennymi całkowitymi przy użyciu logicznych połączeń negacji, koniunkcji i alternatywy; $\neg$, $\wedge$, $\vee$, kwantyfikatora ogólnego (ang. *universal quantifier*) $\forall$ i kwantyfikatora szczegółowego (ang. *existantial quantifier*) $\exists$ [81]. W arytmetyce Presburgera wyróżnia się także zestaw operacji binarnych i unarnych, które przeprowadza się na zbiorach i relacjach.

2.5.1 Relacje i zbiory

W celu zrozumienia dalszej części pracy niezbędne jest wyjaśnienie podstawowych pojęć związanych z formułami Presburgera jak i wybranymi narzędziami [55]:

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

Za pomocą relacji uzyskuje się zwięzły opis dokładnych zależności w porównaniu z tradycyjnym opisem wektora zależności lub odległości, które nie opisują dokładnie zależności danych. Natomiast za pomocą zbioru można opisać iteracje pętli, pomiędzy którymi występują zależności. Przykład:

for i=1 to N do
for j=1 to M do
a(i,j) = a(i,j-1)
endfor
endfor
	Relacja zależności
	R := {[i,j] -> [i,j+1] : 1 <= i <= n && 1 <= j < n}
	Zależne iteracje pętli
	IS := {[i,j] : 1 <= i <= n && 1 <= j <= n}

W dalszej części tego rozdziału opisano operacje arytmetyki Presburgera wraz z przykładami, które są podstawą prezentowanych w niniejszej pracy algorytmów. Operacje te przedstawione są za pomocą notacji matematycznej, natomiast w przykładach użyty jest zapis relacji i zbiorów charakterystyczny dla narzędzi Omega Calculator i Petit [56]. W punkcie 2.6 przedstawiono krótki opis tych narzędzi.

2.5.2. Operacje binarne

Operacje binarne wymagają dwóch argumentów. Argumentami są relacje lub zbiory.

Alternatywa

Operacja alternatywy (ang. *union*) można wykonać zarówno na zbiorach i relacjach. Wymiary argumentów muszą być jednakowe. Jeżeli $r_1 = \{x_1 \to y_1 \mid f_1(x_1, y_1)\}$ oraz $r_2 = \{x_2 \to y_2 \mid f_2(x_2, y_2)\}$ wówczas alternatywą r_1 i r_2 jest wyrażenie $\{x \to y \mid f_1(x,y) \vee f_2(x,y)\}$.

Przykład 1:

Niech będą dane dwa zbiory:

S1 := {[i] : 1 <= i <= 20}

S2 := {[i] : 15 <= i <= 30}

Wynikiem operacji alternatywy S1 $\cup$ S2 jest zbiór: S:= {[i] : 1 <= i <= 30}

Przykład 2:

Niech dane będą dwie relacje:

R1 := {[i,j] -> [i,j+1] : 1 <= i <= n && 1 <= j < n}

R2 := {[i,j] -> [i+1,j] : 1 <= i < n && 1 <= j <= n}

Wynikiem alternatywy relacji R1 $\cup$ R2 jest nowa relacja:

R := {[i,j] -> [i,j+1] : 1 <= i <= n && 1 <= j < n} union {[i,j] -> [i+1,j] : 1 <= i < n && 1 <= j <= n}

Koniunkcja

Operacja koniunkcji (ang. *intersection*) można wykonać zarówno na zbiorach i relacjach. Wymiary argumentów muszą być jednakowe. Jeżeli $r_1 = \{x_1 \to y_1 \mid f_1(x_1, y_1)\}$ oraz $r_2 = \{x_2 \to y_2 \mid f_2(x_2, y_2)\}$ wówczas koniunkcją (przecięciem) r_1 i r_2 jest wyrażenie $\{x \to y \mid f_1(x,y) \wedge f_2(x,y)\}$.

Przykład 1:

Niech będą dane dwa zbiory:

S1 := {[i] : 1 <= i <= 20}

S2 := {[i] : 15 <= i <= 30}

Wynikiem operacji koniunkcji S1 $\cap$ S2 jest zbiór: S:= {[i] : 15 <= i <= 20}

Przykład 2:

Niech dane będą dwie relacje:

R1 := {[i,j] -> [i,j+1] : 1 <= i <= n && 1 <= j < n}

R2 := {[i,j] -> [i+1,j] : 1 <= i < n && 1 <= j <= n}

Wynikiem koniunkcji relacji R1 $\cap$ R2 jest pusta relacja R = ϕ.

W zapisie relacji i zbiorów Omega Calculator koniunkcję ograniczeń oznacza się symbolem *&&*, alternatywę symbolami *OR* lub *union,* lub ||.

Różnica

Operacja różnicy (ang. *difference*) można wykonać zarówno na zbiorach i relacjach. Wymiary argumentów muszą być jednakowe. Jeżeli $r_1 = \{x_1 \to y_1 \mid f_1(x_1, y_1)\}$ oraz $r_2 = \{x_2 \to y_2 \mid f_2(x_2, y_2)\}$ wówczas różnicą r_1 i r_2 jest wyrażenie $\{x \to y \mid f_1(x, y) \wedge \neg f_2(x, y)\}$.

Przykład 1:

Niech będą dane dwa zbiory:

```
S1 := {[i] : 1 <= i <= 20}
S2 := {[i] : 15 <= i <= 30}
```

Wynikiem operacji koniunkcji S1 - S2 jest zbiór: S:= {[i] : 1 <= i < 15}

Przykład 2:

Niech dane będą dwie relacje:

```
R1 := {[i,j] -> [i,j+1] : 1 <= i <= n && 1 <= j < n}
R2 := {[i,j] -> [i+1,j] : 1 <= i < n && 1 <= j <= n}
```

Wynikiem koniunkcji relacji R1 - R2 jest nowa relacja:

```
R := {[i,j] -> [i,j+1] : 1 <= i <= n && 1 <= j < n}
```

Ograniczenie dziedziny

Operacja ograniczenia dziedziny (ang. *restrict domain*) posiada dwa argumenty; pierwszy z nich to relacja, drugi zbiór. Wymiar zmiennych wejściowych relacji musi być równy wymiarowi zbioru. Jeżeli $r_1 = \{x_1 \to y_1 \mid f_1(x_1, y_1)\}$ oraz $r_2 = \{x_2 \mid f_2(x_2)\}$ wówczas ograniczeniem dziedziny relacji r_1 na zbiór r_2 jest relacja $\{x \to y \mid f_1(x, y) \wedge f_2(x)\}$.

Przykład:

```
S := {[i,j] : 1<=i<=10 && 1<=j<=10}
R := {[i,j] -> [i,j+1] : 1 <= i <= 100 && 1 <= j <= 100}
```

Wynikiem operacji R \ S jest relacja R1:

```
R1 := {[i,j] -> [i,j+1] : 1 <= i <= 10 && 1 <= j <= 10}
```

Ograniczenie zakresu

Operacja ograniczenia zakresu (ang. *restrict range*) posiada dwa argumenty; pierwszy z nich to relacja, drugi zbiór. Wymiar zmiennych wyjściowych relacji musi być równy wymiarowi zbioru. Jeżeli $r_1 = \{x_1 \to y_1 \mid f_1(x_1, y_1)\}$ oraz $r_2 = \{x_2 \mid f_2(x_2)\}$

wówczas ograniczeniem dziedziny relacji r_1 na zbiór r_2 jest relacja $\{x \to y \mid f_1(x,y) \wedge f_2(y)\}$.

Przykład

S := {[i,j] : 1<=i<=10 && 1<=j<=10}

R := {[i,j] -> [i,j+1] : 1 <= i <= 100 && 1 <= j <= 100}

Wynikiem operacji R / S jest relacja R1:

R1 := {[i,j] -> [i,j+1] : 1 <= i <= 10 && 1 <= j <= 9}

Aplikacja relacji na zbiorze

W proponowanych algorytmach użyto operacji R(S), czyli aplikacji relacji na zbiorze. Operacja ta wymaga dwóch argumentów: relacji R oraz zbioru S i składa się z dwóch operacji z arytmetyki Presburgera: ograniczonej dziedziny i zakresu. Najpierw wykorzystując argumenty obliczana jest relacja na ograniczonym zbiorze dziedziny S (operacja opisana w punkcie 2.5.2). Następnie wystarczy obliczyć zakres otrzymanej relacji, by uzyskać wynik operacji R(S), (punkt 2.5.3).

Jeżeli $R = \{x_1 \to y_1 \mid f_1(x_1, y_1)\}$ oraz $S = \{x_2 \mid f_2(x_2)\}$, wówczas aplikacją relacji na zbiorze R(S), jest zbiór: $\{y \mid \exists x, \{x \to y \mid f_1(x,y) \wedge f_2(x)\}\}$.

Przykład:

S := {[i,j] : 1<=i<=10 && 1<=j<=10}

R := {[i,j] -> [i,j+1] : 1 <= i <= 100 && 1 <= j <= 100}

Wynikiem operacji R(S) jest zbiór S1:

S1 := {[i,j] : 1<=i<=10 && 2<=j<=11}

2.5.3. Operacje unarne

Operacje unarne wymagają tylko jednego argumentu – zbioru lub relacji.

Operacja dziedziny

Operacja dziedziny (ang. *domain*) można zastosować tylko na relacji. Wynikiem tej operacji dla relacji $r = \{x \to y \mid f(x,y)\}$ jest zbiór $\{x \mid \exists y, f(x,y)\}$.

Przykład:

R := {[i,j] -> [i,j+1] : 1 <= i <= 100 && 1 <= j <= 100}

Wynikiem tej operacji Domain(R) jest zbiór:

S := {[i,j]: 1 <= i <= n && 1 <= j < n}

Operacja zakresu

Operacja zakresu (ang. *range*) można zastosować tylko na relacji. Wynikiem tej operacji dla relacji $r = \{x \to y \mid f(x,y)\}$ jest zbiór $\{y \mid \exists x, f(x,y)\}$.

Przykład:

R := {[i,j] -> [i,j+1] : 1 <= i <= 100 && 1 <= j <= 100}

Wynikiem operacji Range(R) jest zbiór:

S := {[i,j]: 1 <= i <= n && 2 <= j <= n}

Negacja

Argumentem tej operacji może być zarówno relacja i zbiór. Wynikiem operacji dla $r = \{x \to y \mid f(x,y)\}$ jest postać $r = \{x \to y \mid \neg f(x,y)\}$.

Przykład 1:

S := {[i] : 1<=i<=10}

Negacją zbioru S jest zbiór:

¬ S := {[i] : i < 1 union i>10}

Przykład 2:

R1 := {[i,j] -> [i,j+1] : 1 <= i <= n && 1 <= j < n}

Negacją relacji R jest relacja:

¬ R := {[i,j] -> [i',j'] : j <= 0} union {[i,j] -> [i',j'] : n <= j && 1 <= j} union
{[i,j] -> [i',j'] : 1 <= j < n && i <= 0} union {[i,j] -> [i',j'] : 1 <= j < n < i} union
{[i,j] -> [i',j'] : 1 <= j <= n-1, j'-2 && 1 <= i <= n} union
{[i,j] -> [i',j'] : 1, j' <= j < n && 1 <= i <= n} union
{[i,j] -> [i',j+1] : 1 <= i <= n, i'-1 && 1 <= j < n} union
{[i,j] -> [i',j+1] : 1, i'+1 <= i <= n && 1 <= j < n}

Inwersja

Argumentem tej operacji jest tylko relacja. Wynikiem także jest relacja. Dla $r = \{x \to y \mid f(x,y)\}$ inwersją jest $r = \{y \to x \mid f(x,y)\}$.

Przykład:

R := {[i,j] -> [i,j+1] : 1 <= i <= n && 1 <= j < n}

Inwersją relacji R jest relacja R1:

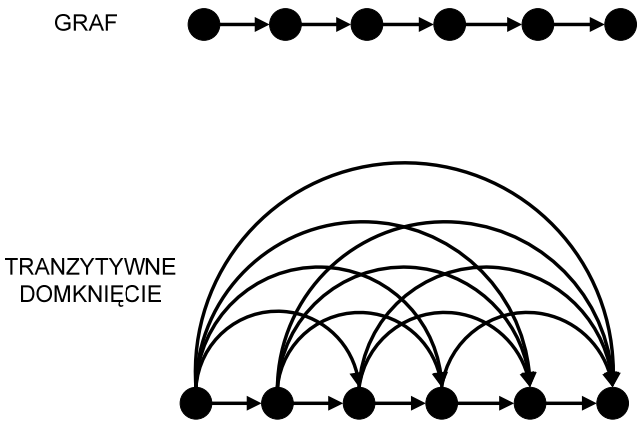
R1 := {[i,j] -> [i,j-1] : 1 <= i <= n && 2 <= j <= n}

Tranzytywne domknięcie

Tranzytywnym domknięciem *(ang. transitive closure)* relacji R nazywamy relację R^* będąca najmniejszym zbiorem takim, że

- $R \subseteq R^*$, $R^* = I \cup R \cup R^2 \ldots$ gdzie I jest relacją tożsamości I:={[J]->[J'] : J=J'}.
- jeżeli (x,y) $\in$ R oraz (y,z) $\in$ R*, to także (x,z) $\in$ R*

Obliczenie tranzytywnego domknięcia jest operacją często spotykana w różnorodnych aplikacjach naukowych [57]. Przykład operacji tranzytywnego domknięcia zilustrowano na rys. 2.5.



Rys. 2.5. Przykład operacji tranzytywnego domknięcia [1]

Istnieją dwie operacje związane z tranzytywnym domknięciem:

- **Operacja tranzytywnego domknięcia (Transitive Closure) -** określana jest przez następujący wzór:

$$x \to z \in F^* \Leftrightarrow x = z \vee \exists y \ s.t. \ x \to y \in F \wedge y \to z \in F^*$$

- **Operacja dodatniego tranzytywnego domknięcia (Positive Transitive Closure) :**

$$x \to z \in F^+ \Leftrightarrow x \to z \in F \vee \exists y \ \ s.t. \ \ x \to y \in F \wedge y \to z \in F^+$$

Zależność między tranzytywnym domknięciem a dodatnim tranzytywnym domknięciem wyrażano jest następująco:

$$R^* = R^+ \cup I$$

Przykład:

R := {[i] -> [i+1] : 1 <= i <= n }

Tranzytywnym domknięciem relacji R jest relacja R*:

R*:= {[i] -> [i'] : 1 <= i < i' <= n}

Dodatnim tranzytywnym domknięciem relacji R jest relacja R+:

R+:= {[i] -> [i'] : 1 <= i <= i' <= n}

W tabeli 2.1 podsumowano opisane operacje arytmetyki Presburgera.

Tab. 2.1. Operacje arytmetyki Presburgera

Operacja	**Oznaczenie**	**Argumenty**	**Wynik**
Operacje binarne			
Alternatywa	$\cup$	Relacje Zbiory	Relacja Zbiór
Koniunkcja	$\cap$	Relacje Zbiory	Relacja Zbiór
Różnica	-	Relacje Zbiory	Relacja Zbiór
Ograniczenie dziedziny	R \ S	Relacja i zbiór	Zbiór
Ograniczenie zakresu	R / S	Relacja i zbiór	Zbiór
Aplikacja relacji na zbiorze	R(S)	Relacja i zbiór	Zbiór
Operacje unarne			
Dziedzina	Domain(R)	Relacja	Zbiór
Zakres	Range(R)	Relacja	Zbiór
Negacja	$\neg$R, $\neg$S	Relacja Zbiór	Relacja Zbiór
Inwersja	Inverse(R)	Relacja	Relacja
Tranzytywne domknięcie	R*	Relacja	Relacja
Tranzytywne domknięcie dodatnie	R+	Relacja	Relacja

Ograniczenia relacji i zbiorów opisywane są poprzez kwantyfikatory *dla każdego* $\forall$ oraz *istnieje, że* $\exists$, a także przy użyciu logicznych połączeń negacji, koniunkcji i alternatywy; $\neg$, $\wedge$, $\vee$. Dzięki nim uzyskuje się dokładny i zwięzły opis zależności.

2.6. Zastosowane narzędzia w oparciu o arytmetykę Presburgera

Na potrzeby niniejszej pracy wybrane zostały narzędzia Petit i Omega Calculator. Petit, narzędzie akademickie [54], pozwala na wyznaczenie relacji zależności zgodnie z analizą zależności zaproponowana przez Pugha oraz Wonnacotta [81]. Pętla programowa poddana analizie zapisana jest w języku Petit. Uzyskane relacje zależności stanowią główne dane wejściowe do opisanych w pracy algorytmów. Ponadto Petit dostarcza informację o rodzaju zależności (prosta, odwrotna, po wyjściu) oraz pomiędzy którymi instrukcjami pętli zachodzi dana zależność.

Omega Calculator [55] umożliwia manipulacje na zbiorach i relacjach krotek. Za pomocą narzędzia można wykonać operacje arytmetyki Presburgera na relacjach zależności uzyskanych za pomocą Petit. Omega Calculator dostarcza aplikację konsolową oraz bibliotekę dla programisty. Za pomocą konsoli można interaktywnie wykonywać operacje na zbiorach i relacjach. W implementacji algorytmów w niniejszej pracy skorzystano z biblioteki Omega Calculator napisanej w języku C. Dokumentacja biblioteki [56] opisuje jak tworzyć relacje oraz zbiory i wykonywać operacje arytmetyki Presburgera w kodzie aplikacji.

Pakiet Omega Calculator udostępnia funkcję generowania pętli *codegen*, która przelicza w **porządku leksykograficznym** afiniczny zbiór iteracji [55]. Porządek leksykograficzny, oznaczany symbolem $\prec$, *n*-krotnego iloczynu kartezjańskiego $A^n=A\times A\ldots\times A$, zdefiniowany jest następująco: niech $a=(a_1,\ldots a_n)$ i $b=(b_1,\ldots,b_n)$ to $a\prec b$, jeżeli $a_1 < b_1$ lub $a_1=b_1$, …., $a_k=b_k$ i $a_{k+1}<b_{k+1}$, gdzie $1\leq k\leq n-1$.
Przykład działania funkcji *codegen* za pomocą konsoli dla zbioru iteracji S:

```
S := { [i,j] : 1 <= i <= n && 1 <= j <= n };
# codegen S;
for(t1 = 1; t1 <= 10; t1++) {
  for(t2 = 1; t2 <= 10; t2++) {
    s1(t1,t2);
  }
}
```

Uzyskany kod tą funkcją posłużył w poniżej zaprezentowanych algorytmach do uzyskania docelowego kodu równoległego. W opracowaniu algorytmów ważne jest zatem znalezienie właściwego zbioru iteracji, przy użyciu arytmetyki Presburgera, służącego do konstruowania docelowej pętli równoległej.

W opisie relacji zależności za pomocą arytmetyki Presburgera może wystąpić kwantyfikator *istnieje, że* (ang. *Exists*). Jest to szczególne zawężenie zbioru, które

występuje na przykład przy ekstrakcji zależności w pętlach modyfikujących zmienne indeksowe o wartość inną niż jeden. Przykład:

for i=1 to N by 2 do
  for j=1 to M do
  a(i,j) = a(i,j-1)
  endfor
endfor
	Relacja zależności uzyskana narzędziem Petit
	{[i,j] -> [i,j+1] : Exists (alpha : 0 = 1+i+2alpha &&
	  1 <= i <= n && 1 <= j < n)}

Powyższe ograniczenie relacji zależności można odczytać, że zmienna *i* mieści się w przedziałach 1 <= i <= n, zmienna *j* w przedziałach 1 <= j < n, i **istnieje taka zmienna całkowita *alpha*,** dla której spełnione jest równanie 0 = 1+i+2alpha. Innymi słowy zmienna *i* jest nieparzysta.

W niektórych przypadkach operacji Omega Calculator można uzyskać ograniczenia typu **UNKNOWN** [55]. Oznaczają one niedokładny opis relacji lub zbioru. Są wynikiem operacji arytmetyki Presburgera, w której nie zdołano uzyskać dokładnego wyniku (z powodów ograniczeń algorytmów zaimplementowanych w narzędziu Omega [56], [81]). Taka relacja lub zbiór może zostać poddana aproksymacji. Wyróżniamy dwa rodzaje aproksymacji niedokładnych relacji:

- Górna granica (ang. *Upper_Bound*) – wyrażenia UNKNOWN są zamieniane na ograniczenia prawdziwe dla każdego przypadku – TRUE
- Dolna granica (ang. *Lower_Bound*) – wyrażenia UNKNOWN są zamieniane na ograniczenia fałszywe dla każdego przypadku - FALSE

Wyniki niedokładny może zostać uzyskany np. w przypadku obliczania tranzytywnego domknięcia skomplikowanej, niejednorodnej relacji.

2.7. Projektowanie równoległych algorytmów

Projektowanie równoległych algorytmów może zostać zrealizowane na wiele sposobów. Często najlepsze rozwiązanie znacznie różni się od sugerowanego przez jego sekwencyjny odpowiednik. Jest to spowodowane faktem, że na postać rozwiązania wpływ mają różne przesłanki, np. efektywność, skalowalność, lokalność, redukcja synchronizacji. W pracy [40] Ian Foster przedstawił metodologię PCAM (ang. *Partitioning-Communication-Agglomeration-Mapping*) tworzenia algorytmów równoległych. Składa się ona z czterech etapów (rys. 2.6):

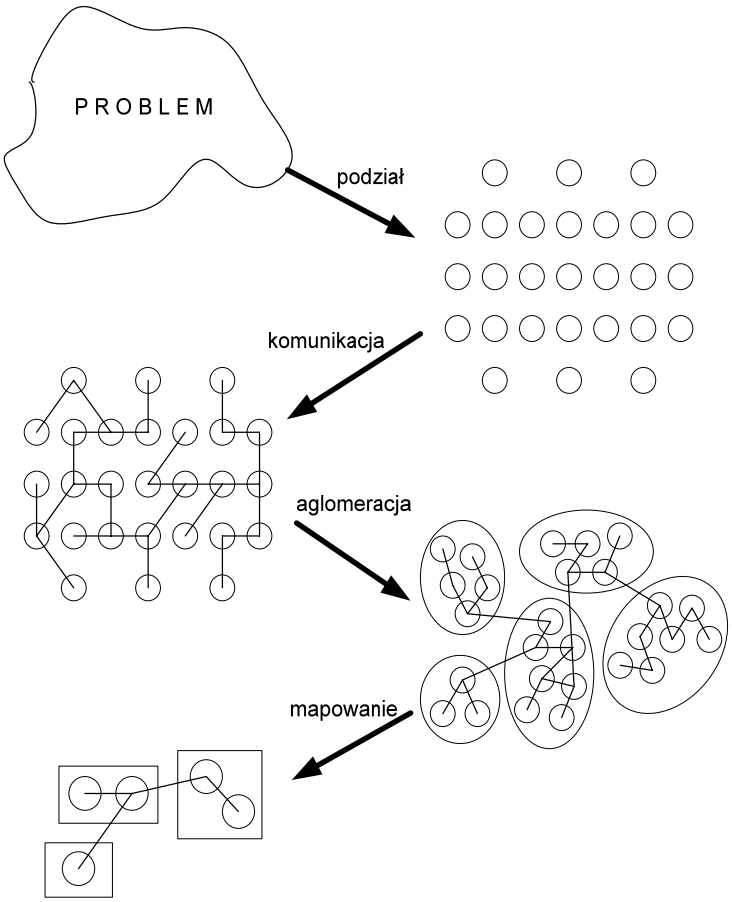
1. Podział (ang. *partitioning*) – obliczenia do wykonania oraz dane na których te obliczenia mają zostać wykonane poddane są podziałowi na mniejsze

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. Na tym etapie określone są algorytmy i 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 wymagane zadania są łączone w większe w celu zredukowania kosztów implementacji lub zwiększenia wydajności.
4. Mapowanie (ang. *mapping*) – zadania mapowane są na procesowy w sposób pozwalający na maksymalizację wykorzystania procesorów oraz minimalizację kosztów komunikacji.

W pierwszych dwóch etapach główny nacisk położony jest na konkurencyjność oraz skalowalność algorytmów. Natomiast w dwóch ostatnich etapach ważniejsza jest kwestia 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), zawierające lub nie zawierające komunikacji między sobą – odpowiedniki dwóch pierwszych kroków. Zdania takie mogą zostać poddane aglomeracji w celu zwiększenia ziarna obliczeń (krok trzeci) i ewentualnego zredukowania punktów synchronizacji. Dowolna technika pozwalająca na mapowanie niezależnych zadań może zostać zastosowana w celu przeprowadzania ostatniego kroku.



Rys. 2.6. Metodologia projektowania algorytmów równoległych [40].

2.8. Podsumowanie

W niniejszym rozdziale przedstawione zostały podstawowe pojęcia związane z przetwarzaniem równoległym. Zaprezentowano podstawowe wskaźniki wykorzystywane w przetwarzaniu równoległym tj. czas obliczeń, przyśpieszenie, efektywność, prawa Amdahla i Gustafsona, skalowalność. Za ich pomocą oceniono jakość proponowanych rozwiązań zrównoleglenia pętli programowej. Przybliżono również model projektowania algorytmów równoległych.

Szczegółowo omówiono zależności występujące w kodzie, ich rodzaje i sposoby reprezentacji z naciskiem na reprezentację zaproponowaną przez Pugha i Wonnacotta [81]. Omówiona arytmetyka Presburgera posłuży do łatwiejszego zrozumienia opisu proponowanych algorytmów zawartego w następnym rozdziale.

3. ALGORYTMY WYSZUKIWANIA FRAGMENTÓW KODU

W niniejszym rozdziale przedstawiono algorytmy umożliwiające automatyczne wyszukiwanie fragmentów kodu dla pętli dowolnie zagnieżdżonych.

Zaproponowano algorytm 3.1 wyszukiwania początków fragmentów kodu w pętlach, dla których relacji zależności mogą tworzyć wspólne początki i końce. Kolejny algorytm 3.2 umożliwia wyznaczenie afinicznego zbioru iteracji fragmentów kodu w porządku leksykograficznym i wygenerowanie pętli równoległej.

W przypadku niemożliwości zastosowania algorytmu 3.2 (gdy niemożliwe jest wyznaczenie zbioru iteracji fragmentów kodu z powodu ograniczeń Omega Calculator) zaproponowano rozwiązanie przebierania iteracji za pomocą pętli *while* z uwzględnieniem topologii zależności (algorytmy 3.3-3.5) i możliwości jej redukcji (algorytmy 3.6-3.8). W przypadku, gdy algorytm 3.1 pozwala tylko na wyszukanie jednego fragmentu kodu, nie można uzyskać wprost równoległości. W algorytmie 3.9 zaproponowano podejście, w którym uzyskany kod równoległy składa się z fragmentów kodu z synchronizacją. Relacje zależności zostają podzielone na dwa zbiory: pierwszy z nich umożliwia wyszukanie wielu fragmentów kodu (np. stosując algorytmy 3.1-2), drugi służy do wyznaczenia synchronizacji pomiędzy nimi.

W celu zrównoleglenia pętli tworzony jest graf zależności RDG (ang. *reduced dependence graph)*. Graf ten podzielony jest następnie na podgrafy SCCs (ang. *strongly connected components)* i umożliwia skuteczniejszą ekstrakcję równoległości. Skorzystano tu z algorytmu wyznaczania niezależnych fragmentów kodu dla grafu zależności o dowolnej strukturze, zaczerpniętego z pracy [85].

Niniejszy rozdział zawiera szczegółowy opis zaproponowanych algorytmów automatycznego wyszukiwania niezależnych fragmentów kodu. Opisane zostały dane

wejściowe jak i wyjściowe algorytmów. Do każdego algorytmu dołączono przykład wraz z omówieniem kolejnych kroków.

3.1. Definicja fragmentu kodu

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

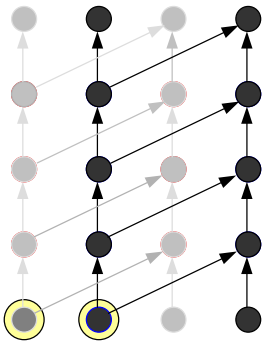
Definicja 1. Niech dany będzie graf zależności D opisany przez relacje zależności. **Fragment kodu** (ang. *slice*) jest luźno spójnie składową (ang. *weakly connected component*) grafu D. Innymi słowy fragment kodu jest maksymalnym podgrafem grafu D, w którym istnieje nieskierowana ścieżka pomiędzy każdą parą wierzchołków.

Fragment kodu jest wolny od synchronizacji (ang. *synchronization-free slice*), jeśli nie istnieją zależności pomiędzy jego operacjami i operacjami należącymi do innych fragmentów kodu. Zaproponowane rozwiązania pozwalają na wyznaczenie fragmentów kodu pozbawionych lub wymagających synchronizacji, które mogą zawierać wspólne początki i końce należące do różnych zależności.

Definicja 2. Początek reprezentatywny fragmentu kodu (ang. *source of slice*) jest to leksykograficznie najmniejszy początek krańcowy zawierający się w tym fragmencie, tj. leksykograficznie minimalna operacja spośród wszystkich operacji należących do fragmentu kodu.

Według powyższych definicji fragment kodu posiada jeden początek reprezentatywny i może posiadać dowolną liczbę początków krańcowych. Początki krańcowe zależności, opisanej za pomocą relacji R, jest to zbiór określony wzorem Domain(R)-Range(R).

Algorytm zaproponowany w rozdziale 3.3 znajduje zbiór początków fragmentów kodu. Na rysunku zilustrowano powyższe definicje. Na rysunku ukazano dwa niezależne fragmenty kodów i zaznaczono okręgiem ich początki. Każdy z dwóch fragmentów kodu oprócz początku reprezentatywnego zawiera także jeden początek krańcowy.



Rys. 3.1. Przykład dwóch fragmentów kodu w przestrzeni iteracji pętli [op. własne]

3.2. Definicja wspólnych iteracji pomiędzy relacjami opisujących zależności pętli

Głównym założeniem pracy jest zwiększanie ekstrakcji równoległości w porównaniu z innymi rozwiązaniami. W pracy [85] zaprezentowano algorytmy oparte o arytmetykę Presburgera umożliwiające zrównoleglenie poprzez podział obszaru iteracji pętli, w której nie występują wspólne iteracje pomiędzy jej relacjami zależności. Uzyskany kod równoległy składał się z łańcuchów iteracji. W poniżej przedstawionych algorytmach założono, że zrównoleglenie będzie dotyczyć także fragmentów kodu z wspólnymi iteracjami, tzn. o innej topologii niż łańcuch. W tym celu dokonano określenia definicji wspólnych iteracji i topologii zależności oraz przybliżono ich rodzaje.

3.2.1. Wyznaczanie wspólnych początków i końców zależności

Zbiór wspólnych iteracji pomiędzy dwoma zależnościami opisanymi za pomocą relacji R1 i R2 można określić następująco:

$CI := (Domain(R1) \cup Range(R1)) \cap (Domain(R2) \cup Range(R2))$

Jak wynika z wzoru jest to część wspólna zbiorów wszystkich zależnych iteracji opisywanych przez relacje zależności R1 i R2. Można zapis ten łatwo rozszerzyć na *n* relacji. Wspólne iteracje mogą też być wspólnymi początkami lub końcami zależności.

Niech relacja R będzię unią relacji dowolnej liczby n relacji:

$R := R1 \text{ union } R2 \text{ union } \ldots Rn$

Zbiór wspólnych początków zależności można określić następującym wzorem:

$CDS := \{[e] : e = R^{-1}(e') = R^{-1}(e'') \;\&\&\; e', e'' \in range(R) \;\&\&\; e' \neq e''\}$

Dla każdej krotki $I \in CDS$ liczność zbioru Range(R(I)) jest większa niż 1.

Zbiór wspólnych końców zależności można określić analogicznym wzorem:

$CDD := \{[e] : e = R(e') = R(e'') \;\&\&\; e', e'' \in domain(R) \;\&\&\; e' \neq e''\}$

Dla każdej krotki $I \in CDS$ liczność zbioru $Range(R^{-1}(I))$ jest większa niż 1.

Warto zauważyć, że powyższe wzory umożliwiają także wyznaczenie wspólnych początków i końców zależności opisanej za pomocą pojedynczej relacji (nie będącej unią wielu relacji).

3.2.2. Topologie zależności

Poprzez topologię zależności autor rozumie topologię grafu tworzonego przez wszystkie zależności pętli opisanych za pomocą relacji.

Posiadając definicję wspólnych początków i końców zależności można określić rodzaje topologii grafu zależności:

Niech relacja R będzie unią relacji dowolnej liczby n relacji opisujących zależności:

R := R1 ∪R2 ∪… Rn,

Topologia zależności opisanych przez relację R jest łańcuchem, jeśli spełniony jest warunek:

$$CDS = \phi \text{ i } CDD = \phi$$

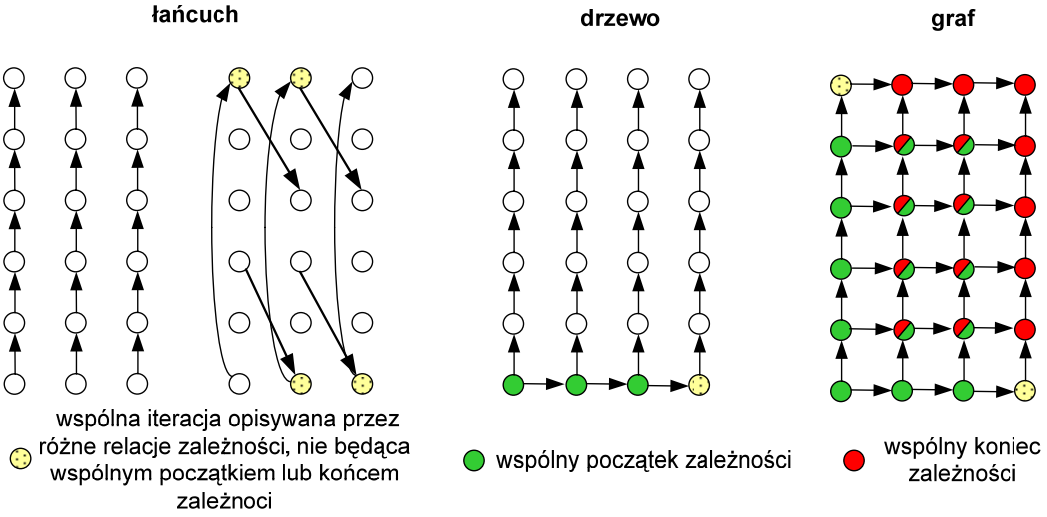
Topologia zależności opisanych przez relację R jest drzewem, jeśli spełniony jest warunek:

$$CDS \neq \phi \text{ i } CDD = \phi$$

Topologia zależności opisanych przez relacje R jest grafem (ani łańcuchem, ani drzewem), jeśli spełniony jest warunek:

$$CDD \neq \phi$$

Innymi słowy: topologia łańcucha nie posiada wspólnych początków i końców zależności. W topologii drzewa pomiędzy zależnościami występują wspólne początki zależności, lecz nie ma wspólnych końców zależności. Występowanie wspólnych końców zależności oznacza topologię grafu. Na rysunku 3.2 zilustrowano na przykładach pojęcia wspólnej iteracji różnych zależności, wspólnego początku zależności, wspólnego końca zależności oraz rodzaje topologii.



Rys. 3.2. Przykłady topologii zależności [op. własne].

Topologia zależności oraz redukcja topologii na prostszą decydują o jakości kodu równoległego w niżej opisanych algorytmach. Analiza topologii zależności jest również ważna w zakresie doboru odpowiedniego algorytmu do zrównoleglenia pętli.

3.3. Algorytm wyszukiwania początków fragmentów kodu

W celu wyznaczenia fragmentów kodu, należy najpierw określić ich początki. Zbiór początków jest niezbędny do wygenerowania kodu równoległego za pomocą dalej opisanych algorytmów.

Jeżeli zbiór relacji zależności R_i, $1 \leq i \leq q$, składa się z krotek wejściowych lub wyjściowych o różnych rozmiarach, należy ustalić ich maksymalny rozmiar *k,* a następnie rozmiar każdej relacji rozszerzyć do *k* w następujący sposób: uzupełnić brakujące wymiary krotki wartościami -1 tj. $e = [e_1, \ldots, e_n]$ rozszerzyć do: $e = [e_1,\ldots,e_n, \underbrace{-1, \ldots, -1}_{n-k}]$.

Przypadek poszerzania wymiaru krotek relacji odnosi się zwykle do pętli nieidealnie zagnieżdżonych i został zaproponowany m.in. w pracy [85]. Normalizacja relacji do tego samego wymiaru jest wymagana dla poniższego algorytmu wyznaczającego zbiór początków fragmentów kodu.

Ideą algorytmu jest najpierw wyznaczenie wszystkich początków fragmentu kodu. Następnie za pomocą operacji tranzytywnego domknięcia sprawdzane jest, czy znalezione początki nie są zależne od pozostałych – innymi słowy, czy nie należą do tego samego fragmentu kodu. Jeśli tak, odrzucane są one ze zbioru początków fragmentów kodu [19].

Algorytm 3.1 Ekstrakcja początków dla fragmentów kodu

Wejście: zbiór $S := \{R_{ij} \mid i,j \in \langle 1,q \rangle\}$ relacji zależności podgrafu SCC, gdzie każda relacja R_{ij} jest unią relacji zależności pomiędzy instrukcjami s_i i s_j oraz q jest liczbą wierzchołków SCC

Wyjście: zbiory Sources(i), $1 \leq i \leq q$ zawierające początki fragmentów kodu (leksykograficznie najmniejsze) będące instancjami instrukcji s_i

1. Dla każdej relacji R_{ij} z zbioru S:
   Rozszerz krotki relacji R_{ij} o dodatkowy wymiar reprezentujący numer i,j instrukcji, np.:
   $R_{i,j} := \{[e] \rightarrow [e']\}$ zamień na $R_{i,j} := \{[e,i] \rightarrow [e',j]\}$
2. Oblicz relację R jako unię wszystkich relacji z zbioru S, $R := \bigcup_{1 \leq i,j \leq q \wedge R_{ij} \in S} R_{ij}$.

3. Dla każdej instrukcji s_i, $1 \leq i \leq q$:

   Znajdź zbiór UDS(i) zawierający początki krańcowe, które są instancjami instrukcji s_i, jako różnicę pomiędzy unią dziedzin wszystkich relacji opisujących zależności z początkiem w instancji instrukcji s_i i unią przeciwdziedzin wszystkich relacji opisujących zależności z końcem w instancji instrukcji s_j:

   $$UDS(i) := \bigcup_{1 \leq k \leq q \wedge R_{ik} \in S} R_{ik} - \bigcup_{1 \leq k \leq q \wedge R_{ki} \in S} R_{ki} .$$

4. Oblicz zbiór $UDS := \bigcup_{1 \leq i \leq q} UDS(i)$.

5. Dla każdej instrukcji s_i, $1 \leq i \leq q$:

   5.1. Utwórz relację R_UCS(i), reprezentującą wszystkie pary początków właściwych zależności, które połączone są w grafie zależności opisywany przez relację R ścieżką nieskierowaną:

   $$R\_UCS(i) := \{[e] \rightarrow [e'] \mid e \in UDS(i), e' \in UDS, e' \succ e, (R^*(e') \cap R^*(e)) \neq \phi\} .$$

   5.2. Oblicz zbiór początków reprezentatywnych fragmentów kodu:

   Sources(i) := UDS(i) – range(R_UCS(i)).

Warto zauważyć, że zakres relacji R_UCS wyznacza **zbiór początków krańcowych w fragmencie kodu, które nie są jego początkiem reprezentatywnym** (leksykograficznie najmniejszą operacją).

Przykład:

Niech dana będzie pętla w języku Petit:

```
for i=1 to n do
    for j=1 to n do
        a(i,j) = a(i,j-1) + a(i-2,j-1)
    endfor
endfor
```

Wejście:

R_{11} := {[i,j] -> [i,j+1] : 1 <= i <= n && 1 <= j < n} union {[[i,j] -> [i+2,j+1] : 1 <= i <= n-2 && 1 <= j < n}

1. Rozszerzenie krotki relacji;

   R_{11} := {[i,j,1] -> [i,j+1,1] : 1 <= i <= n && 1 <= j < n} union {[[i,j,1] -> [i+2,j+1,1] : 1 <= i <= n-2 && 1 <= j < n}

2. R = R_{11}
3. UDS(1) : { Sym=[n] [i,j,v] j = 1 && v = 0 && 1 <= i <= n && 2 <= n }

4. UDS = UDS(1)
5. i ∈ {1}
   5.1. R_UCS(1) : R_UCS{ Sym=[n] [i,j,1] -> [i',j',1] Exists (alpha : 2alpha = i+i' && j' = 1 && j = 1 && 1 <= i <= i'-2 && i' <= n-2 && 4+i' <= i+2n) OR Exists (alpha : 0 = i+i'+2alpha && j' = 1 && j = 1 && 1 <= i <= i'-2 && i' <= n && 2+i' <= i+2n && 3 <= n) }
   5.2. Range(R_UCS) := { Sym=[n] [i,j,v] j = 1 && v = 1 && 3 <= i <= n }
   Sources(1) := {[i,1,1]: 1 <= i <= 2 && 2 <= n}

Koniec przykładu.

Przestrzeń iteracji pętli przedstawia rysunek 3.1, na którym można zauważyć słuszność wyniku w punkcie 5.2. W pętli znaleziono 2 początki fragmentów kodu.

Możliwość skonstruowania relacji R_UCS pozwala na ekstrakcję początków fragmentów kodu również dla pętli, w których relacje zależności posiadają wspólne końce. W pracy autorskiej [19], [76] zaproponowano rozszerzenie dla narzędzia Omega Calculator umożliwiające konstrukcję relacji R_UCS za pomocą arytmetyki Presburgera.

3.4. Generowanie pętli równoległej dla fragmentów kodu opisanych za pomocą ograniczeń afinicznych

W poprzednim punkcie opisano algorytm wyszukiwania początków niezależnych fragmentów kodu. W celu uzyskania pętli równoległej należy wyznaczyć zbiór iteracji należących do fragmentów kodu, dla których określono początki. W niniejszym punkcie zaproponowano algorytm wyznaczenia takiego zbioru i wygenerowania kodu równoległego. Zbiór iteracji jest opisany za pomocą ograniczeń afinicznych [32], porządek wykonania iteracji jest leksykograficzny.

Ideą algorytmu jest wyznaczenie zbioru tranzytywnie zależnych iteracji od początku określonego w postaci sparametryzowanej. Niezbędne jest zatem wykonanie operacji tranzytywnego domknięcia. Algorytm pozwala na uzyskanie niezależnych fragmentów kodu, które mogą być wykonane równolegle.

Algorytm 3.2 Generowanie kodu z zależnościami afinicznymi

Wejście: Zbiory początków Sources(i) i relacje R_UCS(i), $1 \leq i \leq q$, reprezentowane przez afiniczne ograniczenia otrzymane w wyniku działania algorytmu 3.1; relacja R obliczona w kroku 2 algorytmu 3.1

Wyjście: Kod przebierający niezależne fragmenty kodu i ich instrukcje w porządku leksykograficznym podgrafu SCC

1. Wykonaj operacje dla każdego zbioru początków Sources(i):

```
DLA i = 1,q
    genLoops (wejście: Sources(i); wyjście: OuterLoops, L_I);
    DLA każdego I z zbioru L_I
        JEŻELI R_UCS(i) = ∅
            S_Slice := R* (I)
        W PRZECIWNYM WYPADKU
            S_Slice := R* (R_UCS(i)* (I))
        genLoops (wejście: S_Slice; wyjście: InnerLoops, L_J);
        DLA każdego J z zbioru L_J
            genLoopBody (wejście: OuterLoops, InnerLoops, J; wyjście: LoopBody);
```

gdzie:

- *genLoops (wejście: OperSet; wyjście: Loops, VectorList)* jest funkcją generującą zbiór pętli *Loops* przebierających instrukcje zbioru *OperSet* oraz zwracającą listę właściwych, sparametryzowanych wektorów iteracji (znajdujących się w kodzie pętli *Loops)* *VectorList*, składających się ze zmiennych indeksowych pętli,
- *genLoopBody(wejście: OuterLoops, InnerLoops, Iter; wyjście: LoopBody)* to funkcja generująca ciało pętli *LoopBody* dla *InnerLoops* zawierające instrukcje oryginalnej pętli wejściowej wykonanych w iteracji *Iter* i wstawiająca *InnerLoops* jako zagnieżdżone pętle wewnętrzne w odpowiednie miejsce w ciele pętli przebierającej początki *OuterLoops.* W celu wygenerowania *LoopBody* rozpatrywany jest ostatni element krotki reprezentującej zbiór *InnerLoop* (identyfikuje on instrukcję oryginalnej pętli wejściowej i pozwala na właściwy wybór instrukcji wstawianej do ciała pętli wyjściowej).

Poniżej przedstawiono działanie algorytmu na przykładzie z podrozdziału 3.3.

Przykład:

Wejście:

- Sources(1) := {[i,1,1]: 1 <= i <= 2 && 2 <= n}
- R_UCS(1) : R_UCS{ Sym=[n] [i,j,1] -> [i',j',1] Exists (alpha : 2alpha = i+i' && j' = 1 && j = 1 && 1 <= i <= i'-2 && i' <= n-2 && 4+i' <= i+2n) OR Exists (alpha : 0 = i+i'+2alpha && j' = 1 && j = 1 && 1 <= i <= i'-2 && i' <= n && 2+i' <= i+2n && 3 <= n) }
- R := {[i,j,1] -> [i,j+1,1] : 1 <= i <= n && 1 <= j < n} union {[[i,j,1] -> [i+2,j+1,1] : 1 <= i <= n-2 && 1 <= j < n}

1. Wynikiem działania pierwszej funkcji genLoops jest pętla początków:

```
if (n >= 2) {
 for(t1 = 1; t1 <= 2; t1++) {
  s1(t1,1,1);
 }
}
```

oraz zbiór złożony z jednego wektora (gniazda) I = [t1,1,1].

R_UCS(1) ≠ ∅ zatem S_Slice := R* (R_UCS(i)* (I))

S_Slice := { Sym=[n,t1] [i,j,v] Exists (alpha : t1+i+2alpha = 0 && v = 1 && t1+2, 1 <= i <= n && 2 <= j <= n && 5 <= n && 1 <= t1) OR Exists (alpha : t1+i+2alpha = 0 && j = 1 && v = 1 && 1 <= t1 <= i-2 && i <= n && 5 <= n) OR v = 1 && t1 = i && 1 <= i <= n && 2 <= j <= n && 5 <= n OR j = 1 && v = 1 && t1 = i }

Warto zauważyć, że zbiór S_SLICE posiada dodatkową zmienną sparametryzowaną *t1* dla wektora I = [t1, 1, 1]. Zbiór ten wyznacza iteracje należące do fragmentu kodu o początku [t1,1]. Ostatnia zmienna w wektorze oznacza numer jedynej instrukcji w ciele pętli.

Wynikiem działania kolejnych funkcji jest poniższa pętla równoległa. Zbiór L_J składa się z dwóch wektorów J = {[t1,t2,1]; [t1',t2, 1]}. Za pomocą funkcji *genLoopBody* wstawiono zamiast wektorów J oryginalne instrukcje pętli wejściowej.

```
if (n >= 2) {
 par for(t1 = 1; t1 <= 2; t1++) {
  if (t1 >= 1 && n >= t1) {
   for(t2 = 1; t2 <= n; t2++) {
    a[t1][t2] = a[t1][t2-1] + a[t1-2][t2-1];
   }
  }
  if (t1 >= 1) {
   for(t1' = t1+2; t1' <= n; t1' += 2) {
    for(t2 = 1; t2 <= n; t2++) {
     a[t1'][t2] = a[t1'][t2-1] + a[t1'-2][t2-1];
    }
   }
  }
 }
}
```

Pętla zewnętrzna (OuterLoop)

Pętla wewnętrzna (InnerLoop) zamiast gniazda [t1, 1, 1] z pętli zewnetrznej (OuterLoop)

Koniec przykładu.

Algorytm 3.2 pozwala uzyskać kod równoległy dla dowolnej topologii zależności. Jednak nie pozwala on na zrównoleglenie pętli w następujących sytuacjach:

- w przypadku, gdy zastosowane narzędzie nie umożliwia obliczenia dokładnego tranzytywnego domknięcia relacji,
- w przypadku, gdy za pomocą algorytmu 3.1 zostanie uzyskany tylko jeden początek. Wówczas wynikiem działania algorytmu 3.2 jest kod sekwencyjny.

W dalszej części rozdziału przedstawiono kolejne algorytmy, których celem opracowania było umożliwienie ekstrakcji równoległości pomimo powyższych ograniczeń.

3.5. Ekstrakcja równoległości dla pętli niejednorodnych

Ograniczeniem algorytmu 3.2, a także jakichkolwiek transformacji afinicznych jest niemożliwość uzyskania maksymalnej równoległości dla pętli niejednorodnych w ogólnym przypadku [16]. Niech dana będzie pętla:

```
for i=1 to n do
  a(i) = a(2*i)
endfor
```

Rys. 3.3. Zależności pętli dla n=12[16].

Pętla posiada jedną zależność opisywaną przez relację (rys. 3.3).

R1 := {[i] -> [2i] : 1 <= i && 2i <= n}

Korzystając z algorytmu 3.1 wyznaczono zbiór początków:

Sources :={[i,1]: Exists (alpha : 2alpha = 1+i && 1 <= i && 2i <= n)}

Próbując zastosować algorytm 3.2 przy obliczaniu tranzytywnego domknięcia R*, otrzymany wynik jest niedokładny:

R* = {[i] → [i] } union {[i] → [i'] : Exists (alpha : 0 = i'+2alpha & 4i ≤ i' ≤ n & 1 ≤ i & 2i' ≤ n+8i && UNKNOWN)} union {[i] → [2i] : 1 ≤ i && 2i ≤ n} union {[i] → [4i] : 1 ≤ i & 4i ≤ n}

Wyrażenie UNKNOWN oznacza nieznane ograniczenie [56]. Stosując narzędzie Omega Calculator nie można uzyskać dokładnego wyniku, lecz nadal możliwe jest zastosowanie aproksymacji górnej i dolnej [56].

UB = UpperBourdR* = {[i] → [i] } ∪ {[i] → [i'] : Exists (alpha : 0 = i'+2alpha & 4i ≤ i'≤ n & 1 ≤ i & 2i' ≤ n+8i)} ∪ {[i] → [2i] : 1≤ i & 2i≤n},

LB = LowerBoundR* = {[i] → [i] } ∪ {[i] → [2i]: 1≤i & 2i≤n} ∪ {[i] → [4i]: 1≤ i & 4i≤n}.

Po zastosowaniu aproksymacji górnej (wyrażenia UNKNOWN zamieniane są na ograniczenia prawdziwe – TRUE) otrzymany został wynik:

UB({1})={1, 2, 4 , *6,* 8, *10*}, UB({3})={3, 6, 12}, UB({5})={5,10}.

Warto zauważyć, że iteracje {6} i {10} występują jednocześnie w różnych zbiorach. Po zastosowaniu aproksymacji dolnej (wyrażenia UNKNOWN zamieniane są na ograniczenia fałszywe – FALSE) otrzymany został wynik:

LB({1})={1, 2, 4}, LB({3})={3, 6, 12}, LB({5})={5,10}.

Porównując z rysunkiem 3.3 zauważono, że pierwszy fragment nie zawiera iteracji {8}. Stwierdzono, że ani górna, ani dolna aproksymacja nie pozwala na uzyskanie dokładnego wyniku.

W niniejszym punkcie zaproponowano algorytmy, które pozwalają na dokładną ekstrakcję równoległości dla pętli niejednorodnych. Główną ideą jest zastosowanie pętli *while* (dopóki) oraz przebieranie zbioru iteracji w czasie wykonania (ang, *run time*). Istotnym czynnikiem dla takiego zrównoleglenia jest topologia zależności, która została objaśniona w punkcie 3.2.2. Zaproponowano trzy algorytmy dla każdej topologii. Warto zauważyć, że uzyskany kod równoległy jest najwydajniejszy i najmniej skomplikowany dla topologii łańcucha. Dlatego dalej zawarto w rozdziale rozwiązania pozwalające na transformację topologii.

3.5.1. Wyszukiwanie niezależnych łańcuchów

Łańcuch jest najprostszą topologią. Brak wspólnych początków i końców ułatwia skonstruowanie wydajnej pętli *while*, uwzględniając wszystkie zależności pętli wejściowej.

Poniżej zaprezentowany algorytm pozwala na ekstrakcję równoległości dla pętli o wielu relacjach zależności. Pomiędzy zależnościami mogą istnieć wspólne iteracje, które nie są ich wspólnymi początkami lub końcami.

Algorytm 3.3 Wyszukiwanie niezależnych łańcuchów

Wejście: zbiór relacji zależności R_i, $1 \le i \le q$, opisanej w postaci $R_i := [A_iI+B_i] \rightarrow [C_iJ+D_i]$: *ograniczenia*}, gdzie I, J to wektory reprezentujące zmienne odpowiednio wejściowych i wyjściowych krotek, A_i, C_i są macierzami, B_i, D_i wektorami.

Wyjście: kod przebierający niezależne łańcuchy i ich iteracje w porządku leksykograficznym.

1. Oblicz $R := \bigcup_{1 \le i \le q} R_i$
2. Sprawdź, czy relacja R opisuje wspólne początki i końce zależności:
   2.1. Oblicz zbiór: CDS: ={[e] : e = R^{-1}(e') = R^{-1}(e") & e',e"∈ range(R) & e'≠e"}.

Oblicz zbiór: CDD: ={[e] : e = R(e') = R(e'') & e',e''∈ domain(R) & e'≠e''}.

2.2. **Jeżeli** CDS = ∅ i CDD = ∅

Kontynuuj obliczenia od kroku 3.

W przeciwnym wypadku:

Topologia zależności nie jest łańcuchem, przerwij wykonanie algorytmu.

3. Oblicz zbiór początków Sources := domain(R) – range(R) lub stosując algorytm 3.1
4. Wygeneruj kod przebierający niezależne łańcuchy i ich iteracje w porządku leksykograficznym wywołując poniższą funkcję.

4.1. GenOuterLoops(wejście: Sources; wyjście: OuterLoops, L_Iter);

gdzie: funkcja *genOuterLoops (wejście: OperSet; wyjście: Loops, VectorList)* generuje zbiór zewnętrznych pętli, które przebierają iteracje z zbioru *OperSet*, zwraca sparametryzowaną listę wektorów iteracji *VectorList,* pojedynczy wektor *I* z listy *VectorList* opisuje jedno gniazdo tj. wywołanie pętli wewnętrznej, liczba gniazd jest równa długości listy *VectorList*. Warto zauważyć, że w ogólnym przypadku zbiór *Sources* jest unią sparametryzowanego wielościanu, którego ograniczenia są afiniczne, ponieważ jest wynikiem operacji różnicy afinicznych zbiorów. Dlatego w celu implementacji funkcji *genOuterLoops*, można zastosować znane techniki [3], [11], [29], [82], [89]. Wszystkie zewnętrzne pęte *OuterLoops* mogą zostać wykonane równolegle, ponieważ przebierają niezależne początki fragmentów kodu wolnych od synchronizacji.

4.2. Każde zewnętrzne gniazdo wygenerowanych pętli w punkcie 4.1 jest skojarzone z wektorem *Iter* z listy *L_Iter*. W jego miejsce wstaw kod pętli do-while w następującej postaci:

```
Ind = Iter;
do {   s(Ind);   //WYKONAJ DOPÓKI
          // s(Ind) oznacza wykonanie oryginalnej instrukcji z pętli wejściowej w
          // iteracji Ind
   JEŻELI (Ind ∈ domain R1)
      Ind=A1^-1(C1*Ind+D1-B1); // wyznaczenie
                               // następnej iteracji w fragmencie kodu

   W PRZECIWNYM WYPADKU JEŻELI (Ind ∈ domain R2)
      Ind=A2^-1(C2*Ind+D2-B2);
   ...
   W PRZECIWNYM WYPADKU JEŻELI (Ind ∈ domain Rq)
      Ind=Aq^-1(Cq*Ind+Dq-Bq);
```

W PRZECIWNYM WYPADKU PRZERWIJ */* przerwij wykonanie pętli do-while, ponieważ nie istnieją już instrukcje należące do łańcucha */*

} while (true); *// warunek zawsze prawdziwy*

Rozważmy przykład podany w punkcie 3.5

Przykład:

1. R := R1
2. CDS = ∅ i CDD = ∅, zależności pętli tworzą topologię łańcucha.
3. Sources :={[i,1]: Exists (alpha : 2alpha = 1+i && 1 <= i && 2i <= n)}
4. Wynik działania funkcji OuterLopps: wektor I = [t1,1]

```
for(t1 = 1; t1 <= intDiv(n,2); t1 += 2) {
  s1(t1,1);  // gniazdo
}
```

Ostateczny wygenerowany kod:

```
for(t1 = 1; t1 <= intDiv(n,2); t1 += 2) {
   do
   {
      a[t1] = a[2*t1];
      if(1 <=t1i && 2*t1 <= n)
      {
        t1 *= 2;
        continue;
      }
      break;
   }
   while(true);
}
```

Koniec przykładu.

W celu poprawności pętli należy dodatkowo wykonać iteracje niezależne. Ich zbiór można obliczyć następująco: IND = IS – (Domain(R) ∪ Range(R)), gdzie IS to przestrzeń iteracji pętli. Do wygenerowania pętli wykorzystano funkcję *codegen* z pakietu Omega Calculator. Pętlę niezależnych iteracji można wprost zrównoleglić.

```
par for(t1 = max(2*intDiv(intDiv(n+2,2)+2,2)-1,3); t1 <= n; t1 += 2) {
  a[t1] = a[2*t1];
}
```

3.5.2. Wyszukiwanie niezależnych drzew

Drzewa to topologia zależności, w której występują wspólne początki pomiędzy zależnościami. Oznacza to, że przebierając iteracje w czasie wykonania, zamiast jednej, następnej iteracji, wyznaczany jest ich cały zbiór. W pętli *while* należy zadeklarować dodatkowy blok pamięci dla takiego zbioru. Następnie wykonując iteracje należy przejrzeć cały ten zbiór, wykonać jego iteracje, i obliczyć kolejny zbiór następnych.

Algorytm 3.4 Wyszukiwanie niezależnych drzew

Wejście: zbiór relacji zależności S := $\{R_1, R_2, \ldots, R_n\}$, n ≥ 1, należących do podgrafu SCC

Wyjście: kod przebierający niezależne drzewa z zachowaniem wszystkich zależności w pętli

1. Oblicz $R := \bigcup_{1 \le i \le q} R_i$
2. Sprawdź, czy relacja R nie opisuje wspólnych końców zależności:
   2.1. Oblicz zbiór: CDD: ={[e] : e = R(e') = R(e") & e',e"∈ domain(R) & e'≠e"}.
   2.2. **Jeżeli** CDD = ∅
   Kontynuuj obliczenia od kroku 3.
   W przeciwnym wypadku:
   Topologia zależności jest grafem ogólnym (nie drzewem), przerwij wykonanie algorytmu.
3. Oblicz zbiór początków Sources := domain(R) – range(R) lub stosując algorytm 3.1
4. Wygeneruj kod przebierający niezależne drzewa i ich iteracje z zachowaniem wszystkich zależności pętli wejściowej w następujący sposób:

```
genOuterLoops (wejście: Sources; wyjście: OuterLoops, L_I);
  DLA KAŻDEGO I z listy L_I WYKONAJ
    genWhile (wejście: OuterLoops, I; wyjście: WhileLoop);
```

gdzie:

- funkcja *genOuterLoops* jest funkcją opisaną w krok 4.1 algorytmu 3.3
- funkcja *genWhile (wejście: OuterLoops, I; wyjście: WhileLoop)* składa się z 2 kroków:
  1. wygeneruj kod *WhileLoop* o następującej postaci:

```
S':=I;
while (S'≠ ∅); {   // DOPÓKI
  S_tmp:= ∅;
  DLA KAŻDEGO  I∈ S' WYKONAJ
```

```
    stk(I'); /* instrukcja k pętli (k jest zdfiniowana przez dodatkową zmienną
zdefiniowaną w wektorze I opisujący wykonanie iteracji I' (I' reprezentuje n-1
zmiennych w krotce I) */
    DLA KAŻDEGO i, i=1,2,..,n WYKONAJ
      JEŻELI R_i(I')∈ range(R_i)
        J= R_i(I); /* oblicz zbiór operacji do wykonania w następnej iteracji */
        Dodaj J do zbioru S_tmp;
      // koniec JEŻELI
    // koniec DLA KAŻDEGO
  // koniec DLA KAŻDEGO
  S':=S_tmp;  }
```

2. pętle *WhileLoop* wstaw w opdpowiednie gniazda pętli *OuterLoops*

Porządek wykonania iteracji jest zgodny z zależnościami pętli. W każdym kroku pętli *while* obliczany jest nowy zbiór iteracji S' na podstawie relacji zależności. Porządek ten nie jest jednak leksykograficzny, lecz ustalony w tzw. **wolnym planowaniu** (ang. *free-schedule*). Jest to technika, która polega na tym, że instrukcje pętli są wykonywane natychmiast jak tylko są gotowe ich dane wejściowe [7].

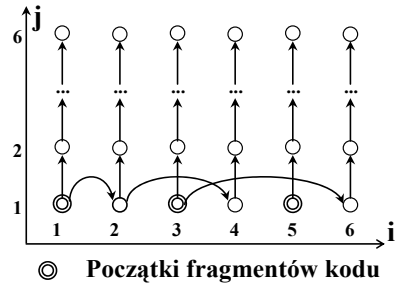
Przykład:

```
for i=1 to n do
 for j=1 to n do
   a(i,j) = (j==1) ? a(2*i,j) : a(i,j-1)
 endfor
endfor
```

Zależności pętli (rys. 3.4):

$R^{1}_{s1,s1}$:={[i,1] → [2i,1]: 1≤i && 2i≤ n},

$R^{2}_{s1,s1}$:={[i,j] → [i,j+1] : 1 ≤ i ≤ n && 1 ≤ j < n }.



Rys. 3.4. Przestrzeń iteracji pętli [17].

1. R := $R^{1}_{s1,s1} \cup R^{2}_{s1,s1}$ ={[i,1]→[2i,1]: 1≤i & 2i≤ n}∪{[i,j]→[i,j+1]:1≤i≤n &1≤j<n};
2. Zbiór CDD jest pusty, kontynuuj od kroku 3.
3. Sources := {[i,1]: Exists (alpha : 2alpha = 1+i && 1 <= i <= n, 2n-3)}.

4. Wygenerowany kod (przedstawiony za pomocą pseudokodu):

```
par for(t=1;t<=min(n,2*n-3);t+=2) /* pętlę zewnętrzne stworzone w Omega Calculator */
{  I=[t,1];
   Dodaj I do S';       /* I jest początkiem drzewa */
   while(S'!= Ø) {
   S_tmp= Ø;
    foreach(vector I=[i,j] ∈ S') {     /* Dla każdego wektora I=[i,j] ∈ S'  */
      s1(I); /* wykonanie instrukcji s1 dla iteracji I */
      if(j==1 && 1<=i && 2*i<=n){/* jeżeli R1(I)∈domain R1*/
            ip = 2*i;  jp = 1;   /* J= [ip,jp]=R1(I) */
          dodaj J=[ip,jp] do S_tmp; }
      if(1<=i && i<=n && 1<=j&&j<n){ /*jeżeli R2(I) ∈domain R2 */
      ip=i;  jp = 1 + j; /* J=[ip,jp]= R2(I) */
          dodaj J=[ip,jp] do S_tmp; }
    }
    S' = S_tmp;
  }
}
```

Koniec przykładu.

Zbiór przetwarzanych iteracji w wygenerowanym kodzie oznaczony jest przez S'. W zbiorze S_tmp gromadzone są iteracje z następnego cyklu. Na koniec pod zbiór S' podstawiany jest S_tmp i rozpoczyna się kolejna iteracja pętli *while*.

3.5.3. Wyszukiwanie niezależnych grafów

Pod pojęciem grafu ogólnego rozumiana jest topologia nie będąca ani grafem, ani drzewem. Zawiera wspólne początki i końce zależności. W pętli *while* samo przechowywanie zbioru iteracji jest niewystarczające dla zachowania wszystkich zależności. Obecność wspólnych końców tworzy następujący warunek: żadna iteracja nie może zostać wykonana, nim wszystkie jej poprzedzające iteracje, od których jest tranzytywnie zależna, nie zostaną wykonane. W tym celu obliczono dodatkową tablicę A zawierającą liczniki iteracji poprzedzających. Wykonanie każdego poprzednika dekrementuje licznik iteracji. Gdy jest on równy 0, iteracja jest dodawana do zbioru iteracji do wykonania.

Algorytm 3.5 Wyszukiwanie niezależnych grafów

Wejście: zbiór relacji zależności S := {R_1, R_2, …, R_n}, n ≥ 1, należących do podgrafu SCC

Wyjście: kod przebierający niezależne grafy z zachowaniem wszystkich zależności w pętli

1. Oblicz $R := \bigcup_{1 \le i \le q} R_i$
2. Oblicz zbiór początków Sources := domain(R) – range(R) lub stosując algorytm 3.1
3. Stwórz tablicę A zawierającą liczbą poprzedników:
   3.1. Zbuduj relację:
   R':={[e]→[e']: R(e)=e' && e∈ domain(R) & e' ∈ range(R) };
   3.2. Oblicz zbiór C := range(R');
   3.3. Wykonaj poniższy kod:

```
DLA KAŻDEGO e ∈ C
   P :=  R^-1(e);
   A(e) = N, gdzie N to liczba elementów w zbiorze P
```

4. Wygeneruj kod przebierający niezależne grafy i ich iteracje z zachowaniem wszystkich zależności pętli wejściowej w następujący sposób:

```
genOuterLoops (wejście: Sources; wyjście: OuterLoops, L_I);
  DLA KAŻDEGO I z listy L_I WYKONAJ
     genWhile (wejście: OuterLoops, I; wyjście: WhileLoop);
```

gdzie:

- funkcja *genOuterLoops* jest funkcją opisaną w krok 4.1 algorytmu 3.3
- funkcja *genWhile (wejście: OuterLoops, I; wyjście: WhileLoop)* składa się z 2 kroków:
  1. wygeneruj kod *WhileLoop* o następującej postaci:

```
S':=I;
while (S'≠ ∅); {   // DOPÓKI
  S_tmp:= ∅;
  DLA KAŻDEGO  I∈ S'  WYKONAJ
    JEŻELI (A(I) = 0)
       stk(I'); /* instrukcja k pętli (k jest zdfiniowana przez dodaktową
zmienną zdefiniowaną w wektorze I opisujący wykonanie iteracji I' (I'
reprezentuje n-1 zmiennych w krotce I) */
    DLA KAŻDEGO i, i=1,2,..,n WYKONAJ
      JEŻELI R_i(I')∈ range(R_i)
         J= R_i(I);  /* oblicz zbiór operacji do wykonania w następnej iteracji */
         Dodaj J do zbioru S_tmp;
         A(J)--;
      // koniec JEŻELI
    // koniec DLA KAŻDEGO
  // koniec DLA KAŻDEGO
S':=S_tmp;
```

}

2. pętle *WhileLoop* wstaw w odpowiednie gniazda pętli *OuterLoops*

Przykład:

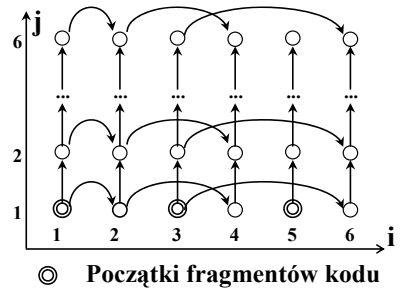
Niech dana będzie pętla:

```
for i=1 to n do
 for j=1 to n do
   a(i,j) = a(2*i,j) + a(i,j-1)
 endfor
endfor
```

Pętla zawiera dwie relacje zależności (rys. 3.5):

$R^{1}_{s1,s1} := \{[i,j] \rightarrow [2i,j]: 1 \leq i \;\&\&\; 2i \leq n \;\&\&\; 1 \leq j < n\}$,

$R^{2}_{s1,s1} := \{[i,j] \rightarrow [i,j+1] : 1 \leq i \leq n \;\&\&\; 1 \leq j < n\}$.



Rys. 3.5. Przestrzeń iteracji pętli [op. własne].

1. $R := R^{1}_{s1,s1} \cup R^{2}_{s1,s1} = \{[i,1] \rightarrow [2i,1]: 1 \leq i \;\&\&\; 2i \leq n \;\&\&\; 1 \leq j < n\} \cup \{[i,j] \rightarrow [i,j+1]: 1 \leq i \leq n \;\&\&\; 1 \leq j < n\}$;
2. Sources := {[i,1]: Exists (alpha : 2alpha = 1+i && 1 <= i <= n, 2n-3)}.

Po obliczeniu tablicy A, wygenerowano następujący kod (zapis w pseudokodzie):

```
par for(t=1;t<=min(n,2*n-3);t+=2) /* pętlę zewnętrzne stworzone w Omega Calculator */
{  I=[t,1];
   Dodaj I do S';       /* I jest początkiem drzewa */
   while(S'!= Ø) {
   S_tmp= Ø;
    foreach(vector I=[i,j] ∈ S') {    /* Dla każdego wektora I=[i,j] ∈ S'  */
      s1(I); /* wykonanie instrukcji s1 dla iteracji I */
      if(1<=j<=n && 1<=i && 2*i<=n){  /* jeżeli R1(I)∈domain R1*/
            ip = 2*i;  jp = j;   /* J= [ip,jp]=R1(I) */
            if(A(J) nie zawiera liczby poprzedników) {   /*oblicz wartość A(I)
                  A(J) = 0;
                  if(1 <= ip && ip <= n && 2 <= jp && jp <= n)
                        A(J)++;
            }
```

```
            else
                A(J)--;
            if(A(j)=0) /*wszystkie iteracje poprzedzające zostały wykonane */
                dodaj J=[ip,jp] do S_tmp;
          }
        if(1<=i && i<=n && 1<=j&&j<n){ /*jeżeli R2(I) ∈domain R2 */
        ip=i;  jp = 1 + j;  /* J=[ip,jp]= R2(I) */
            if(A(J) nie zawiera liczby poprzedników) {   /*oblicz wartość A(I)
                A(J) = 0;
              if( ((ip/2) % = 0) && 1 <= jp && jp <= n && 2 <= ip && ip <= n)
                A(J)++;
            }
            else
                A(J)--;
            if(A(j)=0) /*wszystkie iteracje poprzedzające zostały wykonane */
                dodaj J=[ip,jp] do S_tmp;
       }
       S' = S_tmp;
     }
   }
```

Koniec przykładu.

Zrównoleglenie pętli o topologii grafu wiąże się z największym narzutem na pamięć i obliczenia. Jeżeli czas wykonania instrukcji numerycznych w pętli będzie mniejszy od narzutów czasowych obsługi kodu równoległego, wówczas nie zostanie osiągnięte przyspieszenie obliczeń. Poniżej zaprezentowano trzy algorytmy umożliwiające transformację topologii i uzyskanie bardziej wydajnego kodu równoległego.

3.5.4. Usuwanie nadmiarowych zależności

Redukcja relacji opisujących zależności ma dwie zalety: docelowy kod równoległy jest mniej skomplikowany oraz możliwe jest uzyskanie prostszej topologii. Poniżej przedstawiono algorytm redukcji nadmiarowych zależności. Ideą algorytmu jest sprawdzenie czy ścieżki danej relacji zależności nie są opisywane przez pozostałe. Jeśli taka sytuacja zachodzi, relacja zostaje odrzucona.

Algorytm 3.6 Usuwanie nadmiarowych zależności

Wejście: zbiór q relacji zależności R_i, $1 \leq i \leq q$

Wyjście: zredukowany zbiór relacji zależności

1. Dla każdej relacji R_k, $1 \leq k \leq q$

   1.1. Oblicz $R := \bigcup_{1 \leq i \leq q, i \neq k} R_i$

   1.2. Jeżeli dla każdej krotki e ∈ domain R_k następujący warunek jest prawdziwy:

   $R_k(e) - R+(e) = \phi$

   usuń relację R_k z zbioru relacji.

Dowód: Niech relacja Rk opisuje zależność I -> J. Oznacza to, że instancja instrukcji I musi zostać wykonana przed instancją instrukcji J. Jeżeli dla każdej pary I, J pozostałe relacje zależności opisują ścieżkę I -> ... -> .... J, wówczas tranzytywne domknięcie ich unii R+ zawiera także zależność I->J, natomiast relacja będąca różnicą Rk – R+ jest relacją pustą. Wszystkie zależności opisane przez Rk określone są także za pomocą pozostałych relacji. Możliwe jest zatem wyeliminowanie Rk z zachowaniem wszystkich zależności tranzytywnych.

Przykład:

```
for i=1 to n do
 for j=1 to n do
   a(i,j) = a(i,j-1) +a(i,j-2)
 endfor
endfor
```

Pętla posiada dwie relacje zależności:

```
R1 := {[i,j,1] -> [i,j+1,1] : 1 <= i <= n && 1 <= j < n}
R2 := {[i,j,1] -> [i,j+2,1] : 1 <= i <= n && 1 <= j <= n-2}
```

1. Dla każdej relacji Rk, $1 \leq k \leq 2$

   1.1. R := Rk

   1.2. Dla relacji R1 $R1(e) - R+(e) \neq \phi$

   Relacja R2 jest odrzucona, ponieważ $R2(e) - R+(e) = \phi$

Relacja R2 jest opisywana przez relację R1 i dlatego wyjściowym zbiorem relacji jest już tylko {R1}. Dodatkowo warto zauważyć, że topologia zależności pętli została zredukowana z grafu do łańcucha.

Koniec przykładu.

3.5.5. Konwersja topologii grafu do drzewa lub łańcucha

W niniejszym punkcie opisany jest drugi algorytm transformacji topologii zależności. Jest on dedykowany tylko dla topologii ogólnego grafu. Jego ideą jest próba usunięcia wspólnych końców zależności z zachowaniem semantyki pętli źródłowej

(semantyka pętli wygenerowanej przez algorytm i semantyka pętli źródłowej są identyczne).

Algorytm 3.7 Transformacja topologii grafu do drzewa lub łańcucha

Wejście: zbiór S relacji zależności R_i, $1 \leq i \leq q$ otrzymany z algorytmu 3.6, zbiór początków fragmentu kodu Sour

Wyjście: zbiór zależności S reprezentujący topologię po transformacji, jeśli to możliwe

1. Oblicz $R := \bigcup_{1 \leq i \leq q} R_i$
2. Oblicz zbiór wspólnych końców:

   CDD:= {[e] : e = R(e') = R(e'') && e',e'' ∈ domain(R) && e' ≠ e''}

   Jeżeli CDD ≠ ∅ kontynuuj, w przeciwnym wypadku przerwij algorytm, ponieważ topologia zależności nie jest grafem.

3. Dla każdej relacji R_i, $1 \leq i \leq q$ z zbioru S

   3.1. Stwórz relację R_i' := R_i i poszerz ją o dodatkowe ograniczenia:

   && e ∈ range R_i && e ∉ CDD,

   3.2. Jeżeli $R_i - R_i' = \emptyset$ i $R_i' - R_i = \emptyset$,

   kontynuuj od kroku 3,

   w przeciwnym wypadku

   3.2.1. Stwórz zbiór relacji S', w którym R_i jest zastąpiony przez R_i'

   3.2.2. Oblicz R' jako unię wszystkich relacji z zbioru S'

   3.2.3. Jeżeli R+(Sour) -R'+(Sour)= ∅ i R'+(Sour)- R+(Sour)= ∅,

   zastąp R_i relacją R_i' w zbiorze S

Dowód: Ponieważ w pętli *while* iteracje wykonywane są w kolejności harmonogramowania swobodnego (ang. *free-scheduling*), istotny jest czas wykonania iteracji *e* z zachowaniem wszystkich zależności pętli. Określany jest on na podstawie odległości w ścieżce zależności iteracji *e* od iteracji opisujących początki fragmentów kodu. Zastąpienie relacji Ri przez relację Ri' oznacza odrzucenie ścieżek opisujących zależności, w których zawarte są wspólne końce zależności. Pomimo odrzucenia tych ścieżek odległość *e* od Sour pozostaje taka sama jak i w grafie ogólnym (dzięki warunkowi przedstawionemu w punkcie 3.2.3, który mówi, że relacje muszą prowadzić nadal do tego samego zbioru zakresu iteracji od początku fragmentu kodu). Harmonogramowanie swobodne iteracji zapewnia poprawną semantykę pętli wejściowej dla nowego zbioru relacji, natomiast topologia grafu zależności ulega uproszczeniu.

Przykład:

```
for i=1 to n do
 for j=1 to n do
   a(i,j) = a(i,j-1) +a(i-1,j)
 endfor
endfor
```

Relacje zależności:

```
R1 := {[i,j,1] -> [i,j+1,1] : 1 <= i <= n && 1 <= j < n}
R2 := {[i,j,1] -> [i+1,j,1] : 1 <= i < n && 1 <= j <= n}
```

Początki:

```
Sources [0] := { Sym=[n] [i,j,1] j = 1 && i = 1 && 2 <= n }
```

1. R := {[i,j,1] -> [i,j+1,1] : 1 <= i <= n && 1 <= j < n} union := {[i,j,1] -> [i+1,j,1] : 1 <= i < n && 1 <= j <= n}
2. CDD := {[i,j,1] : 2 <= i <= n && 2 <= j <= n} ≠ ∅
3. Dla każdej relacji R_i, $1 \leq i \leq 2$

   3.1 Dla R1: R1’ = [i,j,1] -> [i',j',1] i = 1 && j' = 1+j && i' = 1 && 1 <= j < n && 1 <= n }

   3.2 Warunek: R1 – R1' =∅ i R1'-R1 =∅, nie jest prawdziwy

   3.2.1 S’ = {R1’, R2};

   3.2.2. R’ = R1’ union R2

   3.2.3 Warunek R+(Sour) -R'+(Sour)= ∅ i R'+(Sour)- R+(Sour)= ∅ jest prawidzwy, zatem relacja R1 jest zastąpiona przez R1’.

   3.1 Dla R2: R2’ { Sym=[n] [i,j,1] -> [i',j',1] i' = 1+i && j' = j && 1 <= j <= n && 1 <= i < n }

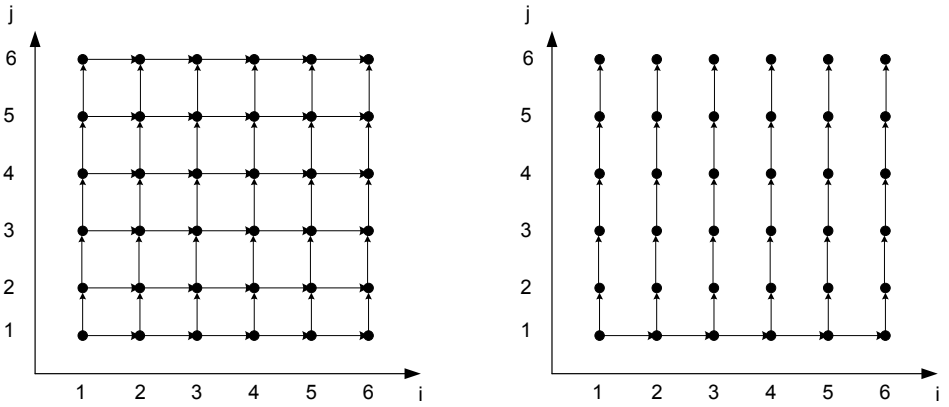
   3.2 Warunek: R2 – R2' =∅ i R2'-R2 =∅, jest prawdziwy. Nie kontynuuj obliczeń dla R2.

S := {R1’, R2}

```
R1’ := { [1,j,1] -> [1,j+1,1] :i' = 1 && 1 <= j < n && 1 <= n }
R2 := {[i,j,1] -> [i+1,j,1] : 1 <= i < n && 1 <= j <= n}
```

Koniec przykładu.

Topologia grafu zależności została zredukowana z grafu ogólnego do drzewa. Poniżej zilustrowano (rys. 3.6) przestrzeń iteracji oraz zależności przed i po wykonaniu algorytmu.



Rys. 3.6. Przestrzeń iteracji i zależności dla S := {R1, R2} i S' := {R1',R2}, n = 6, po wykonaniu algorytmu 3.7 [op. własne].

3.5.6. Generowanie kodu skanującego niezależne łańcuchy z pętli o topologii grafu lub drzewa

Przedstawiony w niniejszym punkcie algorytm redukcji topologii to rozwiązanie dedykowane dla topologii grafów i drzew. Jego wynikiem jest topologia łańcucha. Ideą algorytmu jest próba znalezienia ścieżki (łańcucha) zawierającego wszystkie iteracje grafu lub drzewa.

Algorytm 3.8 Generowanie kodu skanującego niezależne łańcuchy z pętli o topologii grafu lub drzewa

Wejście: zbiór relacji zależności R_i, $1 \leq i \leq q$, zbiór początków fragmentu kodu Sour i relacja R_UCS otrzymane w wyniku działania algorytmu 3.1 dla podgrafu SCC,

Wyjście: Kod przebierający niezależne łańcuchy z podgrafu SCC o zredukowanej topologii (grafu lub drzewa do łańcucha) w porządku leksykograficznym.

1. Oblicz $R := \bigcup_{1 \leq i \leq q} R_i$
2. Wykonaj poniższy kod

```
genOuterLoops (wejście: Sources; wyjście: OuterLoops, L_I);
  DLA KAŻDEGO I z listy L_I WYKONAJ
    JEŻELI R_UCS=∅
      S_SLICE = R*(I)
    W PRZECIWNYM WYPADKU
      S_Slice := R*(R_UCS*(I)) /* oblicz zbiór elementów należących do fragmentu kodu */
    // koniec JEŻELI
  R':={[e]→[e']: e,e' ∈ S_Slice  &&  e ≺ e'  && (!∃ e'' : e'' ∈ S_Slice & e ≺ e'' ≺ e') }
```

```
/* to ograniczenie gwarantuje, że zależności będą tworzyć topologię łańcucha, a jego
elementy są uporządkowanymi leksykograficznie instrukcjami fragmentu kodu */
// koniec DLA KAZDEGO
genWhile (wejście: OuterLoops, I, R'; wyjście: WhileLoop);
```

gdzie:

- funkcja *genOuterLoops* jest funkcją opisaną w krok 4.1 algorytmu 3.3,
- genWhile(wejście: OuterLoops, I, R'; wyjście: WhileLoop) jest funkcją generującą kod *WhileLoop* w poniższej postaci:
  1. Kod wewnętrznych pętli *WhileLoop*

```
S = I;
do {  // WYKONAJ-DOPÓKI
    stk(I'); /* instrukcja k pętli (k jest zdfiniowana przez dodaktową zmienną
    zdefiniowaną w wektorze I opisujący wykonanie iteracji I' (I' reprezentuje n-1
    zmiennych w krotce I) */
     S=R'(S);
}
while (S ∈ domain R') ;
```

  2. Wstaw pętlę *WhileLoop* do gniazd pętli *OuterLoops* odpowiadających sparametryzowanemu wektorowi I

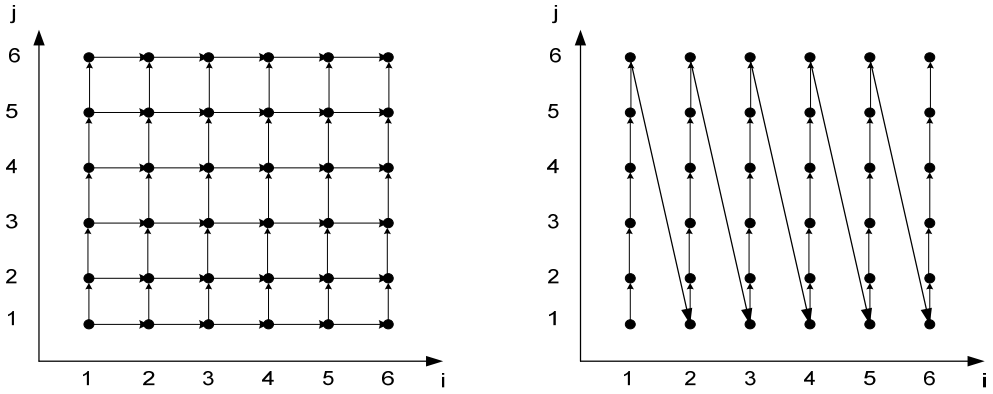
Dowód: Zbiór S_SLICE zawiera zbiór wszystkich iteracji tranzytywnie zależnych od iteracji I, wraz z leksykograficznie większymi początkami krańcowym (dzięki zastosowaniu relacji R_UCS). Do skonstruowania pętli *while* wykorzystano relację R'. Powstaje ona na bazie iloczynu kartezjańskiego wszystkich iteracji ze zbioru S_SLICE i dodaniu warunku kolejności w porządku leksykograficznym. Dodatkowo dodawany jest warunek, że nie może istnieć żadna pośrednia iteracja w tej relacji pomiędzy krotkami dziedziny i zakresu. Oznacza to wyeliminowanie wspólnych początków i końców zależności oraz transformację topologii do łańcucha. Ponieważ zbiór S_SLICE powstaje na bazie wszystkich relacji zależności oraz dodano warunek porządku leksykograficznego wykonania iteracji, wygenerowany kod jest semantycznie identyczny z kodem reprezentującym pętlę wejściową.

Przykład:

Powróćmy do przykładu z poprzedniego punktu 3.5.5. W wyniku działania algorytmu 3.8 otrzymano relację R':

```
R' := {[i,n,1] -> [i+1,1,1] : 1 <= i < n} union  {[i,j,1] -> [i,j+1,1] : 1 <= i <= n && 1 <= j < n}
```

Relacja R' tworzy topologię łańcuchu i można wygenerować pętlę *while* dla tej topologii (rys. 3.7).



Rys. 3.7. Przestrzeń iteracji i zależności dla S := {R1, R2} i S' := {R'}, n = 6, po wykonaniu algorytmu 3.8 [op. własne].

Koniec przykładu.

Główną zaletą kodu równoległego w oparciu o pętlę *while* jest możliwość ekstrakcji równoległości w przypadku, gdy niemożliwe jest wyznaczenie afinicznego zbioru iteracji fragmentów kodu. Wszystkie znane autorowi techniki generowania kodu ograniczone są w zastosowaniu do zbiorów afinicznych [11], [29], [38], [39], [56], [65], [82], [85]. Zastosowanie pętli *while* jest jednak kosztowniejsze od pętli *for*, przebieranie iteracje w czasie wykonania wiąże się z dodatkowym narzutem obliczeniowym i pamięciowym. W pracy [16] zbadano wpływ kosztów obsługi kodu równoległego opartego o pętle *while* oraz obliczeń zawartych w ciele pętli na przyspieszenie i efektywność obliczeń. Zauważono, że im więcej obliczeń znajduje się w pojedynczej iteracji pętli, tym koszty są niższe, natomiast przyspieszenie rośnie. Ważna jest także próba dokonania redukcji topologii. Koszty wykonania kodu równoległego o topologii łańcucha są najmniejsze. W niniejszym podrozdziale zaproponowano trzy algorytmy (3.6-3.8) umożliwiające redukcję topologii zależności.

Przedstawione do tej pory algorytmy pozwalają na ekstrakcję równoległości w postaci niezależnych fragmentów kodu. Nie rozwiązany pozostaje jednak problem, gdy w wyniku algorytmu 3.1 wyszukiwania początków uzyskany zostanie tylko jeden początek, czyli jeden, sekwencyjny fragment kodu. W następnym punkcie przedstawiono algorytm umożliwiający ekstrakcję równoległości w postaci fragmentów kodu z synchronizacją.

3.6. Wyszukiwanie fragmentów kodu z synchronizacją

Równoległość zawarta jest także w pojedynczym fragmencie kodu. Wiele popularnych i aktualnie najskuteczniejszych rozwiązań, w tym transformacje afiniczne [35], [38], [39], [65], [66], [67], nie umożliwiają jednak jej ekstrakcji. Oprócz samego podziału przestrzeni iteracji, należy wyznaczyć zależności pomiędzy nimi. Uwzględnienie wszystkich zależności w pętli umożliwia przesyłanie komunikatów. Duże znaczenie odgrywa także ziarnistość obliczeń. Zbyt duża liczba bloków wiąże się z dużą liczbą komunikatów. Mała liczba dużych bloków iteracji to ryzyko wystąpienia znacznych czasów przestoju [82]. W tym punkcie opisano algorytm wyznaczenia kodu z synchronizacją z przykładem, opisano implementację funkcji przesyłania komunikatów oraz poruszono problem aglomeracji (redukcji liczby komunikatów) i porządku wykonania iteracji.

3.6.1. Algorytm wyszukiwania fragmentów kodu z synchronizacją

Algorytm 3.9 Wyszukiwanie fragmentów kodu z synchronizacją

Wejście: zbiór S1 relacji zależności, tworzących przynajmniej dwa fragmenty kodu; zbiór S2 relacji zależności, które zostaną użyte do synchronizacji fragmentów kodu uzyskanych z relacji zależności z zbioru S1; relacje Rel1 i Rel2 będące uniami wszystkich relacji odpowiednio z zbioru S1 i S2; dla zbiorów S1 i S2 musi być spełniony warunek: (Domain (Rel2) union Range (Rel2)) $\in$ (Domain (Rel1) union Range (Rel1)).

Wyjście: kod przebierający równoległe fragmenty kodu z synchronizacją

1. Wygeneruj kod na podstawie relacji zależności za pomocą algorytmów 3.1, 3.2 lub innych znanych technik [6], [11], [12], [28], [85]; wyszukaj wszystkie zbiory I_j, reprezentujące przestrzeń iteracji instrukcji st_j, j=1,2,...,q; q – jest liczbą instrukcji w pętli,
2. Oblicz zbiór: SET_RECV_i = (Domain(Rel1) $\cup$ Range(Rel1)) $\cap$ Range($R2_i$), dla każdej relacji $R2_i$ z zbioru S2, i=1,2,...,n; n jest liczbą relacji z zbioru S2,
3. Oblicz zbiór: SET_SEND = $Rel2^{-1}$(Domain(Rel1)$\cup$ Range(Rel1)) $\cap$ Range(Rel2),
4. Przed każdą instrukcją st_j, wygenerowanego kodu w kroku 1 oraz dla każdej relacji R2, jeżeli istnieje część wspólna zbiorów SET_RECV_i i I_j wstaw poniższy kod:

```
if(I ∈ SET_RECV_i)
   recv(R2_i^-1(I)); // I jest wektorem iteracji zdefiniowanym przez pętlę otaczającą instrukcę st_j
```

5. Po każdej insrtukcji st_j w wygenerowanym kodzie w kroku 1, jeżeli istnieje część wspólna zbiorów SET_SEND I I_j, wstaw poniższy kod:

```
if(I ∈ SET_SEND)
    send(I); // I jest wektorem iteracji zdefiniowanym przez pętlę otaczającą instrukcę st_j
```

Przykład:

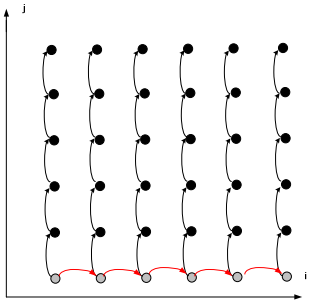
Niech będzie dana pętla:

```
for( i=1; i<=n; i++)
  for( j=1; j<=n; j++)
     a(i,j) = a(i,j-1) + a(i-1,1);
```

Pętla zawiera dwie relacje zależności, podzielone na dwa zbiory S1 i S2:

{S1} = R1 = {[i,j] -> [i,j+1] : 1 <= i <= n && 1 <= j < n},

{S2} = R2 = {[i,1] -> [i+1,1] : 1 <= i < n && j=1}.



Rys. 3.8 Przestrzeń iteracji pętli, na czerwono zaznaczono relację R2,na czarno R1[27].

1. Wygenerowany kod oraz wektory I na podstawie zbioru iteracji relacji R1:

```
if (n >= 2) {
for(t1 = 1; t1 <= n;  t1++) {
   s(t1,1) ; /* a(t1,1) = a(t1,0) + a(t1-1,1); */
   if (n >= t1 && t1 >= 1) {
     for(t2 = 2; t2 <= n; t2++) {
        s(t1,t2);  /* a(t1,t2) = a(t1,t2-1) + a(t1-1,1); */
     }
   }
```

I_1 = {[t1,1] : t1 >= 1 && t1 <=n },

I_2 = {[t1,t2 : t1 >= 1 && t1 <=n && t2 >= 2 && t2 <=n}

2, 3.

SET_RECV_1={[i,1]:2<=i <= n}

SET_SEND={[i,1] : 1 <= i <= n-1}.

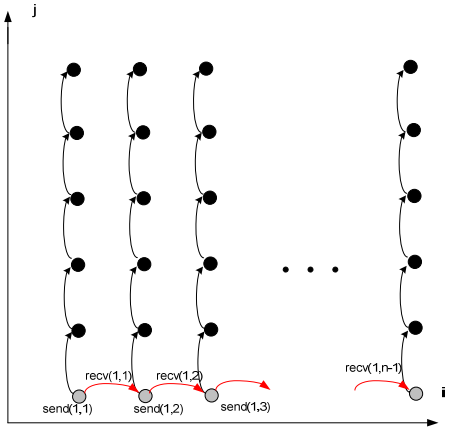
Dla I_2 powyższe zbiory są puste.

4, 5.

```
if (n >= 2) {
par for(t1 = 1; t1 <= n;  t1++) {
  if(2 <= t1 && t1 <= n)
    recv(t1-1,1);
```

```
s(t1,1) ;       // st1
if(1 <= t1 && t1 < n)
   send(t1,1);
if (n >= t1 && t1 >= 1) {
  for(t2 = 2; t2 <= n; t2++) {
    s(t1,t2);  //st2
  } } }
```

Na rysunku 3.9 zilustrowano wyznaczone fragmenty kodu z synchronizacją.



Rys. 3.9. Równoległe wątki z synchronizacją [27].

3.6.2. Implementacja funkcji przesyłania komunikatów w środowisku OpenMP

Środowisko OpenMP [31] nie udostępnia mechanizmu przesyłania komunikatów. Idea takich funkcji została zaczerpnięta z narzędzi PVM (ang. *Parallel Virtual Machine*) [34], [44], i MPI (ang. *Message Passing Interface*) [43], [87], które umożliwiają przekazywanie komunikatów w środowisku rozproszonym (ang. *message-passing*) [74]. Podstawą działania funkcji *send* i *recv* w algorytmie 3.9 są zamki [51]. Umożliwiają one modyfikację zmiennej z wykluczeniem równoczesnego odczytu. Zmienna ta informuje, czy dana iteracja została wykonana. Jeżeli iteracja nie została jeszcze wykonana przez inny wątek, pozostałe czekające wątki są blokowane. Biblioteka OpenMP udostępnia zestaw funkcji zamków [31]. Jednak w standardzie OpenMP można zastosować tylko czekanie aktywne. Oznacza to, że wątek czekający wykonuje w nieskończoność pętlę, w której sprawdzana jest wartość zmiennej. Wiąże to się z marnotrawieniem cyklów procesora i gorszą wydajnością kodu równoległego. Dlatego zastosowano drugą implementację funkcji w oparciu o wątki Posix [30], [72], [86] z wykorzystaniem zmiennych warunkowych [62]. Wątek, który oczekuje na wykonanie iteracji, jest „usypiany” (przekazywany do kolejki procesów czekających na

wykonanie) i nie zajmuje zasobów obliczeniowych procesora. Po wykonaniu iteracji, wątki oczekujące są „budzone" i przywracane do działania. W pracy [27] dokonano porównania obu rozwiązań. Zastosowanie zmiennych warunkowych wiąże się z dodatkowym narzutem pamięciowym, jednak ostatecznie badania wykazały, że kod z zastosowaniem wątków Posix jest wydajniejszy. Poniżej w tabeli 3.1 zaprezentowano funkcje *send* i *recv* w obu wersjach implementacji. Funkcje zostały zaimplementowane w języku C.

Tab. 3.1. Kod funkcji przesyłania komunikatów

	Funkcja send	**Funkcja recv**

OpenMP	void send(int t1, int t2,...,int tn)
	{
	struct s_synch *set_c =
	&SET_SEND[t1][t2]...[tn];
	omp_set_lock(&set_c->mutex);
	set_c->executed = 1;
	omp_unset_lock(&set_c->mutex);
	}
		void recv(int t1, int t2,...,int tn)
		{
		struct s_comm *set_c =
		&SET_RECV[t1][t2]...[tn];
		while(1)
		{
		omp_set_lock(&set_c->mutex);
		if(!(set_c->executed))
		omp_unset_lock(
		&set_c->mutex);
		else {
		omp_unset_lock(
		&set_c->mutex);
		break;
		}
		}
		}

Posix Threads	void send(int t1, int t2,...,int tn)
	{
	struct s_comm *set_c =
	&SET_SEND[t1][t2]...[tn];
	pthread_mutex_lock(
	&set_c->mutex);
	set_c->executed = 1;
	pthread_cond_broadcast(
	&set_c->cond);
	pthread_mutex_unlock(
	&set_c->mutex);
	
	}
		void recv(int t1, int t2,...,int tn)
		{
		struct s_comm *set_c =
		&SET_RECV[t1][t2]...[tn];
		pthread_mutex_lock(&set_c->mutex);
		if(!(set_c->executed))
		pthread_cond_wait(
		&set_c->cond, &set_c->mutex);
		pthread_mutex_unlock(
		&set_c->mutex);
		}

Argumentami funkcji są koordynaty wektora iteracji I. Zmienne informujące o wykonaniu iteracji I zawarte są w tablicy SET_SEND. Dostęp do jej komórek wiąże się z wzajemnym wykluczaniem wątków. W funkcji odbierania komunikatu w implementacji OpenMP znajduje się pętla, sprawdzająca wartość zmiennej. Dodatkowo wydajność pogarsza obowiązek zakładania i zwalniania zamka w jej każdej iteracji. Problem rozwiązany został przez zmienne warunkowe w implementacji Posix. Funkcja *pthread_cond_wait* „usypia" wątek. Jego działanie wznowi wątek, który wykonał żądaną iterację.

3.6.3. Aglomeracja fragmentów kodu z synchronizacją i kolejność wykonania ich iteracji

Wykonanie instrukcji *send* i *recv* wiąże się z dodatkowym narzutem obliczeniowym oraz pamięciowym. Kosz wykonania tych operacji może niwelować korzyści z przyspieszenia obliczeń równoległych. Dlatego warto zredukować liczbę komunikatów i koszt synchronizacji poprzez zastosowanie aglomeracji obliczeń [40]. W tym celu utworzono zbiór nowych fragmentów kodu, tzw. **makro-fragmentów kodu**. Ideą aglomeracji obliczeń jest wyeliminowanie komunikacji wewnątrz makro-fragmentu kodu oraz uzyskanie kodu o większym stopniu gruboziarnistości.

Do kodu wygenerowanego algorytmu 3.9. dodawana jest pętla przebierająca N makro-fragmentów kodu. Granice pętli wewnętrznej zostają zmienione na nowe zmienne sparametryzowane *lb* i *ub.* Pętla wewnętrzna przebiera iteracje makro-fragmentu kodu i są wykonane sekwencyjnie.

W celu uniknięcia synchronizacji wewnątrz fragmentów kodu zostają zmienione warunki instrukcji *if* (krok 4,5 w algorytmie 3.9). Sprawdzają one, czy wektory iteracji I, obliczone za pomocą wyrażeń $R2_i^{-1}(I)$ i $R2(I)$ należą do tego samego makro-fragmentu-kodu. Jeżeli tak, synchronizacja jest zbędna. Zmiany zaznaczono na poniższym kodzie przykładu z punktu 3.6.1.

```
par for w = 1 to N {    //nowa pętla przebierająca N makro-fragmentów kodu
 // oblicz nowe granice lb i ub (dolną i górną odpowiednio)
pack = ((o_ub-1)+1)/N,
lb = pack *(w-1)+o_lb;
ub = (w==N) ? n : pack *w;
  // gdzie o_lb= 1, o_ub=n to dolna i górna granica petli zewnętrznej wygenerowanej w
 algorytmie // 3.9. kroku 1
 if (n >= 2) {
 for(t1 = lb; t1 <= ub;  t1++) {
```

```
  if(2 <= t1 && t1 <= n && !(t1-1>=lb && t1-1 <= ub)) /* odbierz komunikat od innego makro-
fragmentu kodu */
     recv(t1-1,1);
  s(t1,1) ;       // st1
  if(1 <= t1 && t1 < n  && !(i+1>=lb && i+1 <= ub)) /* wyślij komunikat do innego makro-
fragmentu kodu */
      send(t1,1);
  if (n >= t1 && t1 >= 1) {
    for(t2 = 2; t2 <= n; t2++) {
      s(t1,t2);  //st2
    } } }
```

Liczba komunikatów została zmniejszona z *n-1* do *N-1*, gdzie $N < n$. Ilustruje to rysunek 3.10 a).

Kolejność przebierania iteracji w wygenerowanym kodzie w kroku 1 algorytmu 3.9. jest istotna dla uzyskania żądanej równoległości. Analizując rysunek 3.6.3.a), można zauważyć, że po zastosowaniu aglomeracji wykonanie makro-fragmentów kodu staje się niemal sekwencyjne. Następny makro-fragment kodu, składający się z *n*(ub-lb+1)* iteracji, zostaje wykonany po *n*(ub-lb)* iteracji poprzedniego.

Rozwiązaniem jest zmiana kolejności wykonania iteracji z porządku leksykograficznego na harmonogramowanie iteracji (ang. free-scheduling). Oznacza to, wykonanie iteracji w danej jednostce czasu [35].

Topologią pętli jest drzewo (rys. 3.10 b). W algorytmie 3.4 zaproponowano przebieranie iteracji z harmonogramowaniem. W kroku 1 algorytmu 3.9 zamiast generowania kodu w porządku leksykograficznym zastosowano przebieranie iteracji z pętlą *while*. Po zastosowaniu aglomeracji obliczeń, otrzymano poniższy kod:

```
par for w = 1 to N {
 S' = Ø;
 for t=lb to ub {
    I = [t,1]
    Dodaj I do S';
 }
 while(S' != Ø) {
S_tmp= Ø;
foreach(vector I=[i,j] in S') {  /* dla każdego wektora I=[i,j] z zbioru S' */
 if(j==1 && 2 <= i && i <= n && !(i-1>=lb && i-1 <= ub)) /* odbierz komunikat od innego
makr-fragmentu kodu */
  recv(i-1,1);
 s1(I);
 if(j==1 && 1 <= i && i < n && ( !(i+1>=lb && i+1 <= ub) ) ) /* wyślij komunikat do innego
makr-fragmentu kodu */
```

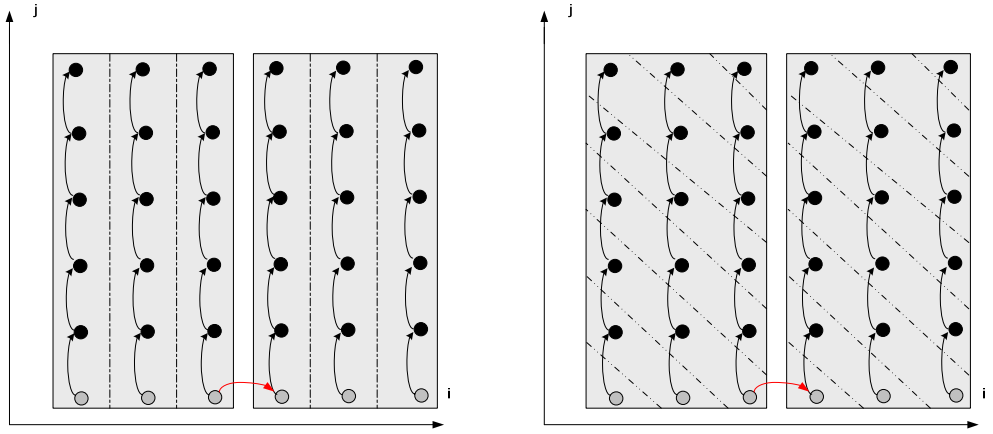
```
  send(i,1);
if(1 <= j && j < n && 1 <= i && i <= n){   /* jeżeli R1(I ∈ domain R1*/
  dodaj J=[i,j+1] do S_tmp;              /* J=[ip,jp]= R2(I) */
if(j==1 && 1<=i && i<n && lb<=i+1 && i+1>=ub) /* jeżeli  R2(I) ∈domain R2
 i  I∈  iteracji makro-fragmentu kodu */
  dodaj J=[i+1,j] do S_tmp;              /* J=[ip,jp]= R2(I) */
  }
   S' = S_tmp;
}
```

W przeciwieństwie do porządku leksykograficznego, gdzie wykonanie kolejnego makro-fragmentu kodu nastąpiło po *n*(ub-lb)* iteracji, harmonogramowanie pozwala na wykonanie następnego makro-fragmentu po $(ub - lb + 1)^2 - \sum_{z=1}^{ub-lb} z$ iteracji.

Dla przykładu, niech n = 100. Zastosowano aglomerację i dokonano podziału przestrzeni iteracji na N=10 makro-fragmentów kodu. Zatem ub-lb+1 = 10, natomiast makro-fragment kodu zawiera 1000 iteracji. Rozpoczęcie obliczeń następnego makro-fragmentu kodu w obu wariantach po wykonaniu liczby iteracji:

- aglomeracja z porządkiem leksykograficznym: 900 iteracji,
- aglomeracja z harmonogramowaniem za pomocą pętli *while*: 55 iteracji.

Porównanie zilustrowano na rysunku 3.10, dla N=2, n=6.



Rys. 3.10. Aglomeracja z kolejnością przebierania iteracji a) w porządku leksykograficznym b) harmonogramowanie iteracji [27].

3.7. Podsumowanie

W niniejszym rozdziale zaprezentowano algorytmy pozwalające na wyznaczenie niezależnych i wymagających synchronizacji fragmentów kodu dla pętli dowolnie zagnieżdżonych. Pierwszym zaproponowanym rozwiązaniem jest wyznaczenie

początków niezależnych fragmentów kodu (algorytm 3.1). W następnym algorytmie 3.2 obliczany jest afiniczny zbiór iteracji fragmentów kodu i generowana pętla równoległa. Zrównoleglenie pętli za pomocą tego algorytmu nie wymaga analizy topologii zależności.

Pomyślne zastosowanie algorytmu 3.2 w ogólnym przypadku dla pętli niejednorodnych jest ograniczone możliwościami wykorzystanego w implementacji narzędzia Omega Calculator. Wyznaczenie niedokładnego tranzytywnego domknięcia relacji zależności uniemożliwia uzyskanie maksymalnej równoległości oraz nie gwarantuje poprawności wyniku. W algorytmach 3.3-3.5 zaproponowano podejście wyznaczania iteracji, oparte na przebieraniu iteracji w czasie ich wykonania za pomocą pętli *while* (dopóki). Wydajność kodu oraz zapotrzebowanie na dodatkowe obliczenia i pamięć podyktowane jest topologią zależności. Możliwość uproszczenia topologii pozwala na uzyskanie wydajniejszego kodu równoległego. Zaproponowano trzy algorytmy 3.6-3.8 do redukcji topologii.

Istnieje klasa pętli, dla których zastosowanie algorytmu 3.1 pozwala na uzyskanie tylko jednego początku niezależnego fragmentu kodu. Pokazano, że równoległość może znajdować się także wewnątrz pojedynczego fragmentu. Zaproponowano algorytm 3.9 umożliwiający ekstrakcję równoległości w pojedynczym fragmencie kodu. Ideą algorytmu jest podzielenie go na nowy zbiór fragmentów kodu wymagających synchronizacji. Zaproponowano aglomerację obliczeń, w celu minimalizacji kosztów komunikacji i zbadano wpływ porządku wykonania iteracji na czasy przestojów makro-fragmentów kodu. Stwierdzono, że uzyskanie optymalnego wyniku czasowego pętli równoległej z synchronizacją zależy od: ziarnistości obliczeń (stosunek liczby komunikatów do liczby iteracji w fragmencie kodu), możliwości zastosowania aglomeracji i wyboru kolejności przebierania iteracji.

W następnym rozdziale zbadano zakres stosowalności algorytmów i jakość wygenerowanego kodu z wykorzystaniem dwóch zestawów pętli testowych.

4. BADANIA EKSPERYMENTALNE

W celu przeprowadzenia badań eksperymentalnych wybrane zostały zestawy testów NAS (ang. *NAS Parallel Benchmarks – NPB*) [8], [50], [90] oraz UTDSP (ang. *University of Toronto Digital Signal Processing*) [61], [79].

Pierwszy z nich to zbiór aplikacji opracowanych przez NASA do szacowania wydajności systemów równoległych. Zestaw NPB zawiera źródła pięciu jąder oraz trzech aplikacji CFD (ang. *Computional Fluid Dynamics* – symulacja dynamiki przepływu). Specyfikacja testów NAS udostępniona jest w postaci algorytmów przedstawiająca problem w taki sposób, aby istniało jedno unikatowe rozwiązanie i możliwe było zweryfikowanie poprawności wyników. Kwestia implementacji, wyboru struktur danych oraz sposobu użycia pamięci pozostaje otwarta dla programisty w celu uniezależnienia się od konkretnej architektury systemowej. Zbiór testów NAS jest często określany jako wyznacznik wydajności komputerów sekwencyjnych i równoległych. Do badań wykorzystano wersję 3.2 [48].

Drugim badanym zestawem testów jest UTDSP opracowany na uniwersytecie w Toronto. UTDSP został stworzony do oszacowania jakości kodu wygenerowanego przez kompilator języka wysokiego poziomu (np. język C) stosowanego w programowalnych procesorach do cyfrowej obróbki sygnałów (DSP – ang. *Digital Signal Processing*). Oszacowanie te zostało użyte w rozwijaniu specjalnych optymalizacji kompilatora w celu polepszenia jakości kodu i modyfikacji pod kątem architektury docelowego procesora dla uproszczenia zadania kompilatora. Aplikacje DSP napisane pierwotnie w języku niskiego poziomu (assembler) w celu stworzenia zestawu UTDSP zostały przepisane w języku C. Zestaw UTDSP został podzielony na dwie klasy programów: 6 jąder (ang. kernels) oraz 10 aplikacji reprezentujących popularne obliczenia z zakresu DSP. Do badań skorzystano z wersji 97.02.12.

W poniższej tabeli 4.1 przedstawiono jądra i aplikacje zestawów testów NAS i UTDSP. W kolumnach zamieszczono ich nazwę oraz krótki opis problemu obliczeniowego.

Tab. 4.1. Jądra i aplikacje w badanych zestawach testów NAS i UTDSP

NAS	**Jądra**	
	FT	Test pomiaru czasu obliczenia równań trójwymiarowej różniczki cząstkowej za pomocą prostej i odwrotnej Szybkiej Transformaty Fouriera
	MG	Test, w którym rozwiązywany jest układ równań Poisson'a w celu zbadania przemieszczenia danych na krótkie i długie dystansy
	CG	Test oparty o gradient sprzężenia (ang. *Conjugate Gradient*) oblicza przybliżoną wartość własną dużej, nieregularnej i rzadkiej macierzy. W testach zawarto nieustaloną strukturę komunikacji i obliczeń przy użyciu macierzy z losowymi lokalizacjami wspisów
	EP	Test (ang. *Embarrassingly Parallel benchmark*) zawiera losową parę odchyleń Gaussowskich zgodnie z zadanym schematem. Celem testu jest oszacowanie górnej granicy wydajności obliczeń zmiennoprzecinkowych.
	UA	Test (ang. *Unstructured Adaptive*) mierzy wpływ ciągłych i nieregularnych odwołań do pamięci.
	Aplikacje CFD	
	BT	Test do rozwiązania trójwymiarowego układu równań Navier-Stokes'a. Skończona liczba różnic rozwiązań problemu oparta jest o przybliżoną faktoryzację metody kierunków naprzemiennych (ang. *Alternating Direction Implicit - ADI*), rozdzielającej wymiary x,y,z.
	SP	Skończona liczba różnic rozwiązań problemu bazuje na przybliżonej faktoryzacji schematu Beam-Warming'a, która rozdziela wymiary x,y,z.
	LU	Symulacja aplikacji, w której zastosowano symetryczną metodę kolejnych nadrelaksacji (ang. *symmetric successive over-relaxation - SSOR*) do rozwiązania siedmio blokowego diagonalnego systemu poprzez podział na dolne i górne trójkątne systemy.
	LU-HP	hiperpłaszczyznowa wersja testów LU
UTDSP	**Jądra**	
	FFT	Szybka transformata Fouriera w wersji Radix-2 DIT (decymacja w czasie).
	FIR	Filtr o skończonej odpowiedzi impulsowej (ang. *Finite Impulse Response filter - FIR filter*)
	IIR	Filtr o nieskończonej odpowiedzi impulsowej (ang. *Infinite Impulse Response filter - FIR filter*)

	LATNRM	Znormalizowany filtr opóźniający (ang. *Normalized lattice filter*)
	LMSFIR	Adaptacyjny filtr FIR z zastosowanym algorytmem najmniejszych średnich kwadratów (ang. *Least mean squares -LMS*)
	MULT	Mnożenie macierzy
	Aplikacje DSP	
	G721_A, G721_B	Dwie implementacje kodera mowy ITU G.721 ADPCM
	V32.MODEM	Koder i dekoder modemu V.32
	ADPCM	Adaptacyjna różnicowa modulacja kodowo-impulsowa kodera mowy
	COMPRESS	Kompresja obrazu z wykorzystaniem dyskretnej transformacji cosinusowej (ang. *Discrete Cosine Transform - DCT*)
	EDGE_DETECT	Wykrywanie krawędzi 2D za pomocą operacji splotu i filtry Sobela
	HISTOGRAM	Rozszerzanie obrazu za pomocą wyrównania histogramu
	LPC	Predykcja liniowa LPC w kodowaniu mowy
	SPECTRAL	Analiza widmowa wykorzystująca uśrednianie periodogramu
	TRELLIS	Dekoder Trellis'a

4.1. Kryteria wyboru pętli do badań eksperymentalnych

Wybór pętli z zestawów NAS i UTDSP dokonano na podstawie struktury pętli oraz relacji opisujących zależności. Pętle posiadają górną i dolną granicę w postaci stałej i sparametryzowanej oraz są dowolnie zagnieżdżone. W ciele pętli dopuszczalne są odwołania do zmiennych skalarnych oraz zmiennych tablicowych. Pętla nie może zawierać instrukcji skoku, instrukcji zmieniających krok pętli oraz odwołań do niezdefiniowanych zewnętrznych funkcji, które mogą generować dodatkowe nieznane zależności danych.

Zestaw pętli NAS zawiera 431 pętli. Uwzględniając powyższe kategorie zakwalifikowano 257 pętli, co stanowi około 60% pętli. Następnym kryterium jest obecność w pętli zależności danych. 149 pętli z 257 posiada zależności co stanowi 58% wybranych pętli.

Zestaw UTDSP zawiera 77 pętli. Odrzucając pętle zawierające instrukcje skoku (np. goto, continue, break), wywołania funkcji oraz pętle nie posiadające żadnych zależności danych otrzymano 34 pętli (około 45%). Statystyki zawarto w tabeli 4.2.

Tab. 4.2. Statystyki ilościowe i procentowe dla testów NPB i UTDSP

Nazwa testu	Wszystkich pętli	Pętli spełniających kryterium struktury		Pętli zakwalifikowanych do analizy		Pętli odrzuconych	
		Ilość	Procent (%)	Ilość	Procent (%)	Ilość	Procent (%)
NAS Parallel Benchmark							
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**
UTDSP Benchmark							
fft	2	2	100	0	0	2	100
fir	2	2	100	2	100	0	0
iir	1	1	100	1	100	0	0
latnrm	3	3	100	3	100	0	0
lmsfir	3	3	100	2	66	1	33,3
mult	2	2	100	2	100	0	0
adpcm	1	0	0	0	0	1	100
compress	4	3	75	3	75	1	25
edge_det.	4	4	100	4	100	0	0
histogram	4	4	100	3	75	1	25
lpc	10	10	100	9	90	1	10
spectral	9	3	33,3	1	11,1	8	88,9
trellis	10	4	40	2	20	8	80
g721, g722	9	1	11,1	1	11,1	8	88,9
v32_modem	4	0	0	0	0	4	100
jpeg	9	1	11,1	1	11,1	8	88,9
Zbiorczo	**77**	**43**	**55,8**	**34**	**44,2**	**43**	**55,8**

Kod równoległy udało się uzyskać dla 90 pętli z testów NAS (61% wszystkich pętli z zależnościami) i 18 pętli z testów UTDSP (55% wszystkich pętli z zależnościami).

Dla pozostałych pętli nie dokonano zrównoleglenia z dwóch powodów:

- nie uzyskano kodu równoległego z powodu ograniczeń implementacyjnych algorytmów: wykorzystane narzędzie Omega Calculator nie umożliwia obliczenia operacji tranzytywnego domknięcia dla dowolnej relacji oraz zawodzi w operacjach dla złożonych relacji w ogólnym przypadku (NAS – 48 pętli, UTDSP – 11 pętli).
- uzyskano tylko jeden fragment kodu oraz z powodu braku równoległości nie dokonano zrównoleglenia pętli (NAS – 11 pętli, UTDSP – 5 pętli). Ekstrakcja równoległości z synchronizacją dla tych przypadków jest niemożliwa, ponieważ nie udało się utworzyć zbiór relacji S1 z wieloma wątkami (wszystkie relacje przyporządkowane zostały do S2). Przykład:

```
for i=1 to 100 do
  a(i) = a(i-1)
endfor
```

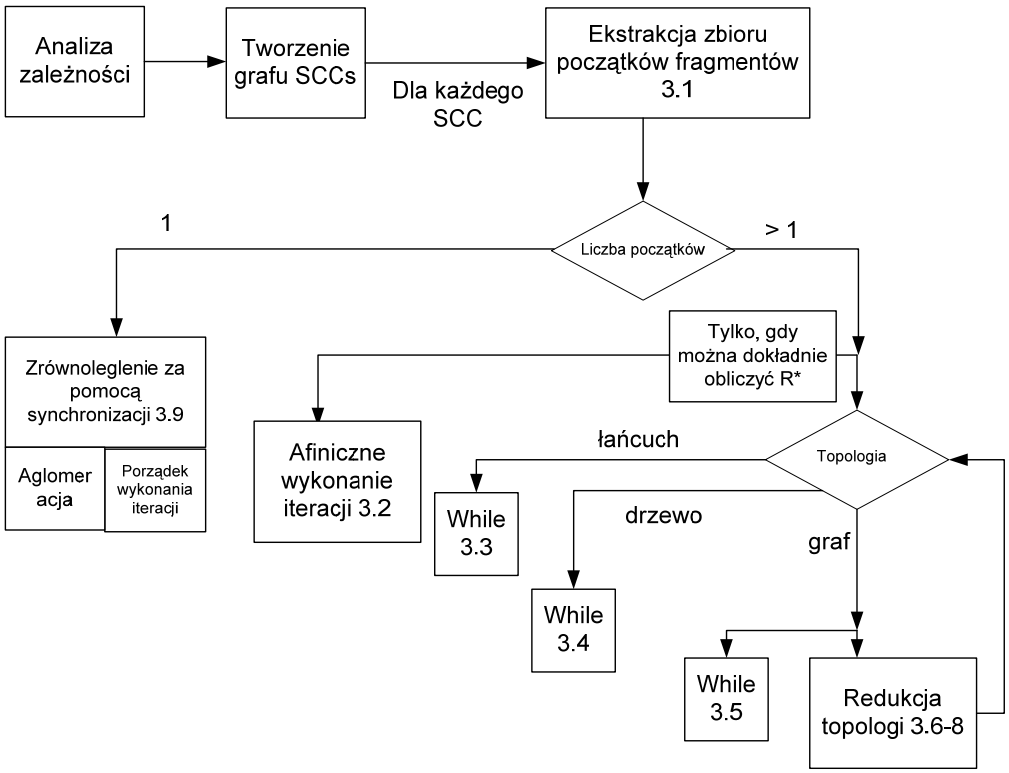
Istnieje tylko jedna relacja R := { [i]->[i+1] : 1<=i<=99}. Ponieważ relacja opisuje zależność z tylko jednym początkiem Sour :={[1]}, zostaje przyporządkowana do S2. Pętla nie zawiera więcej relacji zależności, które mogłyby zostać przyporządkowane do zbioru S1, na podstawie których tworzone są fragmenty kodu z synchronizacją.

4.2. Dobór algorytmów w zrównolegleniu pętli programowej

Zrównoleglanie pętli zaczyna się analizą zależności oraz utworzeniem grafów SCCs (ang. *strongly connected component*, ściśle połączony graf). Dla niektórych przykładów podział na graf SCCs był jedyną możliwością uzyskania kodu równoległego z powodu braku równoległości w poszczególnych SCCs lub niemożliwości zastosowania algorytmów z powodu ograniczeń implementacyjnych narzędzia Omega Calculator.

W następnym kroku dokonana jest ekstrakcja początków niezależnych fragmentów kodu za pomocą algorytmu 3.1 dla każdego grafu SCCs. W przypadku otrzymania tylko jednego fragmentu można dokonać zrównoleglenia za pomocą synchronizowanych fragmentów kodu z wykorzystaniem algorytmu 3.9 z wariantem aglomeracji i wybranego porządku wykonania iteracji. Jeśli niezależnych fragmentów kodu jest więcej do wyboru są: generator pętli przebierający iteracje z afinicznego zbioru 3.2 lub za pomocą pętli *while*. Algorytm 3.2 możliwy jest do zastosowania tylko w przypadku gdy obliczenie tranzytywnego domknięcia daje dokładną postać afiniczną.

W przypadku pętli *while* niezbędna jest analiza topologii. Zrównoleglenia w zależności od topologii można dokonać algorytmami 3.3, 3.4, 3.5 lub w przypadku topologii innej niż łańcuch dokonać próby jej redukcji 3.8. Rys. 4.1 przedstawia schemat doboru algorytmu w zależności od ekstrakcji początków fragmentów kodu i topologii.



Rys. 4.1 Schemat zrównoleglenia pętli i ścieżka doboru właściwych algorytmów.

W przypadku pętli, dla których wyznaczenie początków algorytmem 3.1 okazało się niemożliwe, poddano je dodatkowej modyfikacji. Najczęstszym problemem była niemożliwość obliczenia operacji tranzytywnego domknięcia za pomocą Omega Calculator. Spowodowane to było dużą liczbą relacji opisujące zależności. W badanych przykładach wprowadzono modyfikację polegającej na zamianie zmiennych skalarnych na zmienne tablicowe. Pozwoliło to na znaczną redukcję liczby zależności, obliczenie zbioru początków fragmentów kodu (ang. *slices*) i wygenerowanie pętli równoległej.

W przypadku jeśli zrównoleglenie przeprowadzono za pomocą kilku algorytmów, w badaniach ujęto analizę porównawczą wyników czasowych wykonania otrzymanych pętli równoległych.

W tabeli 4.3 przedstawiono nazwę pętli, topologię, liczbę fragmentów kodu oraz zakres stosowalności algorytmów dla badanych pętli. W kolumnie drugiej umieszczono liczbę relacji zależności pętli oryginalnej wejściowej. Dla niektórych pętli zawarto w kolumnie drugą liczbę, w przypadku gdy ilość relacji zależności została zredukowana

(transformacja tablicowania zmiennych – punkt 4.3 lub użycie algorytmów redukcji zależności zaproponowanych w pracy [85]).

Tab. 4.3 Charakterystyka badanych pętli i zakres stosowalności algorytmów do ich zrównoleglenia

Pętla	Liczba relacji zależn. / po redukcji	Topo-logia	Liczba fragmentów kodu	Algorytmy						
				Pocz	Przebieranie iteracji					
					cod	while				syn
				3.1	3.2	3.3	3.4	3.5	3.8	3.9
NAS Benchmark										
BT_error.f2p_2	110 / 3	g	5	●	●	○	○	○	●	○
BT_error.f2p_3	8 / 4	g	5	●	●	○	○	○	●	○
BT_error.f2p_5	34 / 3	g	5	●	●	○	○	○	●	○
BT_error.f2p_6	8 / 4	g	5	●	●	○	○	○	●	○
BT_initialize.f2p_2	42 / 10	g	N+1	●	●	○	○	○	●	○
BT_initialize.f2p_3	42 / 10	g	N+1	●	●	○	○	○	●	○
BT_initialize.f2p_4	42 / 10	g	N+1	●	●	○	○	○	●	○
BT_initialize.f2p_5	42 / 10	g	N+1	●	●	○	○	○	●	○
BT_initialize.f2p_6	42 / 10	g	N+1	●	●	○	○	○	●	○
BT_initialize.f2p_7	42 / 10	g	N+1	●	●	○	○	○	●	○
BT_initialize.f2p_8	3	ł	5	●	●	●	◙	◙	○	○
BT_initialize.f2p_9	2 / 1	ł	5	●	●	●	◙	◙	○	○
BT_rhs.f2p_1	46 / 1	ł	N3 >= 0 && N2 >=0 =>(N1+1)* (N2+1)* (N3+1)	●	●	●	◙	◙	○	○
BT_rhs.f2p_5	128 / 10	g	N1*N2*N3	●	●	○	○	●	●	○
CG_cg.f2p_3	1	ł	N1	●	●	●	◙	◙	○	○
CG_cg.f2p_4	11 / 2	g	N1 (*)	◘	◘	○	○	○	○	●
CG_cg.f2p_7	2	ł	2*N1*(N3-N2 + 1) +N1	●	●	●	◙	◙	○	○
FT_appft.f2p_1	47 / 6	g	N1	●	●	○	○	○	●	○
FT_auxfnct.f2p_2	1	ł	N3 >= 1 && N2 >= 1 => N1*N2*N3	●	●	●	◙	◙	○	○
LU_erhs.f2p_2	66	g	N4 >= N3+1 => 5 * (N9 - max(N7+1,2))	●	●	○	○	○	●	○
LU_erhs.f2p_3	683	g	32 (**)	○	○	○	○	○	○	○
LU_erhs.f2p_4	683	g	32 (**)	○	○	○	○	○	○	○
LU_erhs.f2p_5	683	g	32 (**)	○	○	○	○	○	○	○
LU_HP_erhs.f2p_2	66	g	N4 >= N3+1 => 5 *(N9 - max(N7+1,2))	●	●	○	○	○	●	○
LU_HP_erhs.f2p_3	683	g	32 (**)	○	○	○	○	○	○	○
LU_HP_erhs.f2p_4	683	g	32 (**)	○	○	○	○	○	○	○
LU_HP_erhs.f2p_5	683	g	32 (**)	○	○	○	○	○	○	○
LU_HP_lnorm.f2p.2	12 / 2	ł	5	Θ	○	●	○	○	○	○
LU_HP_pintgr.f2p_2	109 / 29	g	N2-N1+1	●	●	○	○	●	●	○
LU_HP_rhs.f2p_1	17 / 7	d	N1*N2*N3	●	●	○	●	◙	●	○
LU_HP_rhs.f2p_2	683	g	32 grafy SCC	○	○	○	○	○	○	○
LU_HP_rhs.f2p_3	683	g	32 grafy SCC	○	○	○	○	○	○	○
LU_HP_rhs.f2p_4	683	g	32 grafy SCC	○	○	○	○	○	○	○

LU_pintgr.f2p_2	109 / 29	g	N2-N1+1	●	●	○	○	●	●	○
LU_lnorm.f2p.2	12 / 2	ł	5	Θ	○	●	○	○	○	○
LU_rhs.f2p_1	17 / 7	d	N1*N2*N3	●	●	○	●	◙	●	○
LU_rhs.f2p_2	683	g	32 grafy SCC	○	○	○	○	○	○	○
LU_rhs.f2p_3	683	g	32 grafy SCC	○	○	○	○	○	○	○
LU_rhs.f2p_4	683	g	5*(N9 - max(N7+1,2))	Θ	●	○	○	○	○	○
MG_mg.f2p_1	1	ł	3	●	●	●	◙	◙	○	○
MG_mg.f2p_3	3	ł	N2-N1	●	●	●	◙	◙	○	○
MG_mg.f2p_9	3	ł	N2-N1	●	●	●	◙	◙	○	○
MG_mg.f2p_10	21	g	4 (**)	○	○	○	○	○	○	○
MG_mg.f2p_11	1	ł	N1-2	●	●	●	◙	◙	○	○
MG_mg.f2p_12	1	ł	N1-2	●	●	●	◙	◙	○	○
MG_mg.f2p_13	1	ł	N1-2	●	●	●	◙	◙	○	○
SP_error.f2p_2	110 / 4	ł	5	Θ	○	●	○	○	○	○
SP_error.f2p_3	8 / 3	g	5	●	●	○	○	○	●	○
SP_error.f2p_5	34 / 3	g	5	●	●	○	○	○	●	○
SP_error.f2p_6	8 / 3	g	5	●	●	○	○	○	●	○
SP_initialize.f2p_2	42 / 3	g	N+1	●	●	○	○	○	●	○
SP_initialize.f2p_3	42 / 3	g	N+1	●	●	○	○	○	●	○
SP_initialize.f2p_4	42 / 3	g	N+1	●	●	○	○	○	●	○
SP_initialize.f2p_5	42 / 3	g	N+1	●	●	○	○	○	●	○
SP_initialize.f2p_6	42 / 3	g	N+1	●	●	○	○	○	●	○
SP_initialize.f2p_7	42 / 3	g	N+1	●	●	○	○	○	●	○
SP_initialize.f2p_8	3	ł	10	●	●	●	◙	◙	○	○
SP_nivr	103 / 2	ł	2*N1*N2*N3	Θ	○	●	◙	◙	○	○
SP_rhs.f2p_1	64 / 8	g	N+1	●	●	○	○	○	○	○
UA_adapt.f2p_1	32	g	6 (**)	○	○	○	○	○	○	○
UA_adapt.f2p_2	8	ł	N4 >= 1 && N3 >= 1 && N2 >= 2 => 8*(N1*N3*N4)	●	●	●	○	○	○	○
UA_adapt.f2p_9	32	g	2 (**)	○	○	○	○	○	○	○
UA_adapt.f2p_10	22	g	3 (**)	○	○	○	○	○	○	○
UA_adapt.f2p_11	22	g	4 (**)	○	○	○	○	○	○	○
UA_diffuse.f2p_2	5 / 2	g	N (*)	◘	◘	○	○	○	○	●
UA_diffuse.f2p_3	4	ł	N4 >= 1 && N3 >= 1 && N2 >= 2 => N1*N3*N4	●	●	●	◙	◙	○	○
UA_diffuse.f2p_4	4	ł	N4 >= 1 && N3 >= 1 && N2 >= 2 => N1*N3*N4	●	●	●	◙	◙	○	○
UA_diffuse.f2p_5	4	ł	N4 >= 1 && N3 >= 1 && N2 >= 2 => N1*N3*N4	●	●	●	◙	◙	○	○
UA_mason.f2p_18	1	ł	3	●	●	●	◙	◙	○	○
UA_precond.f2p_3	4	ł	N2 >= 2 => N1-1	●	●	●	◙	◙	○	○
UA_precond.f2p_4	4	ł	N3 >= 2 && N2 >= 2 => N1*(N2-1)	●	●	●	◙	◙	○	○
UA_precond.f2p_5	29	g	N1	●	●	○	○	○	●	○
UA_setup.f2p_1	1	ł	N1	●	●	●	◙	◙	○	○

UA_setup.f2p_6	5 / 2	ł	(intDiv(lx1,2)-max(-N1+lx1+1,4)+1)+(intDiv(lx1-1,2)-max(- N2+lx1+1,1))+3	●	○	●	◙	◙	○	○
UA_setup.f2p_14	15 / 3	d	N1*N2*N3*N4	●	●	○	●	◙	●	○
UA_setup.f2p_15	19	d	N1	●	●	○	○	○	○	○
UA_setup.f2p_16	4	ł	N3 >= 2 && N2 >= 1 => N1*N2	●	●	●	○	○	○	○
UA_transfer.f2p_4	7	g	N2 >= 3 => N1	●	●	○	○	○	●	○
UA_transfer.f2p_7	4	ł	N2 >= 3 => N1	●	●	●	◙	◙	○	○
UA_transfer.f2p_9	4	ł	N2 >= 3 => N1	●	●	●	◙	◙	○	○
UA_transfer.f2p_11	11	g	N3 >= 2 && N2 >= 2 => N1*(N2-1)	●	●	○	○	○	●	○
UA_transfer.f2p_12	11	g	N3 >= 2 && N2 >= 2 => N1*(N2-1)	●	●	○	○	○	●	○
UA_transfer.f2p_13	4	ł	N2 >= 3 => N1	●	●	●	◙	◙	○	○
UA_transfer.f2p_14	15	g	(N2 >= 3 => N1) + (N4 >= 3 && N3 >= 2 => N1*(N3-1))	●	●	○	○	○	●	○
UA_transfer.f2p_15	4	ł	N2 >= 3 => N1	●	●	●	◙	◙	○	○
UA_transfer.f2p_16	8	g	N3 >= 3 && N2 >= 1 => (N1-1)*N2	●	●	○	○	○	●	○
UA_transfer.f2p_17	15	g	(N2 >= 3 => N1) + (N4 >= 3 && N3 >= 2 => N1*(N3-1))	●	●	○	○	○	●	○
UA_transfer.f2p_18	4	ł	N2 >= 3 => N1	●	●	●	◙	◙	○	○
UA_transfer.f2p_19	8	g	N3 >= 3 && N2 >= 1 => (N1-1)*N2	●	●	○	○	○	●	○
UA_utils.f2p_12	39 / 6	g	N1 (*)	◘	◘	○	○	○	○	●
UTDSP Benchmark										
compress_1.t	20 / 2	d	B*B	●	●	○	●	◙	●	●
compress_2.t	29 / 5	g	B*B	●	●	○	○	○	●	○
compress_3.t	29 / 5	g	B*B	●	●	○	○	○	●	○
edge_detect1.t	21 / 3	g	N*N	●	●	○	○	●	●	○
edge_detect2.t	2	g	K	◘	◘	○	○	○	○	●
edge_detect3.t	7	g	(N-K)*(N-K)	●	●	○	○	○	●	○
fir_256_64.t	51 / 11	g	NPOINTS	●	●	○	○	○	●	○
g721_w_1.at	1	ł	N-1	●	●	●	◙	◙	○	○
histogram_3.t	13	g	N	●	●	○	○	○	●	○
jpeg_1.t	3	ł	4*N	●	●	●	◙	◙	○	○
lpc_1.t	19 / 6	g	P	●	●	○	○	○	●	○
lpc_10.t	1	ł	N-1	●	●	●	◙	◙	○	○
lpc_4.t	4	g	N-2	●	●	○	○	●	●	○
lpc_5.t	5	g	D-5	●	●	○	○	○	●	○
lpc_6.t	3	g	D-5	●	●	○	○	○	●	○
lpc_9.t	5	g	D	●	●	○	○	○	●	○
mult_10_10.t	24 / 6	g	A_ROW*B_ROW	●	●	○	○	○	●	○
mult_4_4.t	24 / 6	g	A_ROW*B_ROW	●	●	○	○	○	●	○

Legenda:

ł, d, g – w kolumnie trzeciej Topologia oznaczają odpowiednio łańcuch, drzewo, graf,

● – algorytm zastosowany z powodzeniem

○ – zastosowanie algorytmu jest niemożliwe

◙ - algorytm możliwy do zastosowania, jednak algorytm 3.3, lub 3.4, dostarcza kod równoległy o lepszej wydajności,

◘ - algorytm zastosowany do ekstrakcji równoległości, po wcześniejszym zastosowaniu algorytmu 3.9,

Θ – zastosowanie algorytmu 3.1 znajdowania początków z przyczyn ograniczeń narzędzia Omega Calculator jest niemożliwe, w celu uzyskania kodu równoległego zastosowano zbiór początków krańcowych Domain(R) – Range(R),

(*) - liczba fragmentów kodu wymagających synchronizacji, stosując sam algorytm znajdowania początków uzyskano tylko jeden początek

(**) - tylko podział na grafy SCC

W tabeli 4.4 podano statystykę topologii pętli, dla których uzyskano kod równoległy. Najczęściej występującą topologią jest graf, następnie łańcuch. W obydwu testach najrzadziej występującą topologią okazało się drzewo.

W tabeli 4.5 podsumowano dla ilu pętli wykorzystano poszczególne algorytmy.

Tab. 4.4 Topologie badanych pętli

Benchmark	**Ilość bad. pętli**	**Topologia**					
		Łańcuch		**Drzewo**		**Graf**	
NAS	90	33	36,7%	4	4,4%	53	58,9%
UTDSP	18	3	16,6%	1	5,6%	14	77,8%

Tab. 4.5 Zakres zastosowania prezentowanych algorytmów

Benchmark	**Ilość bad. pętli**	**Algorytmy**						
		3.1	**3.2**	**3.3**	**3.4**	**3.5**	**3.8**	**3.9**
NAS	90	73	67	31	32	35	31	3
UTDSP	18	17	17	3	4	6	14	2

4.3. Transformacje kodu wynikowego

Kod wynikowy został wygenerowany w języku C w sposób automatyczny za pomocą autorskiego narzędzia [75] w następujących etapach:

- ekstrakcja zależności pętli za pomocą narzędzia Petit i próba ich zredukowania [77],

- obliczenie zbioru i wyznaczenie pętli początków za pomocą narzędzia *codegen* przy pomocy autorskich funkcji napisanych dla biblioteki Omega [76],
- uzyskanie dodatkowych danych (ograniczenia relacji, zbiorów) niezbędnych do wygenerowania ostatecznego kodu,
- przekazanie wyników do autorskiego narzędzia do generowania kodu w sposób automatyczny w postaci tekstowej za pomocą technologii COM do bibliotek opracowanych w języku C#,
- generowanie kodu ostatecznego: (wstawianie pętli while, instrukcji synchronizacji, itp.) w środowisku .NET z wykorzystaniem pakietu Wolfram Mathematica 5.2 [94] za pomocą narzędzia Wolfram .NET Link.

W skład projektu wchodzi implementacja proponowanych algorytmów, rozwiązania wspomagające skuteczność narzędzia Omega Calculator oraz zaawansowany i w pełni zautomatyzowany generator kodu, w tym dla pętli *while* i kodu z synchronizacją. Opracowane narzędzie wraz z kodem źródłowym i dokumentacją jest dostępne pod adresem internetowym [75] oraz w załączniku B w postaci elektronicznej. Stanowi ono **znaczący element prac badawczych i potwierdza przydatność oraz poprawność proponowanych rozwiązań w postaci praktycznej implementacji.**

Otrzymane pętle następnie przekonwertowano ręcznie na programy równoległe w standardzie OpenMP [73] w celu sprawdzenia jakości otrzymanego kodu równoległego. Dla zwiększenia lokalności kodu wynikowego zastosowane zostały transformacje: aglomeracja (ang. *agglomeration*) i skalaryzacja tablic (ang. *scalar replacement*).

Aglomeracja – umożliwia zwiększanie ziarna kodu przez łączenie zadań. Redukcja liczby zadań prowadzi do wzrostu wielkości pojedynczego zadania [58]. Dzięki aglomeracji zredukowano koszty zarządzania wątkami oraz zwiększono lokalność kodu poprzez redukcję chybień do pamięci podręcznej procesora [52]. W celu uzyskania optymalnej aglomeracji podczas badań zauważono, że istotny jest dobór zadań tworzący jedno większe ziarno. Zadania powinny odwoływać sie do obszarów pamięci przylegających do siebie lub możliwie bliskich sobie, o ile jest to możliwe.

Skalaryzacja – konwersja wybranych odwołań do zmiennych tablicowych na odwołania do zmiennych skalarnych. Każdorazowe odwołanie do tablicy skutkuje obliczeniem adresu komórki pamięci. W przypadku, gdy w pętli programowej odwołujemy sie to tej samej komórki pamięci wielokrotne obliczenie adresu wprowadza niepotrzebne opóźnienie. Ponieważ kompilatory potrafią efektywnie przypisać zmienne skalarne do rejestrów procesora, odwołania do tej samej zmiennej tablicowej można

przekształcić na odwołania do jednej zmiennej skalarnej. Dzięki temu można uzyskać wielokrotne odwołanie do rejestru .

4.4. Badania wydajności kodu równoległego wybranych pętli

W niniejszym rozdziale przedstawiono szczegółową analizę sześciu pętli (cztery z zestawu NAS oraz dwie z UTDSP) zrównoleglonych za pomocą pragm OpenMP [31], [73]. Głównymi kryteriami wyboru pętli były ilość wyznaczonych niezależnych fragmentów kodu, ilość obliczeń w ciele pętli oraz topologia zależności.

Zrównoleglenie pętli, których ciało zawiera niewiele obliczeń, jest zwykle nieopłacalne ze względu na koszty czasowe potrzebne do utworzenia i synchronizacji wątków. Korzyści czasowe ze zrównoleglenia wykonywania ciała pętli muszą przewyższać straty czasowe poświęcone na zarządzania wielowątkowością.

Poniżej zaprezentowano badania wydajności kodu równoległego pętli. Zawarto kod pętli badanej w języku analizatora zależności narzędzia Petit oraz analizę grafów SCCs. Ze względu na ograniczony rozmiar pracy w rozdziale umieszczono tylko wybrane kody równoległe badanych pętli, natomiast komplet wszystkich wygenerowanych pętli umieszczono w załączniku B. Pętle możliwe do zrównoleglenia oznaczono słowem kluczowym par.

Badania przeprowadzono na środowisku:

- **Workstation Board S5000XVN Intel Xeon Quad Core,** 1.6 Ghz, 8 procesorowy (2 czterordzeniowe jednostki przetwarzające, cache 4 MB), 2 GB RAM, Ubuntu Linux,
- kompilator GNU C++ 4.2

Czas obliczeń wyznaczono za pomocą funkcji *gettimeofday()* i jest różnicą między czasem końca obliczeń równoległych a czasem rozpoczęcia obliczeń pętli.

Poprawność wyników sprawdzono w każdym badaniu w następujący sposób: pętla sekwencyjna i równoległa otrzymały ten sam zbiór danych wejściowych. Wyniki pętli sekwencyjnej tj. wszystkie zmienne (skalarne i tablicowe) modyfikowane w pętli, zostały porównane z wynikami pętli równoległej w celu sprawdzenia ich zgodności.

Kod wyjściowy pętli równoległej stworzony został w oparciu o pragmy OpenMP. Do przeprowadzenia testów wybrano harmonogramowanie statyczne [73] (ang. *static schedule*). W przypadku przeprowadzanych badań moc obliczeniowa jednostek przetwarzających była w 100% przydzielona do określonego zadania. Harmonogramowanie dynamiczne [73] (*dynamic* lub *guided*) uzasadnione byłoby w

przypadku, gdy: ilość obliczeń w poszczególnych iteracjach jest różna, system jest obciążony innymi zadaniami lub ilość wątków jest większa od liczby procesorów.

W związku z powyższym badania ujęte w dalszej części rozdziału przeprowadzone zostały zgodnie z harmonogramowaniem statycznym.

W kolejnych podrozdziałach przedstawiono wyniki badań poddanych testom sześciu wybranych pętli z testów NAS i UTDSP. W tabeli 4.6 zawarto badane pętle oraz grupę pętli o podobnej strukturze i spodziewanie przybliżonej wydajności kodu równoległego. Pełen wykaz grup pętli zawarto w załączniku B. Do oznaczenia podobnej grupy algorytmów wyznaczono 5 charakterystyk:

- topologia – wyznacza ona zakres stosowalności algorytmów zrównoleglenia, stopień skomplikowania kodu, dodatkowe narzuty zasobów pamięciowych i komunikacji,
- obliczenia numeryczne – ich większa ilość w ciele pętli wiąże się z potencjalnie lepszym przyspieszeniem obliczeń równoległych. Wynika z prawa Amdahla [4]. Czas zarządzania wątkami jest z reguły stały, natomiast porcja obliczeń numerycznych jest wykonywana równolegle. Zwiększenie czasu obliczeń ciała pętli niweluje w większym stopniu koszty zrównoleglenia,
- zagnieżdżenie pętli – pętla idealnie zagnieżdżona charakteryzuje się często lepszą lokalnością kodu i łatwiejszym procesem zrównoleglenia,
- jednorodność relacji zależności – wyznacza zakres stosowalności algorytmów zrównoleglenia, obecność tylko jednorodnych relacji zależności pozwala uzyskać zwykle wydajniejszy kod równoległy,
- obecność niezależnych fragmentów kodu – ich brak wymusza wprowadzenie dodatkowych instrukcji synchronizujących, pomniejszających wydajność obliczeń równoległych

Wartości przyspieszenia i efektywności oraz zestawienie wyników czasowych zawarto w tabeli 4.6. Badania przeprowadzono w celu sprawdzenia jakości uzyskanego kodu równoległego dla pętli opisanych w tabeli 4.3.

Tab. 4.6 Grupy pętli podobnych do wybranego zestawu badań

Lp.	Pętla	Topologia / zredukowana topologia	Obliczenia num. typu mnożenie, dzielenie, potęgowanie i inne funkcje mat.	Idealnie zagnieżdżona	Wszystkie relacje jednorodne	Liczba niezależnych fragmentów > 1	Liczba pętli o podobnej strukturze	
							NAS	UTDSP
1	FT_appft.f2p_1	graf / łańcuch	tak	nie	nie	tak	28 (31%)	10 (53%)
2	UA_utils.f2p_12	graf	tak	nie	nie	nie	3 (4%)	1 (7%)
3	Edge_detect_1	graf / łańcuch	nie	tak	tak	tak	8 (9%)	3 (16%)
4	LU_HP_pintgr.f2p_2 Compress_1	dowolna	tak	tak	tak	tak	17 (19%)	3 (16%)
5	BT_rhsf2p_5	graf / łańcuch	tak	tak	nie	tak	11 (12%)	1 (7%)

4.4.1. Pętla FT_appft.f2p_1 (NAS)

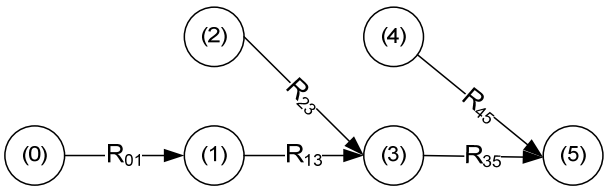
Dane wejściowe pętli z zestawu NAS do algorytmów stanowi oryginalny kod źródłowy zapisany w języku akceptowanym przez program Petit, tab. 4.7.

Przebieg analizy

Zależności badanej pętli posiadają wspólne końce i tworzą topologię grafu. W celu zredukowania liczby zależności niektóre zmienne pętli zostały zamienione na zmienne tablicowe. Pozwoliło to na zredukowanie liczby relacji z 47 do 5.

Zależności pętli:

1. {[i,-1,-1,0] -> [i,-1,-1,1] : 1 <= i <= N1 && 1 <= N3 && 1 <= N2}
2. {[i,-1,-1,1] -> [i,k',-1,3] : 1 <= i <= N1 && 1 <= k' <= N2 && 1 <= N3}
3. {[i,k,-1,2] -> [i,k,-1,3] : 1 <= i <= N1 && 1 <= k <= N2 && 1 <= N3}
4. {[i,k,-1,3] -> [i,k,j',5] : 1 <= i <= N1 && 1 <= k <= N2 && 1 <= j' <= N3}
5. {[i,k,j,4] -> [i,k,j,5] : 1 <= i <= N1 && 1 <= k <= N2 && 1 <= j <= N3}



Wszystkie zależności tworzą jeden graf SCCs. Dla instrukcji (0), która poprzedza pozostałe instrukcje, wyznaczono zbiór początków fragmentów kodu (ang. *slices*):

Sources := {[i,-1,-1,0]: 1 <= i <= N1 && 1 <= N3 && 1 <= N2}

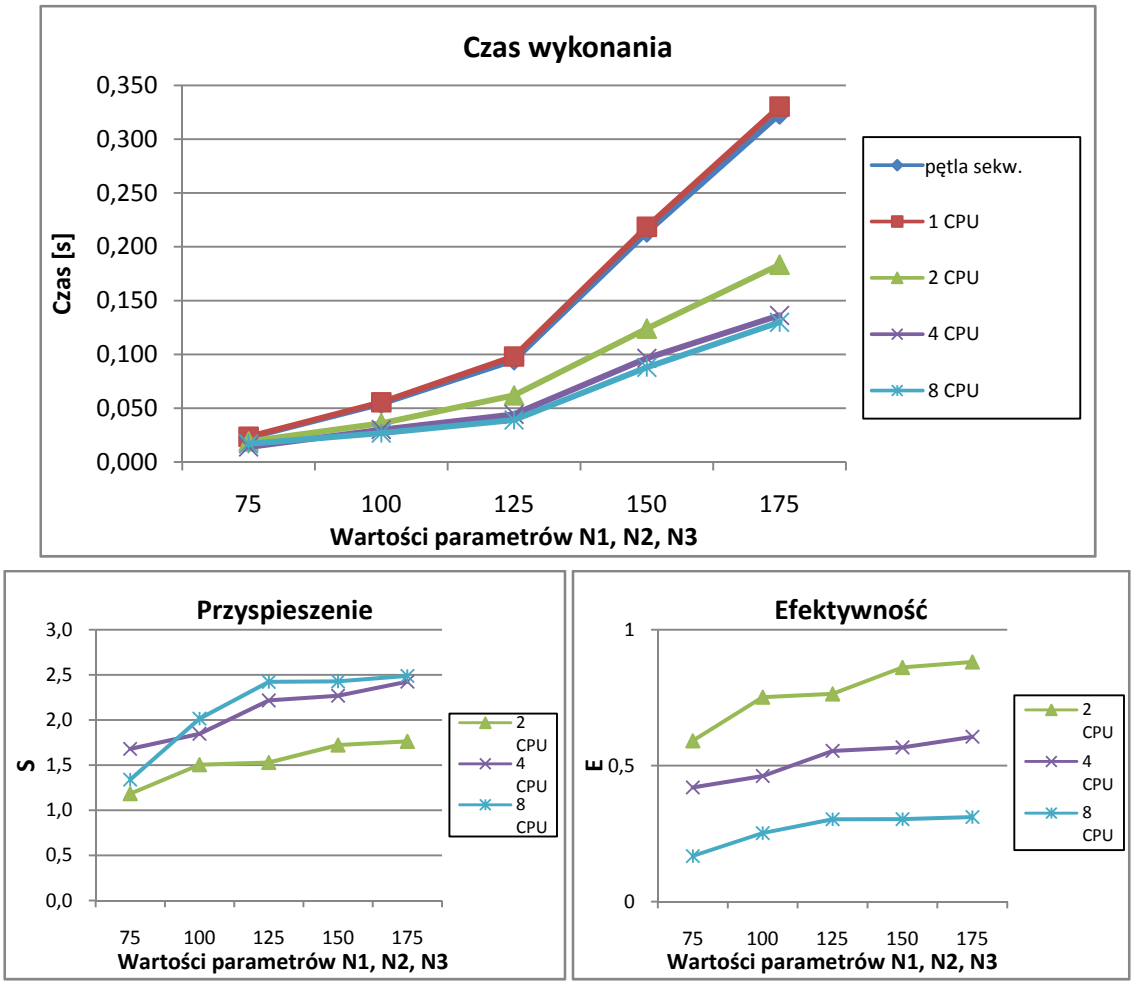
Łącznie otrzymano N1 fragmentów nie wymagających synchronizacji między sobą. Do wyznaczenie ich iteracji wykorzystano algorytmy 3.2 i 3.8. Algorytm 3.5 nie może być wykorzystany ponieważ relacje posiadają niejednorodny wektor zależności i niejednoznaczne jest określenie następnego kroku w pętli *while*.

Wyniki badań

Dla wygenerowanej pętli równoległej zbadano wartości przyspieszenia i efektywności na 2, 4 i 8 procesorach oraz dla różnej liczby iteracji pętli. Badania wykonano dla 5 różnych wartości parametrów górnych granic pętli. Do harmonogramowania iteracji pętli wybrano wariant statyczny (ang. *static*) z domyślną wartością paczki.

Tab. 4.7 Postać sekwencyjna pętli FT_appft.f2p_1

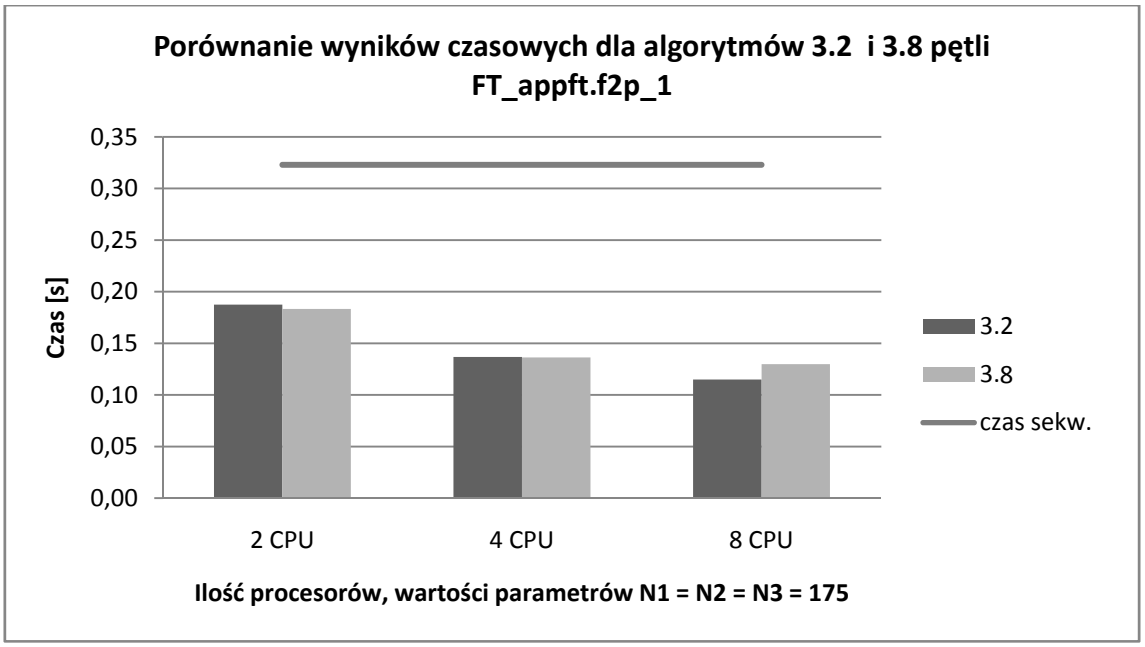
```
if(N1 >= 1 and N2 >= 1 and N3 >=1) then
for i=1 to N1 do
  ii(i) = i-1-((i-1)/n32)*nz
  ii2(i) = ii(i)*ii(i)
  for k=1 to N2 do
    kk(i,k) = k-1-((k-1)/n22)*ny
    ik2(i,k) = ii2(i) + kk(i,k)*kk(i,k)
    for j=1 to N3 do
      jj(i,k,j) = j-1-((j-1)/n12)*nx
      twiddle(j,k,i) = ap*dble(jj(i,k,j)*jj(i,k,j) + ik2(i,k))
    endfor
  endfor
endfor
```



Rys. 4.2. Czas wykonania, przyspieszenie i efektywność pętli FT_appft.f2p_1, algorytm 3.8 [op. własne]

Zwiększanie liczby iteracji koreluje wraz ze wzrostem przyspieszenia obliczeń. Dla przykładu efektywność obliczeń dla N1 = N2 = N3 =75 wyniosła 0,59, podczas gdy dla N1=N2=N3=175 osiągnęła 0,88 dla 2 procesorów dla algorytmu 3.8. Oznacza to, że większa liczba iteracji i obliczeń w ciele pętli niweluje koszty zarządzania wątkami i dodatkowych instrukcji pętli *while*. Na rysunku 4.2 przedstawiono wyniki czasowe wykonania pętli otrzymanej za pomocą algorytmu 3.8.

Rysunek 4.3 przedstawia porównanie czasu wykonania pętli równoległych otrzymanych za pomocą algorytmu 3.2 i 3.8. Wyniki czasowe są zbliżone. Uproszczenie topologii do łańcucha pozwoliło na uzyskanie pętli równoległej opartej o konstrukcję *while* o przyspieszeniu nie wiele gorszym od pętli wygenerowanej generatorem kodu dla zbiorów afinicznych.



Rys. 4.3. Porównanie zrównoleglenia pętli FT_appft.f2p_1 algorytmami 3.2 i 3.8 [op. własne]

4.4.2. Pętla UA_utils.f2p_12 (NAS)

Dane wejściowe pętli z zestawu NAS do algorytmów stanowi oryginalny kod źródłowy zapisany w języku akceptowanym przez program Petit, tab. 4.8.

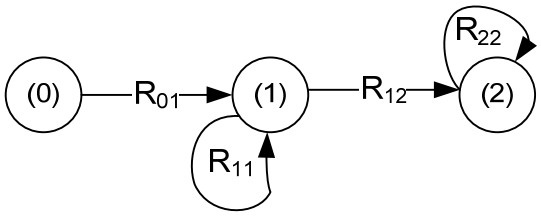
Przebieg analizy

Pętla z zestawu NAS, której zależności posiadają wspólne końce, tworząc topologię grafu. Zmienna *ieltotal* może zostać przekształcona na zmienną tablicową w celu wyeliminowania zależności.

W pętli wyznaczono następujące relacje zależności:

1. {[iel,-1,-1,-1,0] -> [iel,i,j,k,1] : 1 <= iel <= N1 && 1 <= i <= N4 && 1 <= k <= N2 && 1 <= j <= N3}
2. {[iel,k,j,i,1] -> [iel,k,j,i',1] : 1 <= i < i' <= N4 && 1 <= iel <= N1 && 1 <= k <= N2 && 1 <= j <= N3}
3. {[iel,k,j,i,1] -> [iel,k,j',i',1] : 1 <= j < j' <= N3 && 1 <= iel <= N1 && 1 <= k <= N2 && 1 <=i <= N4 && 1 <= i' <= N4}
4. {[iel,k,j,i,1] -> [iel,k',j',i',1] : 1 <= k < k' <= N2 && 1 <= iel <= N1 && 1 <= j <= N3 && 1 <= i <= N4 && 1 <= i' <= N4 && 1 <= j' <= N3}
5. {[iel,k,j,i,1] -> [iel,-1,-1,-1,2] : 1 <= iel <= N1 && 1 <= k <= N2 && 1 <= j <= N3 && 1 <= i <= N4}
6. {[iel,-1,-1,-1,2] -> [iel+1,-1,-1,-1,2] }

Dla badanej pętli wyznaczono graf SCC. Instrukcja (2) musi zostać wykonana przed (1), z kolei ta przed (0) honorując zależności R_{01} i R_{12}. Wyznaczony graf jest grafem cyklicznym, czyli początek i koniec zależności znajduje się w różnych instancjach tej samej instrukcji, (1) i (2).



Algorytm do znajdowania początków 3.1 uniemożliwia zrównoleglenia pętli ponieważ relacje tworzą tylko jeden fragment o początku [1,-1,-1,-1,0]. W tym przypadku możliwe jest zastosowanie tylko algorytmu do wyznaczania równoległości z synchronizacją – algorytm 3.9.

Do zbioru relacji tworzących wątki S1 przyporządkowano relacje 1 – 5. Szósta relacja zaliczona została do zbioru S2 niezbędny do wskazania punktów synchronizacji. Wówczas można wyznaczyć zbiór początków na podstawie relacji z S1:

```
Sources := {[i,-1,-1,-1,0]: 1 <= i <= N1 && 1 <= N3 && 1 <= N2}
```

Fragmenty i ich iteracje zostały wygenerowane za pomocą algorytmu 3.2. Kod pętli wzbogacono o instrukcje synchronizacji, aby zachować zgodność z ostatnią zależnością. Warto zauważyć, że instrukcje *send* i *recv* zawierają tylko 1 argument, ponieważ wymiar tablicy zamków jest tylko jeden. Pozostałe argumenty równe są -1 oraz dotyczą tej samej instrukcji, stąd nie ma potrzeby ich dodawania. Dzięki temu mechanizm synchronizacji w pętli równoległej posiada mniejsze zapotrzebowanie na pamięć.

Niezależne bloki wykonywane są natychmiast, natomiast synchronizacja dokonywana jest po wykonaniu obliczeń fragmentów kodu. Dlatego w badaniach wybrano rozmiar paczki *chunksize=1*, aby wszystkie bloki zostały uruchomione od razu. W zrównolegleniu pętli z synchronizacją nie wykorzystano aglomeracji ponieważ w tym przypadku prowadziłaby ona tylko do usekwencyjnienia kodu. W tabeli 4.8 zamieszczono kod wejściowy i równoległy pętli.

Tab. 4.8 Postać sekwencyjna i równoległa pętli UA_utils.f2p_12

```
for iel=1 to N1 do
  ieltotal=0.
  isize=size_e(iel)
  for k=1 to N2 do
    for j=1 to N3 do
      for i=1 to N4 do
        ieltotal=ieltotal+ta1(i,j,k,iel)*w3m1(i,j,k)  *jacm1_s(i,j,k,isize)
      endfor
    endfor
  endfor
  total=total+ieltotal
endfor
```

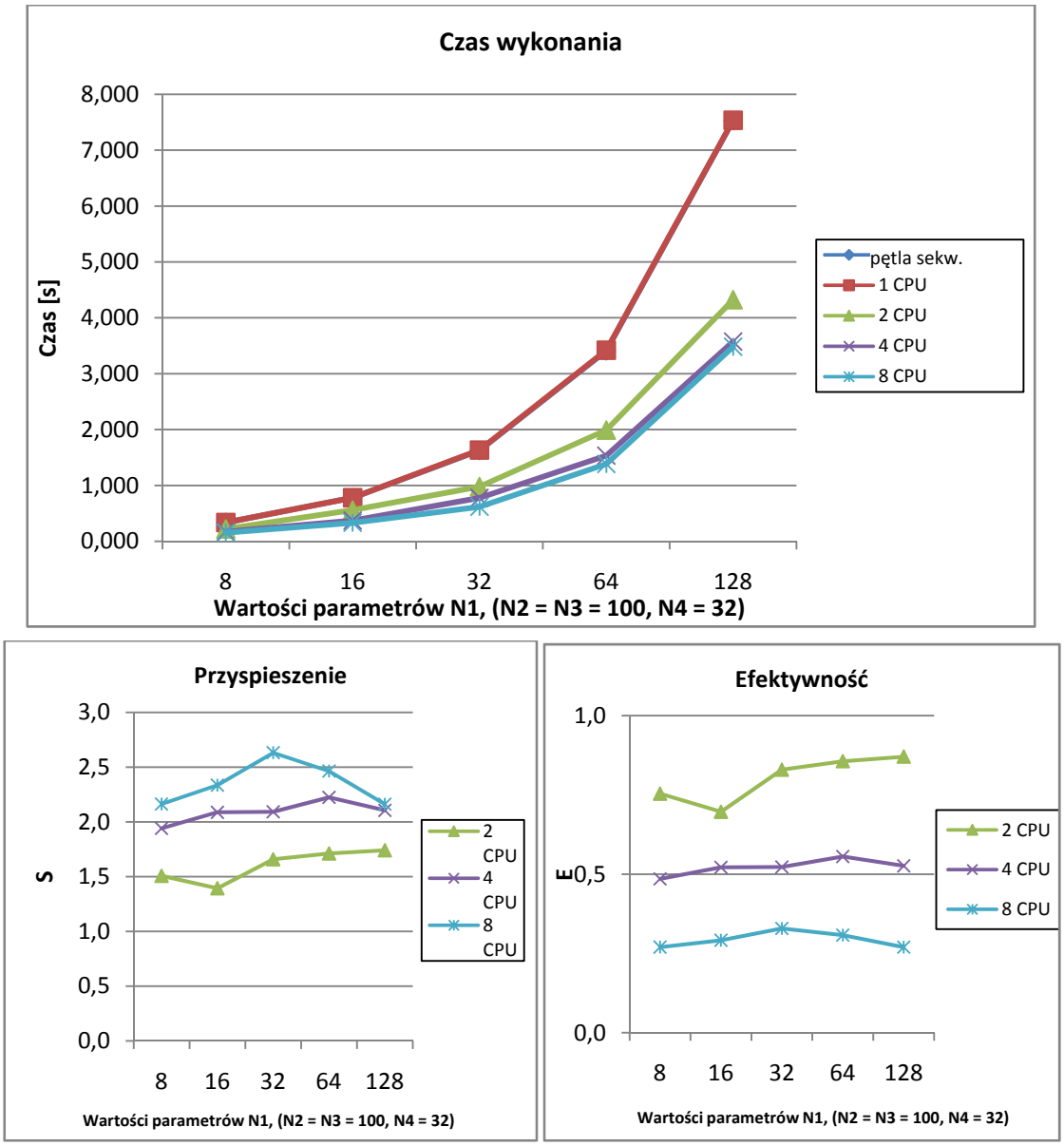
(a) Pętla poddana analizie

```
if (N2 >= 1) {
  par for(t1 = 1; t1 <= N1; t1++) {   //par
     ieltotal(t1)=0;
     for(t2 = 1; t2 <= N2; t2++) {
       for(t3 = 1; t3 <= N3; t3++) {
         for(t4 = 1; t4 <= N4; t4++) {
          ieltotal(t1)=ieltotal(t1)+ta1(t4,t3,t2,iel)*w3m1(t4,t3,t2)  *jacm1_s(t4,t3,t2,isize(t1))
         }
       }
     }
   if(t1 > 1)
     recv(t1-1);
   total=total+ieltotal(t1);
   if(t1 < N1)
    send(t1);
  }
}
```

(b) Pętla w postaci równoległej

Wyniki badań

Badania pętli przeprowadzono na 2, 4 i 8 procesorach oraz różnej liczbie parametrów pętli wyznaczających jej górne granice oraz liczbę iteracji. Przy małej liczbie iteracji koszty zarządzania wątkami i synchronizacji są duże w porównaniu do obliczeń. Z kolei dla bardzo dużej liczby iteracji zwiększyła się liczba punktów synchronizacji. Z tych powodów efektywność obliczeń nie zawsze rośnie wraz z liczbą iteracji, co ma miejsce w innych badanych pętlach.



Rys. 4.4. Czas wykonania, przyspieszenie i efektywność pętli UA_utils.f2p_12 [op. własne]

Dla dwóch procesorów efektywność rośnie wraz z liczbą iteracji i najlepszą uzyskano dla N1=N2=N3=128. Z kolei dla 4 procesorów najlepszą efektywność uzyskano gdy parametry były równe 64, dla 8 procesorów 32. Przy większej liczbie iteracji i wątków koszty synchronizacji wzrastają. Rysunek 4.4 ukazuje, że uzyskany czas pętli równoległej jest nadal zdecydowanie krótszy od sekwencyjnego odpowiednika pomimo kosztów synchronizacji wątków.

4.4.3. Pętla Edge_detect_1 (UTDSP)

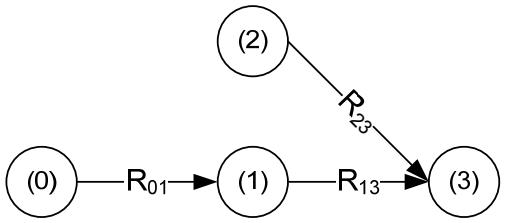
Badana pętla pochodzi z aplikacji *Edge_detect* z zestawu testów UTDSP. Dane wejściowe do algorytmów stanowi oryginalny kod źródłowy zapisany w języku akceptowanym przez program Petit, tab. 4.9.

Przebieg analizy

W pętli wykorzystano algorytm do znajdowania początków 3.1 oraz iteracje zostały wyznaczone w celu analizy porównawczej trzema algorytmami 3.2, 3.5 i 3.8. Pętla zawiera następujące trzy zależności:

1. {[i,j,0] -> [i,j,2] : 0 <= i < N && 0 <= j < N}
2. {[i,j,1] -> [i,j,2] : 0 <= i < N && 0 <= j < N}
3. {[i,j,2] -> [i,j,3] : 0 <= i < N && 0 <= j < N}

Dla badanej pętli wyznaczono następujący graf SCC



Topologia zależności jest grafem ogólnym. Znaleziono następujący zbiór początków:

Sources := {[i,j,0]: 0 <= i < N && 0 <= j < N}

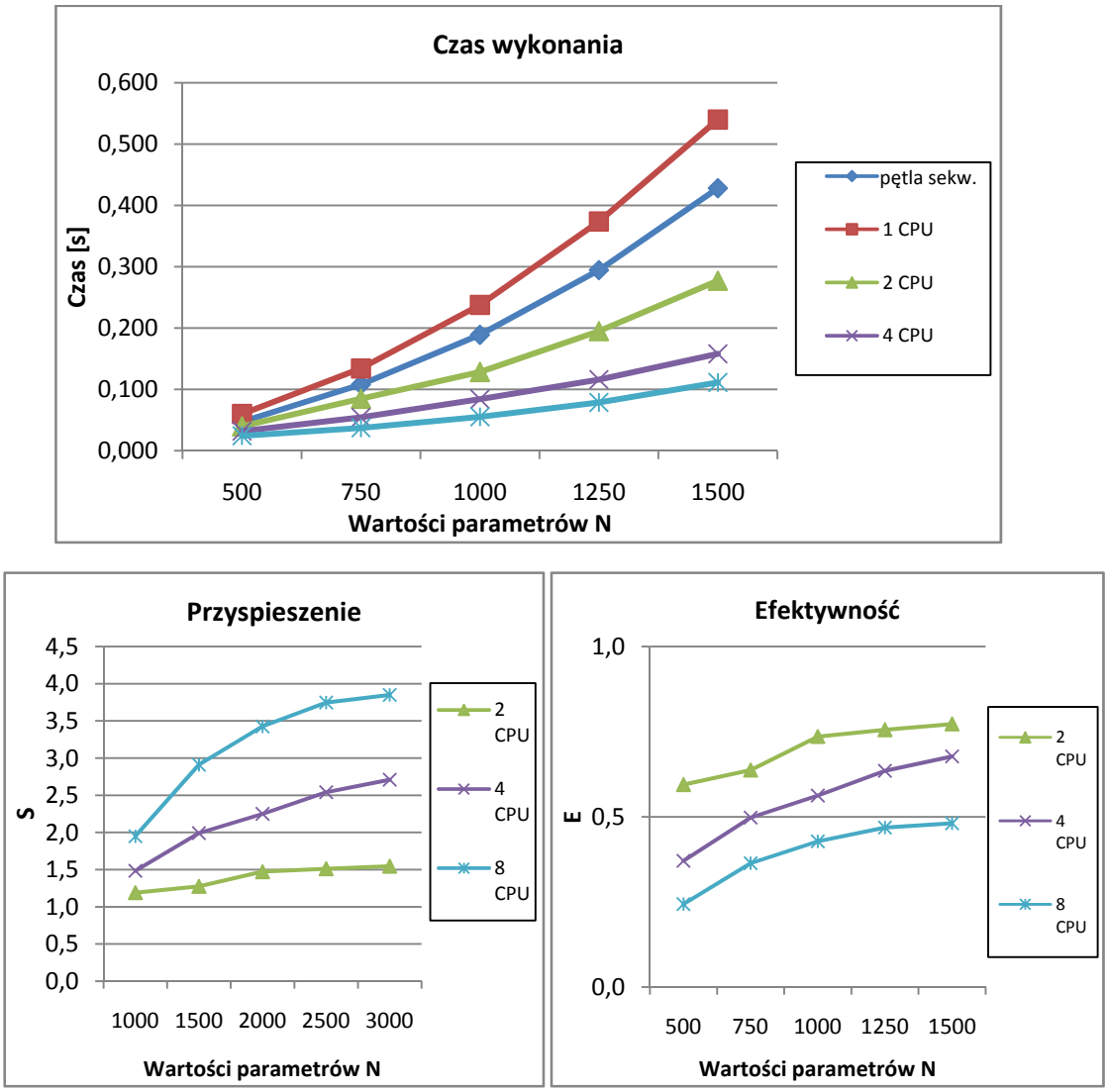
Pętla równoległa składa się z N niezależnych fragmentów kodu.

Tab. 4.9. Postać sekwencyjna pętli Edge_detect_1

```
for i = 0 to N-1 do
  for j = 0 to N-1 do
    temp1(i,j) = abs(image_buffer1(i,j));
    temp2(i,j) = abs(image_buffer2(i,j));
    temp3(i,j) = (temp1[t1][t2] > temp2[t1][t2]) ? temp1[t1][t2] : temp2[t1][t2];
    image_buffer3(i,j) = = (temp3[t1][t2] > T) ? 255 : 0;
  endfor
endfor
```

Wyniki badań

Postać równoległa pętli została zbadana z wykorzystaniem 2, 4 i 8 procesorów oraz dla różnych wartości parametru górnych granic pętli.

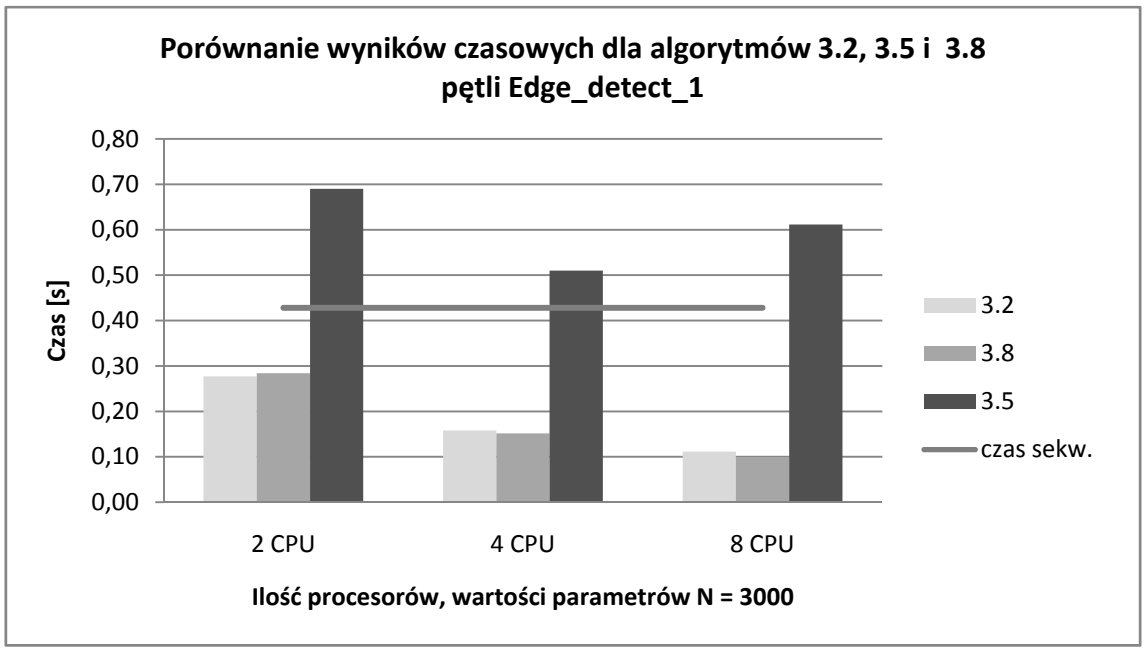


Rys. 4.5 Czas wykonania, przyspieszenie i efektywność pętli Edge_detect_1 dla algorytmu 3.2 [op. własne]

Najlepsze wyniki czasowe uzyskano przy użyciu największej liczby procesorów dla algorytmów 3.2 oraz 3.8. W przypadku algorytmu 3.5 uzyskano gorsze wyniki spowodowane nakładem obliczeniowym i dodatkowymi odwołaniami do pamięci przez wątki (rys. 4.5).

Efektywność jest skorelowana w tym przypadku wraz z liczbą iteracji dla każdej liczby wątków. Najlepszą efektywność podobnie jak w innych badanych pętlach uzyskano dla 2 procesorów.

Wyniki czasowe kodu równoległego uzyskanego za pomocą algorytmów 3.2 i 3.8 są zbliżone i znacznie krótsze od czasu wykonania sekwencyjnego odpowiednika. Natomiast narzuty na obsługę pętli równoległej wygenerowanej z wykorzystaniem algorytmu 3.5 okazały się za duże i nie uzyskano przyspieszenia obliczeń (rys 4.6).



Rys. 4.6 Porównanie zrównoleglenia pętli Edge_detect_1 algorytmami 3.2, 3.3 i 3.8 [op. własne]

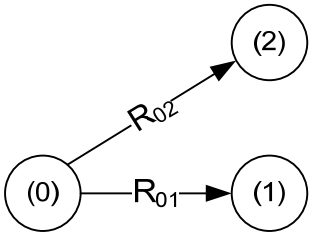
4.4.4. Pętla Compress_1 (UTDSP)

Badana pętla pochodzi z aplikacji *Compress* - kompresji obrazu z wykorzystaniem dyskretnej transformacji cosinusowej z zestawu UTDSP. Dane wejściowe pętli do algorytmów stanowi oryginalny kod źródłowy zapisany w języku akceptowanym przez program Petit, tab. 4.10.

Przebieg analizy

Pętla jest podwójnie i idealnie zagnieżdżona. Dzięki zamianie zmiennych w pętli na postać tablicową zredukowano liczbę zależności do dwóch:

1. {[m,n,0] -> [m,n,1] : 0 <= m <= B && 0 <= n <= B}
2. {[m,n,0] -> [m,n,2] : 0 <= m <= B && 0 <= n <= B}



Zależności opisywane przez relacje posiadają tylko wspólne początki oraz tworzą topologię drzewa. Topologia drzewa w badanych pętlach występuje znacznie rzadziej niż graf lub łańcuch. Do zrównoleglenia pętli wykorzystano algorytm znajdujący początki fragmentów 3.1 oraz algorytm przebierający iteracje za pomocą pętli *while* dla zależności o topologii drzewa 3.4. Dla porównania kod równoległy został wygenerowany także algorytmami 3.2 i 3.8. Zbiór znalezionych początków:

```
Sources := {[m,n,0]: 0 <= m <= B && 0 <= n <= B}
```

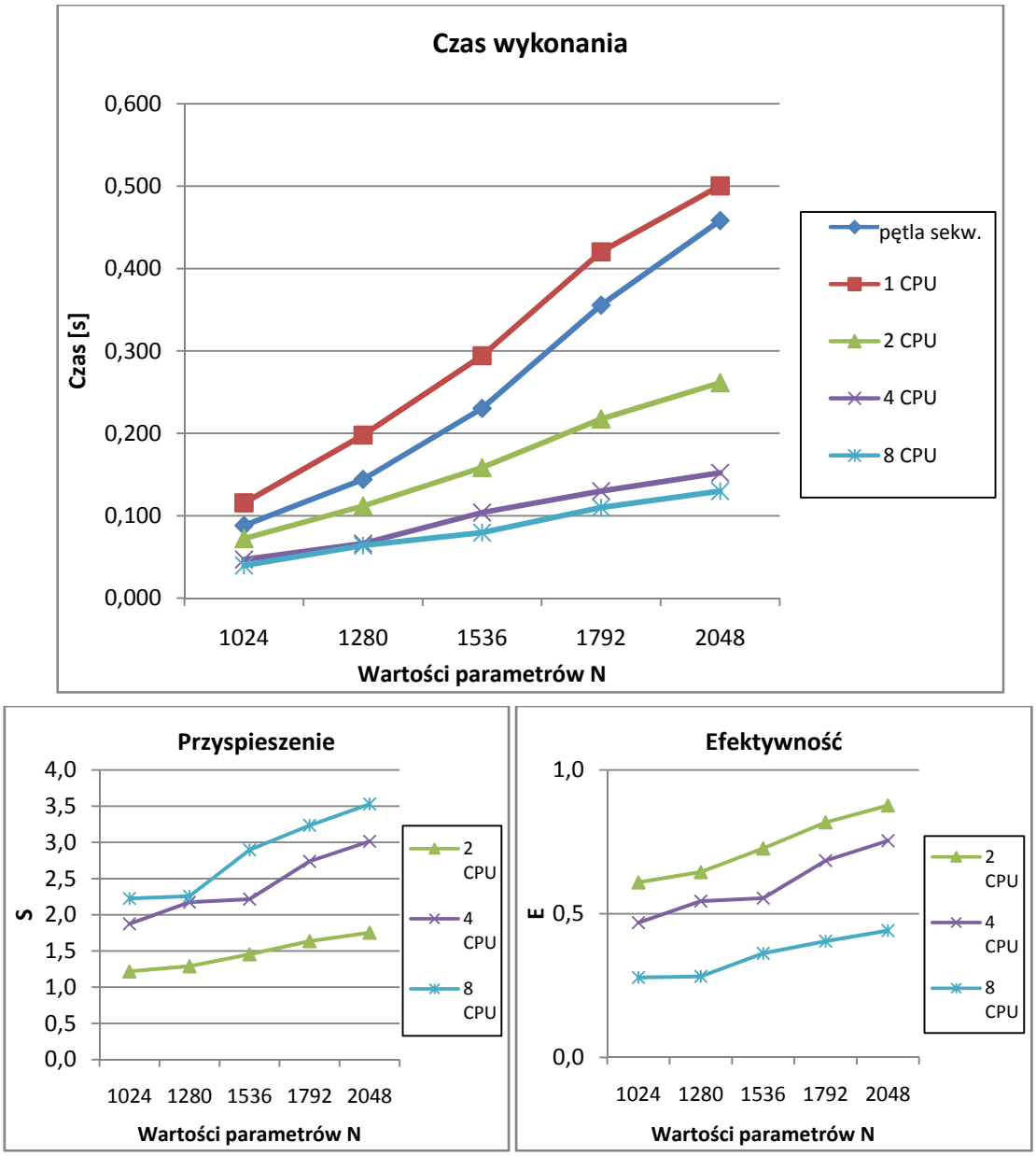
Poniżej przedstawiono postać analizowanej pętli.

Tab. 4.10 Postać sekwencyjna pętli Compress_1

```
for m=0 to B do
 for n=0 to B do
   temp_cos(m,n)=cos(m*factor1 * (2*n + 1)) / B;
   cos1(m,n)=temp_cos(m,n);
   cos2(n,m)=temp_cos(m,n);
  endfor
endfor
```

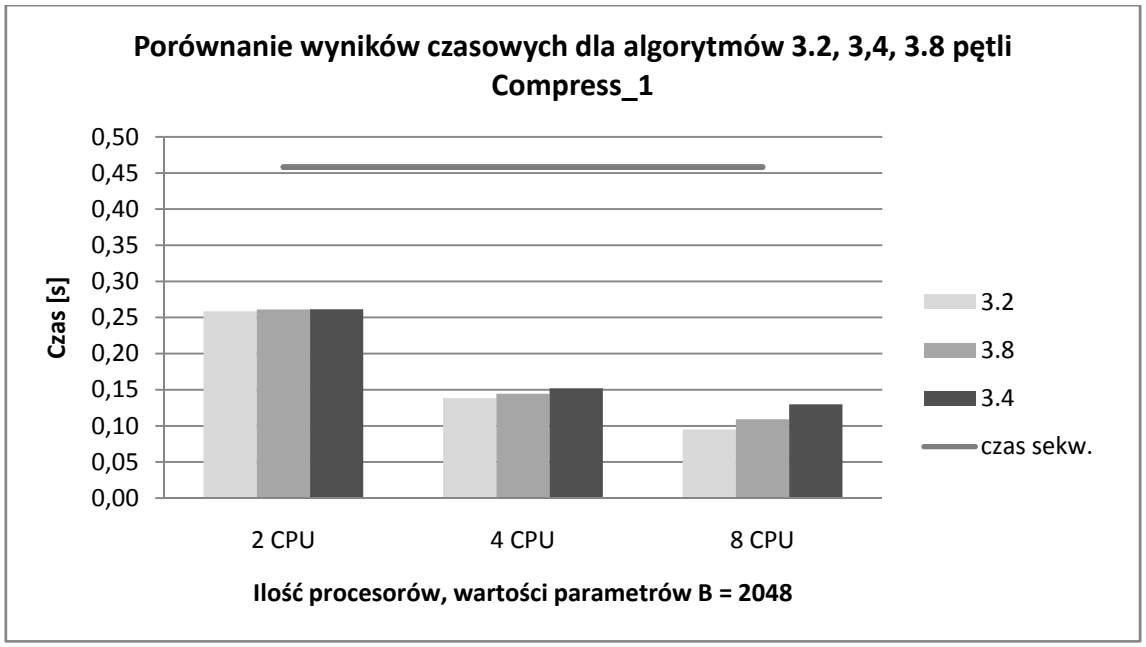
Wyniki badań

Na rysunku 4.7 przedstawiono wyniki czasowe badanej pętli zrównoleglonej algorytmem 3.4. Przyspieszenie rośnie wraz z liczbą iteracji. Dla małej liczby iteracji obliczenia z wykorzystaniem ośmiu procesorów jest mniej korzystne w porównaniu z czteroma. Koszty zarządzania większą liczbą wątków są niwelowane, gdy rośnie liczba iteracji oraz porcja obliczeń numerycznych w ciele pętli.



Rys. 4.7 Czas wykonania, przyspieszenie i efektywność pętli Compress_1 [op. własne]

W badaniach nad pętlą Compress_1 uzyskano zbliżone wyniki czasowe (rys. 4.8) dla kodu uzyskanego algorytmami 3.2, 3.4 i 3.8. Największe różnice czasowe odnotowano przy badaniu pętli równoległych z wykorzystaniem 8 procesorów. Najkrócej wykonana została pętla uzyskana generatorem kodu przebierającym elementy afinicznego zbioru iteracji. Drugi czas przypadł pętli przebierającej iteracje za pomocą konstrukcji *while* z zredukowaną topologią do łańcucha 3.8. Najdłużej wykonała się pętla uzyskana algorytmem 3.4 ze względu na dodatkowe zapotrzebowanie na pamięć i obliczenia.



Rys. 4.8 Porównanie zrównoleglenia pętli Compress_1 algorytmami 3.2, 3.4, i 3.8 [op. własne]

4.4.5. Pętla BT_rhsf2p_5 (NAS)

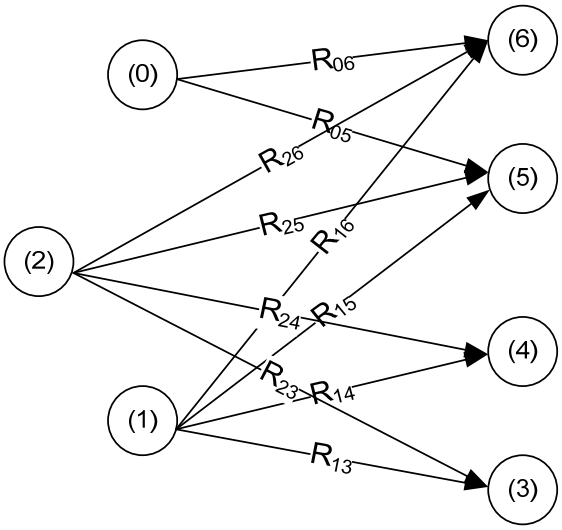
Badana pętla została wybrana z zestawu testów NAS i zbioru aplikacji *CFD*. Dane wejściowe pętli do algorytmów stanowi oryginalny kod źródłowy zapisany w języku akceptowanym przez program Petit, tab. 4.11.

Przebieg analizy

W ciele pętli znajduje się najwięcej odwołań do pamięci spośród wszystkich badanych ośmiu pętli w tym rozdziale. Pętla jest potrójnie i idealnie zagnieżdżona. Relacje zależności pętli poddanej analizie tworzą topologię grafu:

1. {[k,j,i,0] -> [k,j,i,v] : 1 <= k <= N1 && 1 <= j <= N2 && 1 <= i <= N3 && 11 <= N3 && 11 <=N2 && 11 <= N1 && 5<=v<=6}
2. {[k,j,i,1] -> [k,j,i,v] : 1 <= k <= N1 && 1 <= j <= N2 && 1 <= i <= N3 && 11 <= N3 && 11 <=N2 && 11 <= N1 && 3 <=v<=6}
3. {[k,j,i,2] -> [k,j,i,v] : 1 <= k <= N1 && 1 <= j <= N2 && 1 <= i <= N3 && 11 <= N3 && 11 <=N2 && 11 <= N1&& 3 <=v<=6}

Zależności opisane przez relacje posiadają niejednorodne wektory zależności, co uniemożliwia zastosowanie algorytmu 3.5. Możliwe jest nadal zastosowanie algorytmów do generowanie pętli 3.2 i algorytmu 3.8 z redukcją grafu do łańcucha.



Dla badanej pętli wyznaczono graf SCC. Instrukcje (3-6) muszą być poprzedzone wykonaniem instrukcji (1) i (2), natomiast instrukcja (0) poprzedza (5-6).

Wyznaczono zbiór początków:

Sources := {[k,j,i,0] : 1 <= k <= N1 && 1 <= j <= N2 && 1 <= i <= N3 }

W pętli wykorzystano algorytm do znajdowania początków 3.1, natomiast iteracje niezależnych fragmentów kodu zostały wyznaczono algorytmem 3.2.

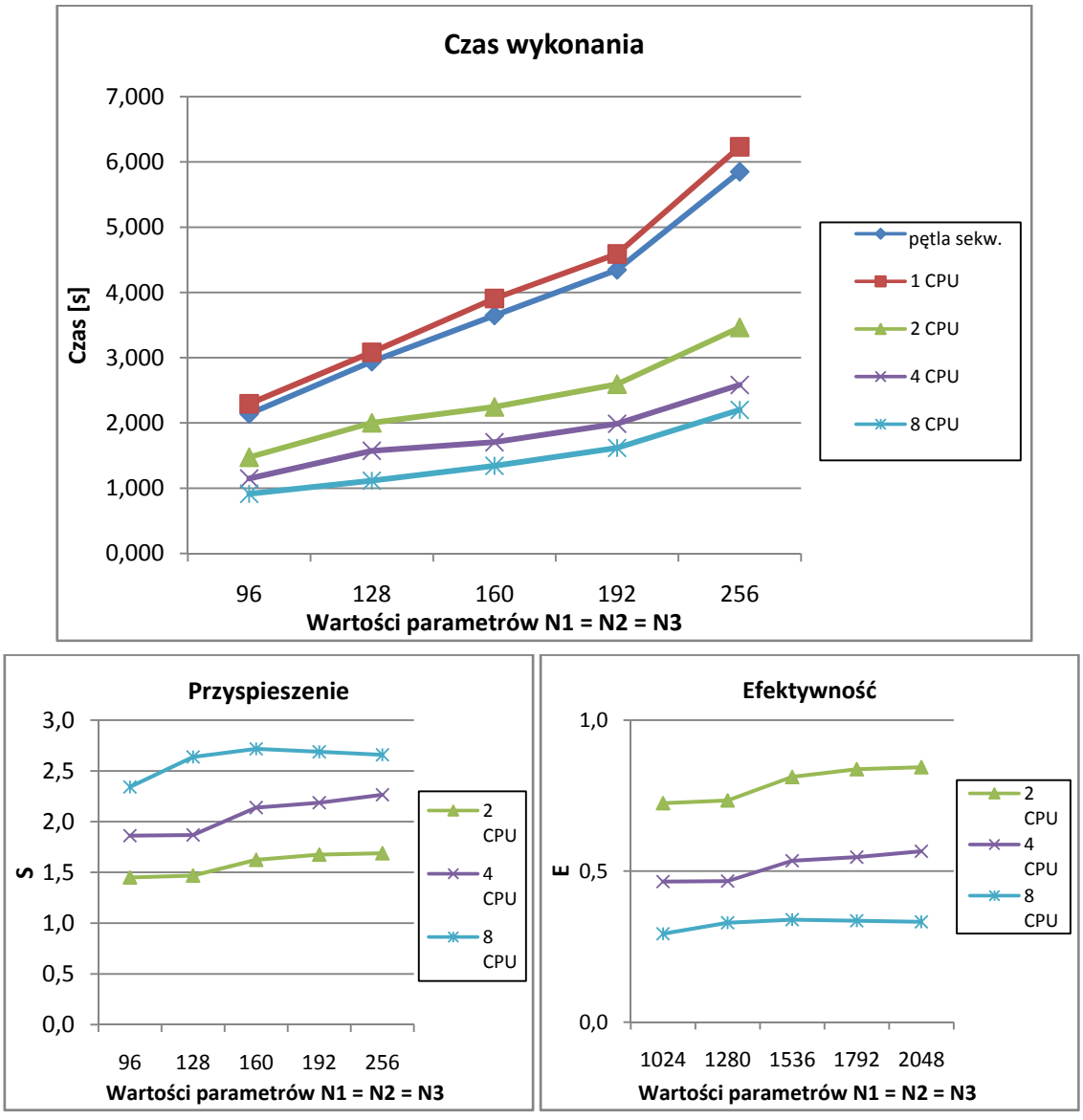
Tab. 4.11. Postać sekwencyjna pętli BT_rhsf2p_5

```
for k=1 to N1 do
 for j=1 to N2 do
  for i=1 to N3 do
    wijk(i,j,k) = ws(i,j,k)
    wp1(i,j,k)  = ws(i,j,k+1)
    wm1(i,j,k)  = ws(i,j,k-1)
    rhs(1,i,j,k) = rhs(1,i,j,k) + dz1tz1 * (u(1,i,j,k+1) - 2.0*u(1,i,j,k) + u(1,i,j,k-1)) - tz2 *
(u(4,i,j,k+1) - u(4,i,j,k-1))   //instrukcja niezależna od pozostałych
    rhs(2,i,j,k) = rhs(2,i,j,k) + dz2tz1 * (u(2,i,j,k+1) - 2.0*u(2,i,j,k) + u(2,i,j,k-1)) + zzcon2 *
(us(i,j,k+1) - 2.0*us(i,j,k) + us(i,j,k-1)) - tz2 * (u(2,i,j,k+1)*wp1(i,j,k) - u(2,i,j,k-1)*wm1(i,j,k))
    rhs(3,i,j,k) = rhs(3,i,j,k) + dz3tz1 * (u(3,i,j,k+1) - 2.0*u(3,i,j,k) + u(3,i,j,k-1)) + zzcon2 *
(vs(i,j,k+1) - 2.0*vs(i,j,k) + vs(i,j,k-1)) - tz2 * (u(3,i,j,k+1)*wp1(i,j,k) - u(3,i,j,k-1)*wm1(i,j,k))
    rhs(4,i,j,k) = rhs(4,i,j,k) + dz4tz1 * (u(4,i,j,k+1) - 2.0*u(4,i,j,k) + u(4,i,j,k-1)) + zzcon2*con43 *
(wp1(i,j,k) - 2.0*wijk(i,j,k) + wm1(i,j,k)) - tz2 * (u(4,i,j,k+1)*wp1(i,j,k) - u(4,i,j,k-1)*wm1(i,j,k) +
(u(5,i,j,k+1) - square(i,j,k+1) - u(5,i,j,k-1) + square(i,j,k-1)) *c2)
    rhs(5,i,j,k) = rhs(5,i,j,k) + dz5tz1 * (u(5,i,j,k+1) - 2.0*u(5,i,j,k) + u(5,i,j,k-1)) + zzcon3 *
(qs(i,j,k+1) - 2.0*qs(i,j,k) + qs(i,j,k-1)) + zzcon4 * (wp1(i,j,k)*wp1(i,j,k) - 2.0*wijk(i,j,k)*wijk(i,j,k)
+ wm1(i,j,k)*wm1(i,j,k)) + zzcon5 * (u(5,i,j,k+1)*rho_i(i,j,k+1) - 2.0*u(5,i,j,k)*rho_i(i,j,k) +
u(5,i,j,k-1)*rho_i(i,j,k-1)) - tz2 * ( (c1*u(5,i,j,k+1) - c2*square(i,j,k+1))*wp1(i,j,k) - (c1*u(5,i,j,k-1)
- c2*square(i,j,k-1))*wm1(i,j,k))
  endfor
 endfor
endfor
```

Z zaznaczoną instrukcją w tabeli 4.11 nie są powiązane żadne zależności. Zaliczono ją do zbioru instrukcji niezależnych. Z tego powodu w kodzie wyjściowym uzyskano dwie pętle równoległe.

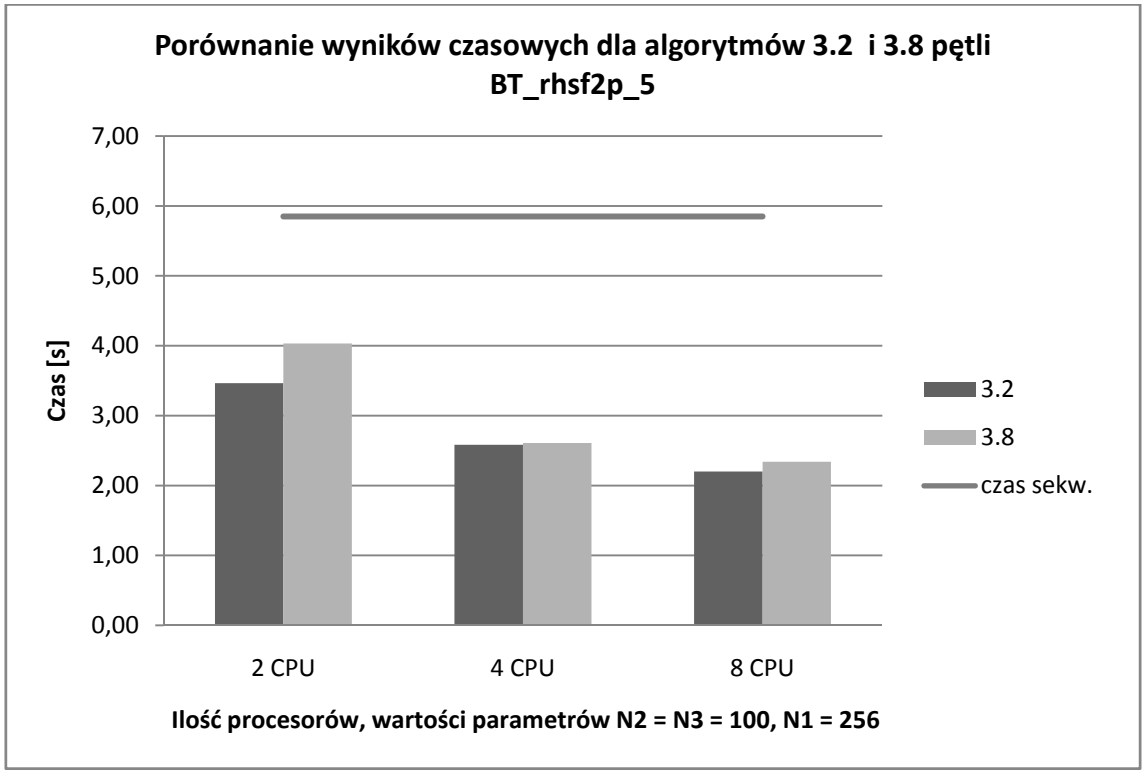
Wyniki badań

Na rysunku 4.9 przedstawiono wyniki czasowe pętli w postaci sekwencyjnej i równoległej uzyskanej algorytmem 3.2. Efektywność obliczeń równoległych rośnie wraz z liczbą iteracji. Najlepszą efektywność uzyskano dla dwóch procesorów. Koszty zarządzania wątkami i czas dostępu do pamięci dzielonej jest dla dwóch wątków najmniejszy. Najkrótszy czas wykonanie pętli bez względu na liczbę iteracji odnotowano dla ośmiu procesorów.



Rys. 4.9 Czas wykonania pętli BT_rhsf2p_5 [op. własne]

Pętla została zrównoleglona także algorytmem 3.8. Na rysunku 4.10 przedstawiono porównanie wyników czasowych. Czas wykonania kodu równoległego wygenerowanego za pomocą algorytmu 3.8 jest nieco dłuższe, lecz wciąż dużo krótsze od czasu wykonania sekwencyjnego odpowiednika. Dla obydwóch algorytmów uzyskano przyspieszenie obliczeń równoległych.



Rys. 4.10 Porównanie zrównoleglenia pętli BT_rhsf2p_5 algorytmami 3.2 i 3.8 [op. własne]

4.4.6. Pętla LU_HP_pintgr.f2p_2 (NAS)

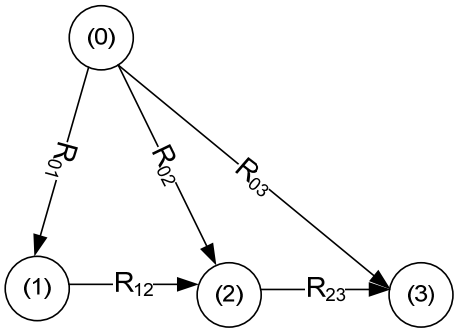
Badana pętla wybrana została z zestawu NAS. Dane wejściowe do algorytmów stanowi oryginalny kod źródłowy zapisany w języku akceptowanym przez program Petit, tab. 4.12 a).

Przebieg analizy

Badana pętla jest idealnie i podwójnie zagnieżdżona. Pętla posiada dolne i górne granice w postaci sparametryzowanej. W pętli znaleziono pięć relacji zależności, pomiędzy którymi występują wspólne początki i końce. Topologia zależności jest zatem grafem ogólnym.

1. {[j,i,0] -> [j,i,1] : N3 <= i <= N4 && N1 <= j <= N2}
2. {[j,i,0] -> [j,i,3] : N3 <= i <= N4 && N1 <= j <= N2}
3. {[j,i,0] -> [j,i,2] : N3 <= i <= N4 && N1 <= j <= N2}
4. {[j,i,1] -> [j,i,2] : N3 <= i <= N4 && N1 <= j <= N2}
5. {[j,i,2] -> [j,i,3] : N3 <= i <= N4 && N1 <= j <= N2}

Dla badanej pętli wyznaczono następujący graf SCC.



Ponieważ wektory zależności dla każdej relacji są jednorodne można zastosować algorytm 3.5 do przebierania iteracji w grafie. Ponadto możliwe jest również wykorzystanie algorytmu 3.8 redukującego topologię zależności do łańcucha oraz algorytmu 3.2.

Do wyznaczenia początków fragmentów kodu wykorzystano algorytm 3.1.

Sources := {[[j,i,0]: N3 <= i <= N4 && N1 <= j <= N2}

W tabeli 4.12 przedstawiono kod wejściowy i równoległy uzyskany algorytmem 3.8.

Tab. 4.12. Postać sekwencyjna i równoległa pętli LU_HP_pintgr.f2p_2

```
for j=N1 to N2 do
  for i=N3 to N4 do
     k(j,i) = ki1
     phi1(i,j) = c2*(  u(5,i,j,k(j,i)) - 0.50 * (  u(2,i,j,k(j,i)) ** 2 + u(3,i,j,k(j,i)) ** 2 +  +u(4,i,j,k(j,i)) **
2 ) / u(1,i,j,k(j,i)) )
     k(j,i) = ki2
     phi2(i,j) = c2*(  u(5,i,j,k(j,i)) - 0.50 * (  u(2,i,j,k(j,i)) ** 2 + u(3,i,j,k(j,i)) ** 2 + +u(4,i,j,k(j,i)) ** 2
) / u(1,i,j,k(j,i)) )
  endfor
endfor
```

(a) Pętla poddana analizie

```
if (N3 <= N4) {
 par for(t1 = N1; t1 <= N2; t1++) {  //par
   for(t2 = N3; t2 <= N4; t2++) { //par
     {
       j = t1;
       i = t2;
       v = 0;
       do
       {
```

```
            if(v==0)
            {
               _k[j][i] = ki1;
            }
            else if(v==1)
            {
                 k  =  _k[j][i];
                 phi1[i][j] = c2 * (u[5][i][j][k] - 0.50 * (  pow(u[2][i][j][k], 2) + pow(u[3][i][j][k], 2) +
pow(u[4][i][j][k], 2 )) / u[1][i][j][k] );
            }
            else if(v==2)
            {
                 _k[j][i] = ki2;
            }
            else
            {
                 k =  _k[j][i];
                 phi2[i][j] = c2 * (u[5][i][j][k] - 0.50 * (  pow(u[2][i][j][k], 2) + pow(u[3][i][j][k], 2) +
pow(u[4][i][j][k], 2 )) / u[1][i][j][k] );
            }
           if(t1 == j && i == t2 && 0 <= v && v <= 2 && N1 <= j && j <= N2 && N3 <= t2 && t2 <=
N4)
           {
             v++;
             continue;
           }
           break;
         }
         while(true);
       }
     }
   } }
```

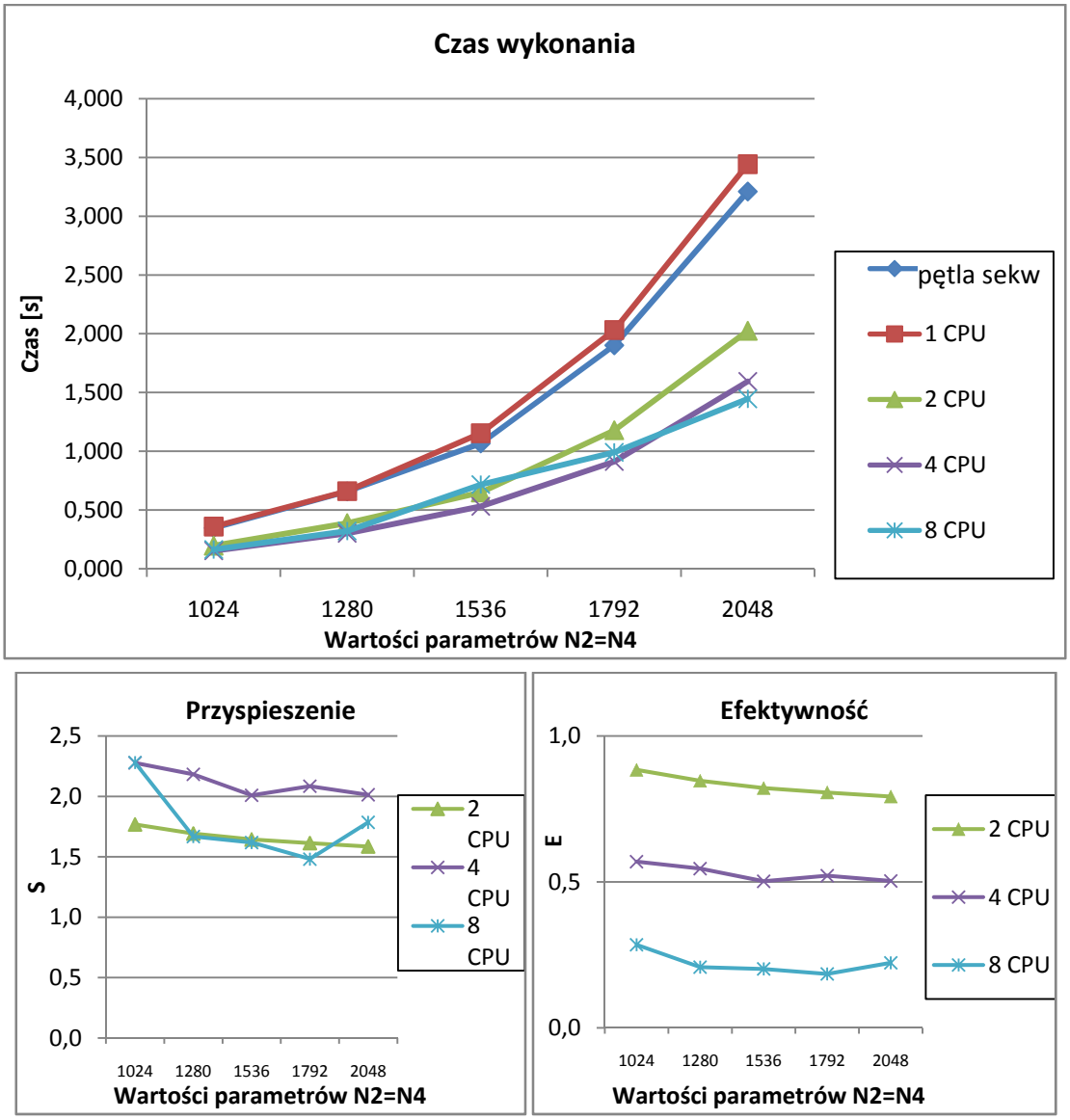
(b) Postać równoległa (algorytm 3.8)

Wyniki badań

Zrównoleglając pętlę za pomocą algorytmu 3.5 z wykorzystaniem dodatkowej tablicy A wprowadza się znaczący koszt odwoływania do dodatkowej struktury danych. Ponadto kod jest dużo bardziej skomplikowany. Postać równoległa uzyskana za pomocą algorytmu 3.8 pozwoliła na znaczne skrócenie czasu pętli równoległej. Najmniej skomplikowany i najbardziej wydajny kod uzyskano algorytmem 3.2. Zdecydowaną różnicę między algorytmami odnotowano dla 2 procesorów i największej badanej liczbie iteracji (2048).

Dla ośmiu procesorów uzyskano gorsze wyniki czasowe niż dla czterech w przypadku wszystkich algorytmów. Koszty zarządzania ośmioma wątkami w tym przypadku okazały się zbyt duże, zwłaszcza w przypadku algorytmu 3.5. Podyktowane to jest transformacją zamiany zmiennych skalarnych na tablicowe oraz wprowadzenie dodatkowej tablicy w przypadku algorytmu 3.5. Rywalizacja o dostęp do pamięci przez 8 wątków związana jest z większymi kosztami czasowymi. Pomimo tego odnotowano

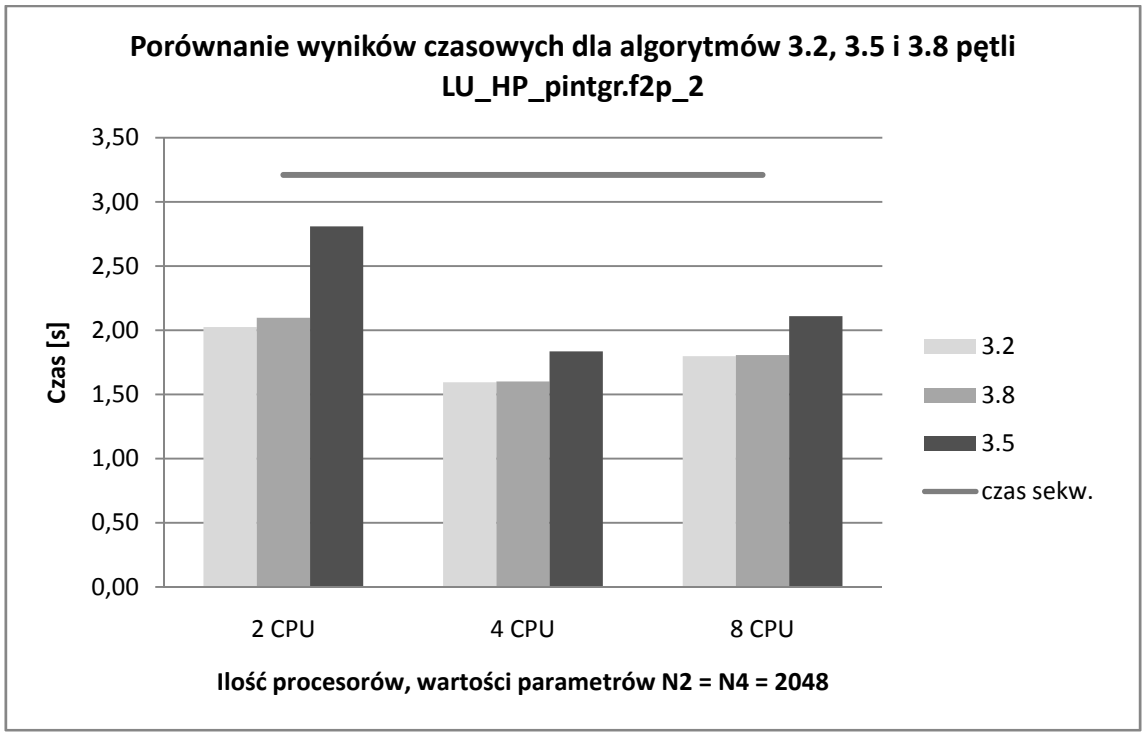
skrócenie czasu wykonania pętli w postaci zrównoleglonej. Zrównoleglenie dla tej pętli jest przede wszystkim korzystne przy użyciu dwóch i czterech procesorów. Zaangażowanie większej liczby procesorów nie pozwoliło na uzyskanie dalszego skrócenia czasu obliczeń równoległych.



Rys. 4.11. Czas wykonania, przyspieszenie i efektywność pętli LU_HP_pintgr.f2p_2 dla algorytmu 3.2 [op. własne]

Na rysunku 4.11 przedstawiono wyniki czasowe dla algorytmu 3.2. Na rysunku 4.12 natomiast zawarto porównanie wyników czasowych dla wszystkich algorytmów. Można zauważyć, że czasy wykonania wszystkich pętli równoległych okazały się krótsze niż czas wykonania sekwencyjnego odpowiednika. Wyniki czasowe pętli 3.2 i 3.8 są porównywalne, natomiast wyniki czasowe 3.5 są znacznie gorsze podyktowane znacznie bardziej skomplikowanym kodem i większym zapotrzebowaniem na pamięć.

Dla tego przykładu zrównoleglonego algorytmami 3.8 i 3.5 długość wygenerowanego kodu wynosi odpowiednio 53 i 230 linii. Podkreśla to zaletę redukcji topologii zależności w celu uzyskania efektywniejszego i krótszego kodu.



Rys. 4.12 Porównanie zrównoleglenia pętli LU_HP_printgr.f2p_2 algorytmami 3.2, 3.5 i 3.8 [op. własne]

4.5. Wnioski z przeprowadzonych badań

W tabeli 4.13 zaprezentowano wyniki badań opisanych wyżej sześciu pętli. W kolumnie pierwszej umieszczono nazwę pętli. W kolumnach 2-7 zawarto przyspieszenie i efektywność dla odpowiednio 2, 4, i 8 procesorów. W kolumnie ósmej podano wartości parametrów. Ostatnia kolumna przedstawia użyte algorytmy w zrównolegleniu. Tabela zawiera porównanie wyników czasowych dla pętli zrównoleglonych na wiele sposobów. W tabeli 4.14 porównano długość kodu (liczba linii) wygenerowanych pętli równoległych.

Tab. 4.13. Zestawienie wyników z przeprowadzonych badań

Pętla	Wykorzy-stane algorytmy	2 CPU		4 CPU		8 CPU		Wartości parametrów granic pętli
		S	E	S	E	S	E	
FT_appft.f2p_1	3.1, 3.2	1,432138	0,716069	1,304807	0,326202	1,400632	0,175079	N1=N2=N3=75
		1,639327	0,819663	2,104162	0,526041	2,244381	0,280548	N1=N2=N3=125
		1,723353	0,861676	2,378130	0,594532	2,809395	0,351174	N1=N2=N3=175
	3.1, 3.8	1,182793	0,591397	1,679825	0,419956	1,340172	0,167521	N1=N2=N3=75
		1,527889	0,763945	2,216099	0,554025	2,421894	0,302737	N1=N2=N3=125
		1,762459	0,881229	2,424170	0,606043	2,487847	0,310981	N1=N2=N3=175
UA_utils.f2p_12	3.1, 3.9	1,508252	0,754126	1,941514	0,485378	2,162287	0,270286	N1=8, N2 = N3 = 100, N4 = 32
		1,659549	0,829775	2,092061	0,523015	2,631807	0,328976	N1=32, N2 = N3 = 100, N4 = 32
		1,741399	0,870700	2,105848	0,526462	2,161466	0,270183	N1=128, N2 = N3 = 100, N4 = 32
Edge_detect_1	3.1, 3.2	1,189109	0,594555	1,483917	0,370979	1,949512	0,243689	N = 1000
		1,471766	0,735883	2,250137	0,562534	3,425652	0,428207	N = 2000
		1,545078	0,772539	2,709477	0,677369	3,848639	0,481080	N = 3000
	3.1, 3.5	0,477612	0,238806	0,745395	0,186349	0,601971	0,075246	N = 1000
		0,514205	0,257103	0,736241	0,184060	0,651305	0,081413	N = 2000
		0,620471	0,310235	0,839580	0,209895	0,700029	0,087504	N = 3000
	3.1, 3.8	1,308766	0,654383	1,520577	0,380144	1,841608	0,230201	N = 1000
		1,439549	0,719774	2,805729	0,701432	3,387632	0,423454	N = 2000
		1,507305	0,753653	2,828495	0,707124	4,388548	0,548569	N = 3000
Compress_1	3.1, 3.2	1,438588	0,719294	1,829526	0,457382	3,043153	0,380394	N = 1024
		1,778246	0,889123	2,561646	0,640411	3,585969	0,448246	N = 1536
		1,772141	0,886071	3,302694	0,825673	4,822089	0,602761	N = 2048
	3.1, 3.4	1,217944	0,608972	1,876131	0,469033	2,225433	0,278179	N = 1024
		1,454198	0,727099	2,216979	0,554245	2,896410	0,362051	N = 1536
		1,753029	0,876515	3,014381	0,753595	3,529404	0,441176	N = 2048
	3.1, 3.8	1,219752	0,609876	1,867287	0,466822	2,165001	0,270625	N = 1024
		1,411673	0,705836	2,466000	0,616500	3,060839	0,382605	N = 1536
		1,754224	0,877112	3,172383	0,793096	4,196222	0,524528	N = 2048

BT_rhsf2p_5	3.1, 3.2	1,450520	0,725260	1,862723	0,465681	2,341982	0,292748	N1 = 96, N2 = N3 = 100
		1,624352	0,812176	2,138801	0,534700	2,716657	0,339582	N1 = 160, N2 = N3 = 100
		1,688488	0,844244	2,264519	0,566130	2,658194	0,332274	N1 = 256, N2 = N3 = 100
	3.1, 3.8	1,482480	0,741240	2,292252	0,573063	2,327101	0,290888	N1 = 96, N2 = N3 = 100
		1,401003	0,700501	2,057201	0,514300	2,709238	0,338655	N1 = 160, N2 = N3 = 100
		1,450822	0,725411	2,243623	0,560906	2,499391	0,312424	N1 = 256, N2 = N3 = 100
LU_HP_pintgr.f2p_2	3.1, 3.2	1,767797	0,883898	2,277410	0,569353	2,279280	0,284910	N1 = N2 = 1024
		1,643410	0,821705	2,009098	0,502275	1,618081	0,202260	N1 = N2 = 1536
		1,585318	0,792659	2,013469	0,503367	1,785091	0,223136	N1 = N2 = 2048
	3.1, 3.5	1,070654	0,535327	1,611880	0,402970	1,559988	0,194998	N1 = N2 = 1024
		1,212045	0,606022	1,671909	0,417977	1,470979	0,183872	N1 = N2 = 1536
		1,142819	0,571409	1,748983	0,437246	1,521981	0,190248	N1 = N2 = 2048
	3.1, 3.8	1,479098	0,739549	1,951386	0,487846	1,852654	0,231582	N1 = N2 = 1024
		1,613576	0,806788	1,906052	0,476513	1,542884	0,192861	N1 = N2 = 1536
		1,530740	0,765370	2,005559	0,501390	1,776684	0,222086	N1 = N2 = 2048

Tab. 4.14. Długość wygenerowanego kodu równoległego (w liniach) dla poszczególnych pętli i wykorzystanych algorytmów

Pętla	Pętla wejściowa	Algorytm					
		3.2	3.3	3.4	3.5	3.8	3.9
FT_appft.f2p_1	26	25	-	-	-	69	-
UA_utils.f2p_12	17	-	-	-	-	-	21
Edge_detect_1	22	13	-	-	117	32	-
Compress_1	19	15	-	68	-	31	-
BT_rhsf2p_5	28	21	-	-	-	46	-
LU_HP_pintgr.f2p_2	24	24	-	-	230	53	-

Wnioski z przeprowadzonych badań są następujące:

- zaproponowane algorytmy w rozdziale trzecim pozwalają uzyskać zbiór niezależnych fragmentów kodu dostępnych w pętlach poddanych analizie,
- w badanych pętlach uzyskano zadowalające wyniki tzn. czas wykonania kodu na maszynie wieloprocesorowej był krótszy niż czas wykonania jego sekwencyjnego odpowiednika,
- zwiększenie ziarna kodu z wykorzystaniem aglomeracji umożliwiło osiągnięcie większego przyspieszenia obliczeń równoległych dla niezależnych fragmentów kodu, z kolei dla kodu z synchronizacją najlepsze przyspieszenie wynika z optymalnego doboru wielkości ziarna kodu z uwzględnieniem redukcji punktów synchronizacji
- skalaryzacja tablic zwłaszcza w algorytmach 3.4 i 3.5 pozwoliła na dodatkową redukcję czasu wykonania kodu i zwiększenie lokalności kodu równoległego,
- w większości przypadków pętlę równoległą uzyskano stosując więcej niż jeden algorytm. Badania dowiodły, że w przypadku pętli równoległej z konstrukcją *while* ważna jest możliwość transformacji topologii grafu zależności. Wygenerowany kod dla topologii łańcucha charakteryzował się zbliżonymi wynikami czasowymi do kodu uzyskanego generatorem pętli przebierającym zbiory afiniczne,
- w badanych pętlach wskazano przykłady, dla których zrównoleglenie za pomocą pętli *while* pozwala na uzyskanie przyspieszenia obliczeń, pomimo kosztów dodatkowych operacji dla wszystkich topologii zależności,
- po wyznaczeniu początków niezależnych fragmentów kodu należy wyznaczyć iteracje za pomocą generatora kodu przebierającego zbiory afiniczne (np. generowanie iteracji za pomocą *codegen* i algorytm 3.2 w celu osiągnięcia najlepszej wydajności. Jeśli jest to niemożliwe i w relacjach zależności występują nieafiniczne wyrażenia lub wynik operacji tranzytywnego domknięcia jest niedokładny wyznaczenie zbioru iteracji umożliwić może wykorzystanie ich przebierania za pomocą pętli *while,*
- w pętlach o mniejszej liczbie iteracji korzyści wynikające z wykonania fragmentów kodu na wielu procesorach nie rekompensowały kosztów poniesionych na zarządzanie wątkami. W większości przypadków wzrostowi iteracji i obliczeń towarzyszył również wzrost wydajności obliczeń równoległych,
- zwiększenie liczby wątków (jeden wątek mapowany na jeden procesor) do wykonania tego samego zadania najczęściej prowadzi do gorszej efektywności

kodu z powodu kosztów zarządzania wątkami. Wyjątek stanowiła pętla UA_utils.f2p_12, w której zastosowano algorytm 3.9. Przyporządkowanie procesorom największej możliwej liczby synchronizowanych wątków spowodowało uzyskanie maksymalnej równoległości i zredukowanie czasu przestojów. Aglomeracja obliczeń w tym przypadku prowadziłoby tylko do sekwencyjnego wykonywania kolejnych bloków,

- w badanych przykładach zastosowanie harmonogramowania swobodnego iteracji pętli okazało się wystarczające, z powodu między innymi równych porcji obliczeń w fragmentach oraz możliwości skorzystania w pełni mocy obliczeniowej dostępnej na maszynie równoległej,
- pomimo wprowadzenia transformacji zamiany zmiennych na tablice i redukcji zależności, w większości badanych pętli nadal występowały wspólne iteracje pomiędzy zależnościami. Topologia w takich przypadkach to drzewo lub graf ogólny (nie drzewo). W większości przypadków uniemożliwia to zastosowanie algorytmów przedstawionych w pracy [85]. Wyznaczenie wydajnego kodu równoległego za pomocą zaproponowanych algorytmów w niniejszej pracy pozwoliło na zwiększenie ekstrakcji równoległości w badanych zestawach testowych.

Pełne zestawienie wyników czasowych i statystyk z przeprowadzonych badań umieszczono w załączniku B na płycie CD. Zestawienie statystyk pozostałych pętli z badanych zestawów testów NPB i UTDSP oraz kod równoległy wszystkich badanych pętli uzyskany za pomocą przedstawionych w pracy algorytmów zawarto także w załączniku B.

5. PRACE POKREWNE

Niniejsza dysertacja stanowi rozwinięcie badań nad zrównolegleniem pętli programowych za pomocą znajdowania niezależnych fragmentów kodu (ang. *slices*). W pracach Pugha i Rossera [80] zaproponowano partycjonowanie przestrzeni iteracji (ang. *iteration space slicing framework*). Podział przestrzeni podyktowany jest optymalizacją komunikacji między procesorowej – transformacje łączenia pętli (ang. *loop fusion*), tolerancji opóźnień w przesyłaniu komunikatów i możliwości ich łączenia. Podobnie jak w zaproponowanych algorytmach wykorzystano operację tranzytywnego domknięcia w wyznaczeniu niezależnych fragmentów kodu (ang. *synchronization-free slices*). Jednak Pugh i Rosser nie zaproponowali w [80] algorytmu automatycznego wyznaczenia niezależnych fragmentów kodu. Nie uwzględniono również możliwości zrównoleglenia pętli programowych za pomocą fragmentów kodu z synchronizacją.

Transformacje unimodularne umożliwiają uzyskanie równoległości gruboziarnistej w pętlach idealnie zagnieżdżonych z przemieszczeniem do najbardziej zewnętrznego gniazda pętli. Do transformacji unimodularnych zalicza się transformacje zmiany kolejności pętli (ang. *loop interchanges*), odwrócenie pętli (ang. *loop reversal*), przekoszenie pętli (ang. *loop skewing*) oraz kombinacje tych transformacji [9], [92], [93]. Transformacje unimodularne posiadają jednak kilka istotnych ograniczeń:

- operacje należące do pojedynczej iteracji są niepodzielne - uniemożliwia to opisanie ważnych transformacji np. podział i dystrybucja pętli lub zmiana kolejności wykonania wyrażeń. W zaproponowanym algorytmie wyznaczania początków niezależnych fragmentów kodu (podrozdział 3.3) wprowadzono dodatkową zmienną w relacjach zależności oznaczająca numer instrukcji pętli, umożliwia to wyznaczenie równoległości również pomiędzy instrukcjami tej samej iteracji,

- zrównoleglenie pętli nieidealnie zagnieżdżonych jest bezpośrednio niemożliwe - istnieją jedynie rozwiązania pozwalające na przekształcenie pętli nieidealnej do pętli idealnie zagnieżdżonej [95],
- 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*),
- tranformacje unimodularne nie umożliwiają ekstrakcji równoległości gruboziarnistej, w przypadku gdy pomiędzy fragmentami istnieją zależności.

J.M. Anderson i M.S. Lam [5] przedstawili algorytm pozwalający na automatyczną dekompozycję danych oraz obliczeń z optymalizacją zarówno równoległości jak i lokalności. Zaproponowane rozwiązanie jednak ograniczone jest do sekwencji idealnie zagnieżdżonych pętli.

Afiniczne transformacje (ATF – ang. affine transformation framework) są obecnie najbardziej rozwiniętą techniką ekstrakcji równoległości [13], [35], [38], [39], [65], [66], [67]. Łączy ona dużą klasę transformacji [53] i pozwala na uzyskanie równoległości pozbawionej synchronizacji dla maszyn z pamięcią dzieloną oraz równoległości pozbawionej komunikacji w systemach rozproszonych. Transformacje ATF mogą zostać użyte dla instrukcji wektorowych oraz architektur SIMD i MIMD. Zwiększają także lokalność kodu i redukują zapotrzebowanie na pamięć.

Główną ideą ATF jest afiniczne przekształcenie pętli w taki sposób, że zależne od siebie operacje składają się na tą samą część. Transformacje opisują afiniczne mapowanie każdej instrukcji pętli. M-wymiarowe mapowanie części dla instrukcji *s* pętli jest m-wymiarowym afinicznym wyrażeniem:

$$\overline{\phi_s} = C_s \bar{i} + \overline{c_s}$$

Znalezienie transformacji afinicznej odbywa się następująco [35], [38], [39]: dla zbioru q zależności $D = \{\bar{I}_j \rightarrow \bar{J}_j, j = 1,2,....,q\}$ należy znaleźć m-wymiarową afiniczną przestrzeń podziału mapowania $\overline{\phi_s} = C_s \bar{i} + \overline{c_s}$ taką, że:

$$\bar{\phi}_{si}(\overline{I_j}) = \bar{\phi}_{sik}(\overline{J_j}) = \overline{P_m}$$

gdzie *si* i *sk* są instrukcjami, pomiędzy którymi zachodzi zależność $\bar{I}_j \rightarrow \bar{J}_j$; C_s jest $m \times n$ macierzą stałych, $\overline{P_m}$ jest wektorem identyfikującym procesor, który wykonuje instrukcje tworzące zależność.

Należy zaznaczyć, że metoda ATF nie wykrywa całkowitej równoległości zawartej w pętlach programowych [13]. W celu wykazania, że proponowane rozwiązania niniejszej dysertacji umożliwiają zwiększenie ekstrakcji równoległości w pętlach programowych, dokonano porównania rozwiązań w rozdziale 5.2.

Feautrier przedstawił algorytm wyszukujący *„picewise affine schedule”* do minimalizacji czasu wykonania programu [38], [39]. Algorytm Feautriera pozwala ustalić moment wykonania instrukcji bez przypisania jej do jednostki wykonawczej. Uzyskanie kodu równoległego za pomocą tej transformacji nie jest zadaniem prostym, a kod wynikowy może być skomplikowany [93]. Zgodnie z wiedzą autora ta metoda, ze względu na dużą złożoność, nie została zaimplementowana w żadnym z obecnie dostępnych kompilatorów.

Jedyne znane autorowi rozwiązania pozwalające na automatyczne wyznaczenie niezależnych fragmentów kodu z wykorzystaniem partycjonowania przestrzeni pętli zawarto w dysertacji Siedleckiego [85] i w publikacjach [19], [21], [22], [23]. Opisane algorytmy służą do wyszukiwania drobno- i gruboziarnistej równoległości w pętlach programowych z zależnościami afinicznymi. Proponowane rozwiązania zostały zbadane z wykorzystaniem testów NAS i Livermoore [85]. Zaprezentowane algorytmy posiadają następujące ograniczenia:

- niemożliwe jest wyznaczenie niezależnych fragmentów kodu dla części przestrzeni iteracji pętli, w której relacje zależności opisują wspólne iteracje (topologie: drzewo, graf),
- niemożliwe jest zrównoleglenie pętli sparametryzowanej gdy wyrażenia tranzytywnego domknięcia są nieliniowe w ogólnym przypadku.

Przedstawione rozwiązanie wykorzystuje do zrównoleglenia pętli również operacje tranzytywnego domknięcia oraz arytmetykę Presburgera. Zawarte wyniki niniejszej dysertacji stanowią kontynuację tych badań. W celu zbadania zwiększenia ekstrakcji równoległości dokonano porównania efektywności obu rozwiązań w zrównolegleniu testów NAS. Wyniki umieszczono w rozdziale 5.3.

5.1. Zwiększenie ekstrakcji równoległości w porównaniu z transformacjami afinicznymi

Wyróżnia się następujące ograniczenia transformacji ATF w przypadku ekstrakcji równoległości [13], [16]:

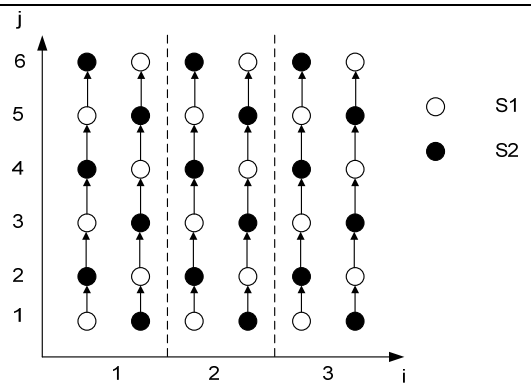
- równoległości pozbawionej synchronizacji zawartej w części domeny pętli,
- równoległości wymagającej synchronizacji,
- równoległości w ogólnym przypadku pętli niejednorodnych,
- ekstrakcji wszystkich wątków zawartych w ciele pętli,
- ekstrakcji wątków wymagających synchronizacji nie pozwalając na zwiększenie lokalności kodu i (lub) zmniejszenie zapotrzebowania na zasoby pamięciowe,
- równoległości z zależnościami nieafinicznymi.

Transformacje ATF w ogólnym przypadku nie pozwalają na ekstrakcję wszystkich wątków w pętli. W tabeli 5.1 zawarto przykład pętli zaczerpnięty z pracy [13]. Zrównoleglając pętle transformacją ATF uzyskano *n* niezależnych wątków. Analizując zależności i przestrzeń iteracji można zauważyć, że pętla równoległa składa się z *2n* wątków. W celu porównania zrównoleglono pętle algorytmami 3.1 i 3.2. Już na etapie znajdowania początków dokonano ekstrakcji dwóch zbiorów *n* początków, a następnie wygenerowano 2 niezależne pętle zawierające *n* niezależnych fragmentów kodu – łącznie *2n* fragmentów kodu.

Tab. 5.1. Zrównoleglenie pętli i uzyskanie wszystkich niezależnych fragmentów kodu

Pętla wejściowa – przykład nr 5 z pracy [13]	**Kod równoległy uzyskany proponowanymi algorytmami 3.1 i 3.2**

for i=1 to n do
  for j=1 to m do
  a(i,j) = b(i,j)+c(i,j)
  a(i,j-1) = a(i,j+1)
  endfor
endfor
	if (m >= 2) {
	par for(t1 = 1; t1 <= n; t1++) {
	a(t1,1) = b(t1,1)+c(t1,1);
	if (t1 >= 1 && m >= 2 && n >= t1) {
	a(t1,2-1) = a(t1,2+1) ;
	}
	if (t1 >= 1 && n >= t1) {
	for(t2 = 3; t2 <= m; t2++) {
	if (intMod(t2+1,2) == 0) {
	a(t1,t2) = b(t1,t2)+c(t1,t2);
	}
	if (intMod(t2,2) == 0) {
	a(t1,t2) = a(t1,t2);



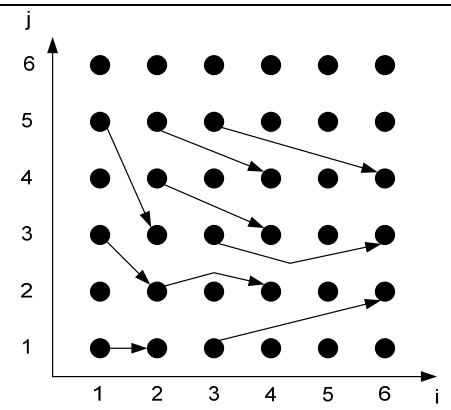

```
        }
       }
      } } }

    if (m >= 2) {
     par for(t1 = 1; t1 <= n; t1++) {
      a(t1,1) = a(t1,1);
      if (t1 >= 1 && m >= 2 && n >= t1) {
       a(t1,2) = b(t1,2)+c(t1,2);
      }
      if (t1 >= 1 && n >= t1) {
       for(t2 = 3; t2 <= m; t2++) {
        if (intMod(t2,2) == 0) {
         a(t1,t2) = b(t1,t2)+c(t1,2);
        }
        if (intMod(t2+1,2) == 0) {
         a(t1,t2) = a(t1,t2);
        }
       } }
     } }
```

W pracy [13] przytoczono inny przykład (tabela 5.2.), który dowodzi, że w ogólnym przypadku pętli niejednorodnych transformacje ATF zawodzą. Na podstawie układu równań afinicznych zależności pętli stwierdzono, że nie pozwala on za pomocą transformacji ATF dokonać ekstrakcji niezależnych wątków. Analizując ten przykład i wykorzystując algorytmy 3.1 i 3.3 uzyskano *min(intDiv(n,2)2*m-1)* niezależnych fragmentów kodu. W tabeli 5.2 zawarto przykład i uzyskaną postać równoległą.

Tab. 5.2. Zrównoleglenie pętli i uzyskanie wszystkich niezależnych fragmentów kodu

Pętla wejściowa – przykład nr 4 z pracy [13]	**Kod równoległy uzyskany proponowanymi algorytmami 3.1 i 3.3**

for i=1 to n do
  for j=1 to m do
  a(2*j+3,i+1) = a(i+j+3,2*i+1)
  endfor
endfor
	par for(t1 = 1; t1 <= min(intDiv(n,2),2*m-1); t1++) {
	  for(t2 = 1+intMod((-t1-1),2); t2 <= min(-t1+2*m,intDiv(t1+1,4)); t2 += 2) {
	  while_body(t1,t2);
	  }
	  if (intMod(-t1-1,2) == 0) {
	  for(t2 = 2*intDiv(intDiv(t1+3+3,4)--1+1,2)+-1; t2 <= min(intDiv(2*m+t1-1,4),2*m-t1); t2 += 2) {
	  while_body(t1,t2);
	  }
	  }
	  if (intMod(-t1,2) == 0) {
	  }



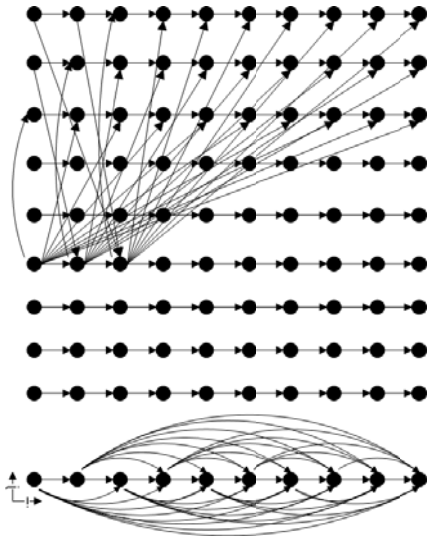

```
  for(t2 = 2*intDiv(intDiv(t1+2*m+1+3,4)--t1+1,2)+-t1; t2
<= min(-t1+2*m,m); t2 += 2) {
    while_body(t1,t2);
  }
}

void while_body(int t1, int t2, int t3)
{
   int i, j, ip, jp;
   ip = i = t1;
   jp = j = t2;
   do
   {
     a(2*jp+3,ip+1) = a(ip+jp+3,2*ip+1);
     if(is_int((float)( -(-i - j)/2)) && 1 <= j && j <= m && j+i
<= 2*m && 1 <= i && 2*i<= n)
     {
       ip = 2*i;
       jp = i/2 + j/2;
       i = ip;
       j = jp;
      continue;
     }
     break;
   }
   while(true);
}
```

W pracy [13] opisano przykład, który dowodzi niemożliwość ekstrakcji równoległości wolnej od synchronizacji zawartej w poddomenie dziedziny pętli. Dla poniższej pętli w tabeli 5.3 nie można uzyskać afinicznej transformacji pozwalająca na ekstrakcję równoległości z powodu braku niezerowego rozwiązania [13]. Analizując ten przykład i wykorzystując algorytmy 3.1 i 3.2 uzyskano 7 niezależnych fragmentów kodu.

Tab. 5.3. Zrównoleglenie pętli i uzyskanie wszystkich niezależnych fragmentów kodu

Pętla wejściowa – przykład nr 2 z pracy [13]	**Kod równoległy uzyskany proponowanymi algorytmami 3.1 i 3.2**

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
	par for(t2 = 1; t2 <= 7; t2++) {
	  for(t1 = 1; t1 <= 10; t1++) {
	  if (t2 <= 10 && t1 >= t2-7 && t2 >= 9) {
	  a(t2+4,t2+1) = a(t1+2*t2+1,t1+t2+3);
	  }
	  if (t1 <= 1) {
	  a(t2+4,t2+1) = a(t1+2*t2+1,t1+t2+3);
	  }

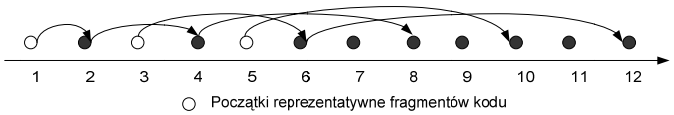


```
    for(t2' = max(-t1+t2+2,t2); t2' <=
min(intDiv(9*t2-1,8),10); t2'++) {
      a(t2'+4,t2'+1) = a(t1+2*t2'+1,t1+t2'+3);
    }
    if (t2 == 5) {
      for(t2' = 8; t2' <= 10; t2'++) {
        a(t2'+4,t2'+1) = a(t1+2*t2'+1,t1+t2'+3);
      }
    }
  }

}
```

Dla pętli podanej w tabeli 5.4 znalezienie dokładnego wyniku tranzytywnego domknięcia w celu wyznaczenia afinicznego zbioru iteracji za pomocą narzędzia Omega Calculator jest niemożliwe. Uzyskana relacja posiada ograniczenia niedokładne typu UNKNOWN. Za pomocą znanych transformacji afinicznych nie można dokonać zrównoleglenia pętli o zależnościach niejednorodnych w ogólnym przypadku. Ekstrakcję równoległości uzyskano wykorzystując algorytmy 3.1 i 3.3. Topologia zależności jest łańcuchem. Uzyskano *intDiv(n,2)* niezależnych fragmentów kodu [16].

Tab. 5.4 Zrównoleglenie pętli i uzyskanie wszystkich niezależnych fragmentów kodu

Pętla wejściowa	**Kod równoległy uzyskany proponowanymi algorytmami 3.1 i 3.3**

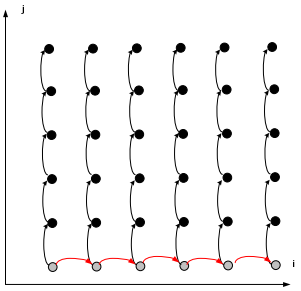
for i=1 to n do
a(i)=a(2*i)
endfor
 
 	
	par for(t1 = 1; t1 <= intDiv(n,2);
	t1 += 2) {
	i = t1;
	do
	{
	a(i) = a(2*i);
	if(1 <= i && 2*i <= n)
	{
	i *= 2;
	continue;
	}
	break;
	}
	while(true); }

Transformacje afiniczne uniemożliwiają także ekstrakcję wątków z synchronizacją [13]. W rozdziale 3.6 opisano zrównoleglenie pętli za pomocą fragmentów kodu wymagających synchronizacji. Wykazano, że dla pętli w tabeli 5.5 niemożliwe jest dokonanie zrównoleglenia za pomocą transformacji afinicznych. Analizując pętle dokonano ekstrakcji równoległości wykorzystując algorytmy 3.1 i 3.9. Uzyskano *n* fragmentów kodu z synchronizacją.

W tabeli 5.6 podsumowano porównanie proponowanych algorytmów z transformacjami ATF.

Tab. 5.5. Zrównoleglenie pętli i uzyskanie wszystkich niezależnych fragmentów kodu

Pętla wejściowa	**Kod równoległy uzyskany proponowanymi algorytmami 3.1 i 3.9.**

for(i=1; i<=n; i++)
for(j=1; j<=n; j++)
a(i,j) = a(i,j-1) + a(i-1,1);
 
	if (n >= 2) {
	par for(t1 = 1; t1 <= n; t1++) {
	if(2 <= t1 && t1 <= n)
	recv(t1-1,1);
	a(t1,1) = a(t1,0) + a(t1-1,1);
	if(1 <= t1 && t1 < n)
	send(t1,1);
	if (n >= t1 && t1 >= 1) {
	for(t2 = 2; t2 <= n; t2++) {
	a(t1,t2) = a(t1,t2-1) +
	a(t1-1,1);
	}
	}
	}

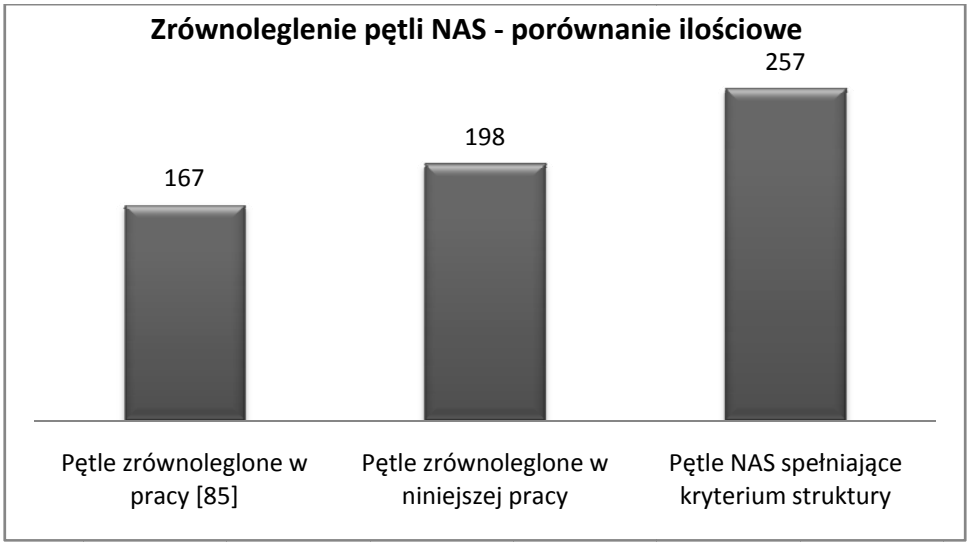
Tab. 5.6. Zwiększanie ekstrakcji równoległych fragmentów kodu – porównanie z transformacjami ATF

Przykład z tabeli	**Liczba wątków po zastosowaniu ATF**	**Liczba fragmentów kodu po zastosowaniu zaproponowanych algorytmów**
5.1	n	2n
5.2	0	min(intDiv(n,2)2*m-1)
5.3	0	7
5.4	0	intDiv(n,2)
5.5	0	n

5.2 Porównanie wyników badań do rozwiązań opartych o algorytmy do ekstrakcji równoległości w pętlach z zależnościami afinicznymi

W pracy Siedleckiego przedstawiono algorytmy wyszukiwania drobno-i gruboziarnistej równoległości w pętlach programowych z zależnościami afinicznymi [85]. Algorytmy zostały zbadane również na zestawie pętli testowych NAS. W niniejszym rozdziale dokonano porównania pomiędzy rozwiązaniami z pracy [85] i proponowanymi algorytmami w niniejszej dysertacji.

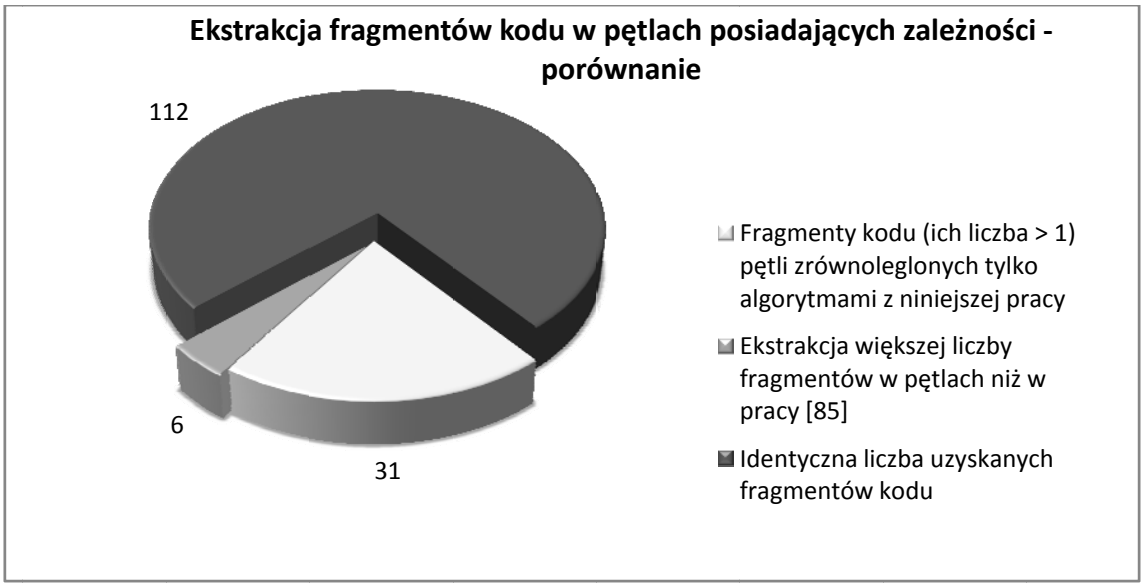
Zbiór badanych pętli NAS spełniających kryterium struktury liczy 257 pętli. Wykorzystując algorytmy [85] uzyskano kod równoległy dla 167 pętli (65%). Zaproponowane algorytmy w niniejszej pracy pozwoliły na uzyskanie równoległego kodu w 198 pętlach (77%). Porównanie zaprezentowano na rysunku 5.1.



Rys. 5.1. Porównanie liczby zrównoleglonych pętli z zestawu NAS w porównaniu z rozwiązaniem [85] [op. własne].

W podrozdziale 4.1 wprowadzono dodatkowe kryterium badań, w którym analizowane są tylko te pętle, które zawierają zależności danych. Według tego kryterium zbiór NAS zawężono z 257 do 149. Z tego zbioru w pracy [85] wyznaczono kod równoległy dla 59 pętli. Korzystając z algorytmów zaprezentowanych w niniejszej pracy zrównoleglenia dokonano dla 90 pętli.

Dla 112 pętli uzyskano ten sam zbiór początków i tą samą liczbę fragmentów. **Dla 37 pętli dokonano zwiększenia stopnia równoległości w porównaniu z rozwiązaniem** [85] - dla 31 pętli uzyskano zbiór fragmentów, dla których nie uzyskano kodu równoległego algorytmami z pracy [85], dla pozostałych 6 pętli uzyskano większą liczbę (sparametryzowaną) fragmentów kodu niż w pracy [85], (rysunek 5.3.2).



Rys. 5.2. Ekstrakcja równoległości w zestawie pętli NAS – porównanie dwóch podejść z niniejszej pracy i [85], [op. własne].

W tabeli 5.7 przedstawiono pętle, w których uzyskano inną postać zbioru początków i liczbę równoległych fragmentów kodu.

Tab. 5.7 Porównanie ekstrakcji równoległości (liczby fragmentów kodu) algorytmami przedstawionymi w pracy [85] oraz algorytmami zaproponowanymi w niniejszej pracy.

Lp.	**Pętla**	**Liczba fragmentów uzyskanych zaproponowanymi algorytmami w pracy [85]**	**Liczba fragmentów uzyskanych zaproponowanymi algorytmami w niniejszej pracy**
1	BT_initialize.f2p_2	1	N1+1
2	BT_initialize.f2p_3	1	N1+1
3	BT_initialize.f2p_4	1	N1+1
4	BT_initialize.f2p_5	1	N1+1
5	BT_initialize.f2p_6	1	N1+1
6	BT_initialize.f2p_7	1	N1+1
7	BT_initialize.f2p_8	0	5
8	BT_initialize.f2p_9	0	5
9	BT_rhsf2p_5.t	0	N1*N2*N3
10	CG_cg.f2p_4.t	1	N z synchronizacją
11	CG_cg.f2p_7.t	0	2*N1*(N3-N2 + 1) +N1
12	FT_appft.f2p_1.t	0	N1
13	LU_erhs.f2p_2.t	4	N1
14	LU_HP_erhs.f2p_2.t	0	N1
15	LU_HP_pintgr.f2p_2.t	0	N2-N1+1
16	LU_HP_rhs.f2p_1.t	0	N1*N2*N3
17	LU_pintgr.f2p_2.t	0	N2-N1+1
18	LU_rhs.f2p_1.t	0	N1*N2*N3

19	MG_mg.f2p_1	0	3
20	MG_mg.f2p_3	4	N2-N1
21	MG_mg.f2p_9	4	N2-N1
22	MG_mg.f2p_11.t	0	N1-2
23	MG_mg.f2p_12.t	0	N1-2
24	MG_mg.f2p_13.t	0	N1-2
25	SP_initialize.f2p_2	3	N1+1
26	SP_initialize.f2p_3	3	N1+1
27	SP_initialize.f2p_4	1	N1+1
28	SP_initialize.f2p_5	1	N1+1
29	SP_initialize.f2p_6	1	N1+1
30	SP_initialize.f2p_7	1	N1+1
31	SP_initialize.f2p_8.t	0	10
32	UA_diffuse.f2p_2.t	1	N1 z synchronizacją
33	UA_precond.f2p_5	0	N1
34	UA_setup.f2p_6	2	(intDiv(lx1,2)-max(-N1+lx1+1,4)+1)+ (intDiv(lx1-1,2)-max(-N2+lx1+1,1))+3
35	UA_utils.f2p_12.t	1	N1 z synchronizacją
36	UA_setup.f2p_15.t	0	N1
37	UA_setup.f2p_1.t	0	N1

Porównując wyniki prezentowanych algorytmów w niniejszej pracy i przedstawionych wyników zrównoleglenia testów NAS w [85] stwierdzono, że dla 149 pętli posiadających zależności w zestawie NAS **uzyskano kod równoległy dla 37 pętli więcej, tj. 25%.**

Wyniki przeprowadzonej analizy porównawczej w niniejszym rozdziale pozwalają wysunąć wniosek, że została potwierdzona teza pracy - zaproponowane algorytmy umożliwiają zwiększenie ekstrakcji równoległości w pętlach programowych w porównaniu do transformacji afinicznych [35], [38], [39], [65], [66], [67] i metod automatycznego znajdowania zbioru niezależnych fragmentów kodu [80], [85].

6. PODSUMOWANIE

W niniejszej pracy zaproponowano zbiór algorytmów umożliwiających zwiększenie stopnia ekstrakcji równoległości (rozdział 3) poprzedzając niezbędną wiedzą dla zrozumienia rozwiązań z naciskiem na wyjaśnienie podstaw arytmetyki Presburgera (rozdziały 1 i 2). W rozdziale 4 przedstawiono metodykę badań eksperymentalnych, charakterystykę systemów badawczych oraz wyniki przeprowadzonych eksperymentów z zastosowaniem autorskiego narzędzia. Rozwiązania porównano z transformacjami afinicznymi oraz metodami znajdowania niezależnych fragmentów kodu (rozdział 5).

Zaproponowano algorytm wyszukiwania początków fragmentu kodu dla podziału przestrzeni iteracji z uwzględnieniem obecności wspólnych początków i końców zależności. Przedstawiono algorytm wyznaczający afiniczny zbiór iteracji dla niezależnych fragmentów kodu. Zaprezentowano także algorytmy do przebierania iteracji w czasie wykonania za pomocą konstrukcji *while* dla dowolnej topologii zależności oraz algorytmy umożliwiający uproszczenie topologii. Zaproponowano rozwiązanie umożliwiające ekstrakcję równoległości z jednego niezależnego fragmentu kodu w postaci zbioru zależnych fragmentów kodu z synchronizacją.

W rozdziale piątym zawarto opis prac pokrewnych do zagadnienia poruszanego w niniejszej dysertacji. Poruszono kwestię ograniczeń istniejących rozwiązań oraz pokazano zalety zaproponowanego podejścia. Zaprezentowano zbiór przykładów, dla których jedna z najmocniejszych aktualnie metod (ATF) zawodzi podczas ekstrakcji równoległości, natomiast zaproponowane podejście pozwala na ich zrównoleglenie. Dokonano również porównania algorytmów z rozwiązaniem opartym na podziale przestrzeni iteracji [85] z wykorzystaniem zestawu testów NAS Benchmark.

Do najważniejszych wyników pracy należą:

- algorytmy umożliwiające zwiększenie stopnia ekstrakcji równoległości w pętlach programowych,
- stworzenie narzędzia akademickiego będącego implementacją zaproponowanych algorytmów i generatorem kodu.

Główną zaletą opracowanych algorytmów jest to, że pozwalają na wyznaczenie równoległości w pętlach programowych, dla których zawodzą najskuteczniejsze algorytmy przekształcenia pętli znane autorowi. Implementacja proponowanych algorytmów oraz badania eksperymentalne pozwalają stwierdzić, że możliwe jest użycie opracowanych rozwiązań w kompilatorach akademickich i przemysłowych do automatycznej transformacji pętli sekwencyjnych na ich odpowiedniki równoległe.

6.1. Wnioski końcowe

W oparciu o zaproponowane algorytmy oraz przeprowadzone badania eksperymentalne sformułowano następujące wnioski końcowe:

- Zastosowanie zbioru algorytmów 3.1 – 3.9 pozwala na wyznaczenie gruboziarnistej równoległości w pętlach sparametryzowanych idealnie oraz dowolnie zagnieżdżonych, jednorodnych i niejednorodnych.
- Niezależne fragmenty kodu przyczyniają się do zwiększenia i efektywnego wykorzystania lokalności kodu, w szczególności lokalności tymczasowej (ang. *temporal locality*).
- Zastosowanie algorytmu 3.1 pozwala na wyszukanie początków fragmentów kodu o dowolnej topologii zależności, tj. w przypadku, gdy występują wspólne początki i końce pomiędzy relacjami zależności.
- 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.
- Algorytmy redukujące topologię zależności pozwalają na uzyskanie kodu równoległego o lepszej wydajności.
- W przypadku, gdy niemożliwe jest wyznaczenie niezależnych fragmentów kodu dla całej przestrzeni iteracji pętli, można dokonać próby zrównoleglenia pojedynczego fragmentu i wyznaczenie równoległości z synchronizacją w postaci zbioru zależnych fragmentów kodu.
- Wyznaczenie kodu równoległego, za pomocą pętli dopóki (typu *while*), dla pętli o dowolnej topologii zależności umożliwia uzyskanie przyspieszenia

obliczeń i umożliwia ekstrakcję równoległości w przypadku, gdy niemożliwe jest afiniczne przebieranie iteracji fragmentów kodu.

- Redukcja zależności danych i ich topologii pozwala na uzyskanie bardziej wydajnego kodu równoległego.
- Przebieranie iteracji w czasie wykonania za pomocą pętli dopóki (ang. *while*) pozwala na zwiększenie wydajności kodu równoległego z synchronizacją i z aglomeracją obliczeń.
- W oparciu o zaproponowane algorytmy możliwe jest opracowanie kompilatorów automatycznie generujących kod równoległy, którego czas wykonania za pomocą wielu procesorów jest krótszy niż jego sekwencyjny odpowiednik.
- Zaproponowane w niniejszej pracy algorytmy umożliwiają wyznaczenie równoległości dla większego spektrum pętli w porównaniu do znanych przekształceń pętli, w tym transformacji afinicznych i transformacji bazujących na podziale przestrzeni iteracji pętli.
- W ramach prac badawczych opracowano w pełni zautomatyzowane narzędzie do generowania kodu według zaproponowanych algorytmów. W jego skład wchodzą rozwiązania wspomagające możliwości biblioteki Omega Calculator oraz udoskonalono mechanizmy przetwarzania relacji i zbiorów. Generator kodu został zaimplementowany w środowisku .NET z wykorzystaniem pakietu Wolfram Mathematica.
- Zaproponowane algorytmy pozwalają na udzielenie odpowiedzi na wiele pytań z dziedziny transformacji pętli programowych m.in.:
  - Czy możliwe jest zwiększanie ekstrakcji równoległości w porównaniu do aktualnie znanych, najmocniejszych metod i jakie koszty się z tym wiążą ?
  - Jaka jest jakość kodu równoległego z synchronizacją?
  - Czy przebieranie iteracji w czasie wykonania kodu umożliwia przyspieszenie obliczeń równoległych?
  - Czy wygenerowany kod jest efektywny?
  - Jak dużo równoległości zostało pominiętych przez znane transformacje?

6.3. Dalsze badania

W trakcie badań napotkano na wiele ograniczeń i problemów, których rozwiązanie umożliwiłoby zwiększenie – i tak już szerokiego - zakresu stosowalności proponowanych rozwiązań:

- 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.
- Rozszerzenie możliwości istniejących narzędzi o nowe algorytmy w przetwarzaniu dowolnych relacji oraz zbiorów, a także udoskonalenie mechanizmu upraszczania ich postaci.
- Wzbogacenie implementacji algorytmów o możliwość ekstrakcji równoległości z wewnętrznie zagnieżdżonych pętli oraz z sekwencji niepowiązanych ze sobą pętli.
- Opracowanie algorytmów umożliwiających wyznaczenie pełnej równoległości z synchronizacją, z jednoczesną redukcją liczby komunikatów i czasów przestoju, dobierających automatycznie podział relacji, porządek wykonania iteracji oraz liczbę makro-fragmentów. Innymi słowy znalezienie uniwersalnego rozwiązania umożliwiającego automatyczną analizę i wyznaczenie optymalnego kodu z synchronizacją w ogólnym przypadku.

Powyższymi zagadnieniami autor pragnie zająć się w swoich przyszłych badaniach.

ZAŁĄCZNIK A

SPECYFIKACJA WYKORZYSTANEGO SPRZETU DO BADAŃ ESKPERYMENTALNYCH

Badania eksperymentalne przeprowadzono z wykorzystaniem komputera równoległego **Workstation Board Intel Xeon S5000XVN Quad Core** wyposażonej w procesory Intel Quad-Core znajdującej się na Wydziale Informatyki Politechniki Szczecińskiej. Charakterystyka maszyny badawczej została zamieszczona poniżej.

Intel Quad-Core

Tab. A.1. Charakterystyka systemu Intel Quad-Core

Procesor:	Intel Xeon E5310, 1.6 GHz
Liczba procesorów:	2 czterordzeniowe
Max. liczba jednostek przetwarzających dla użytkownika:	8
Pomięć operacyjna:	2 GB
Pojemność dyskowa:	400 GB
System operacyjny:	Ubuntu Linux
Kompilator:	GCC (ver. 4.2)

ZAŁĄCZNIK B

PŁYTA CD

Na dołączonej płycie CD w zależności od katalogu umieszczono następujące dodatki:

- SFS_Project – implementacja algorytmów, narzędzie w postaci skompilowanej, w wersji instalacyjnej oraz źródła i dokumentacja. Źródła zapisane w projekcie Microsoft Visual C++ 6. Wymagania: .NET Framework 2.0 i Wolfram Mathematica 5.2.
- Experiments – wyniki badań:
  - zestawienie wszystkich wyników czasowych, wartości przyspieszenia i efektywności dla badanych pętli w rozdziale 4,
  - klasyfikacja wszystkich pętli z punktu 4.4,
  - zródła pętli zestawów NAS i UTDSP oraz ich format w języku wejściowym narzędzia Petit,
  - kody źródłowe OpenMP badanych pętli z punktu 4.5
- Tools – narzędzia pomocnicze: biblioteka Posix Threads, Omega Calculator i Petit w wersji 2.1, Wolfram .NETLink.

PUBLIKACJE WŁASNE[1]

1. J. Berlińska, M. Pałkowski : Badanie charakterystyki wydajności w środowisku .NET z wykorzystaniem modeli statystycznych, IX Sesja Naukowa Informatyki 2005.
2. Bielecki W., Pałkowski M., Siedlecki K., : Badania efektywności metod wyszukiwania gruboziarnistej rownoległości w pętlach programowych, X Sesja Naukowa Wydziału Informatyki Politechniki Szczecińskiej, 213-228, Szczecin 2005.
3. M. Mościcki, M. Pałkowski, : Zarządzanie procesem testowania na przykładzie Rational TestManager, X Sesja Naukowa Wydziału Informatyki Politechniki Szczecińskiej, 299-312, Szczecin 2005.
4. W. Bielecki, M. Pałkowski : Extracting coarse-grained parallelism in computer simulation applications, Advanced Computer Systems, 13th International Multi-Conference, Vol. II, Szczecin 2006, s. 237-246.
5. W. Bielecki, A Beletska, M. Pałkowski, P. San Pietro : Extracting synchronization-free chains of dependent iterations in non-uniform loops, Advanced Computer Systems, 14th International Multi-Conference, Polish Journal of Environmental Studies, Vol. 16 No. 5B, Międzyzdroje 2007, s. 165-171.
6. Pałkowski M. Rozszerzenie biblioteki Omega Calculator do celów zrównoleglania pętli programowych, Metody Informatyki Stosowanej II/2007, Kwartalnik Komisji Informatyki PAN Oddział Gdańsk, 2008.
7. Bielecki W., Beletska A., Palkowski M., San Pietro P., Extracting synchronization-free trees composed of non-uniform loop operations, Algorithms and Architectures for Parallel Processing, Lecture Notes in Computer Science Volume 5022/2008, Springer Berlin / Heidelberg, 2008, s. 185-195.
8. Bielecki W., Pałkowski M. Using message passing for developing coarse-grained applications in OpenMP, Proceedings of Third International Conference on Software and Data - ICSOFT 2008, Porto, Portugalia 2008, s. 145-153.

[1] sortowanie zgodnie z datą publikacji

9. Pałkowski M. Upraszczanie relacji zależności i zbiorów iteracji w celu podniesienia wydajności generowanego kodu równoległego - Publikacja zgłoszona do Kwartalnika Komisji Informatyki PAN Oddział Gdańsk, Metody Informatyki Stosowanej.
10. Bielecki W., Pałkowski M., Beletska A., Extracting sources of synchronization-free slices by means of the enlarged Omega Library, artykuł wysłany na konferencję IPDPS09, IEEE Symposium on Parallel and Distributed Processing, 2009.

BIBLIOGRAFIA

1. Allen R., Kennedy K. Optimizing compilers for modern architectures: A Dependence- based Approach. Morgan Kaufmann Publishers, Inc., 2001.
2. Almasi G. S., Gottlieb A.. Highly parallel computing. Redwood City, Benjamin/Cummings, 1989.
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, s. 39-50.
4. Amdahl G. On the Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities. Proceedings of the AFIPS Conference, 1967.
5. Anderson J. M., Lam M. S. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. Computer Systems Laboratory, Stanford University, CA 94305.
6. Axford, T. Concurrent programming : fundamental techniques for real-time and parallel software design. Chichester, John Wiley, 1989.
7. Bacon D. F., Graham S. L., Sharp O. J. Compiler Transformations for High-Performance Computing. California : Computer Science Division, University of California, Berkeley, 1993.
8. 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,RNR Technical Report RNR-94-007, 1994.
9. Banerjee U. Unimodular transformations of double loops. Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing. 1990, s. 192-219.
10. Barney, B. Introduction to Parallel Computing, https://computing.llnl.gov/tutorials/parallel_comp/.
11. 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, s. 7-16.
12. Beletska A., Bielecki W., Siedlecki K., San Pietro P. Finding Synchronization-Free Slices of Operations in Arbitrarily Nested Loops. In Proceedings of ICCSA'2008, LNCS.

13. Beletska A., San Pietro P. Extracting Coarse-Grained Parallelism with the Affine Transformation Framework and its Limitations, Electronic Modelling, no. 5, 2006.
14. Berlińska J., Pałkowski M., Badanie charakterystyki wydajności w środowisku .NET z wykorzystaniem modeli statystycznych, PS, M X SNI, 2005.
15. Bielecki W. Esentials of Parallel and distributed Computing, Informa, Szczecin, 2002.
16. W. Bielecki, A Beletska, M. Pałkowski, P. San Pietro : Extracting synchronization-free chains of dependent iterations in non-uniform loops, Advanced Computer Systems, 14th International Multi-Conference, Polish Journal of Environmental Studies, Vol. 16 No. 5B, Międzyzdroje 2007, s. 165-171.
17. Bielecki W., Beletska A., Palkowski M., San Pietro P., Extracting synchronization-free trees composed of non-uniform loop operations, Algorithms and Architectures for Parallel Processing, Lecture Notes in Computer Science Volume 5022/2008, Springer Berlin / Heidelberg, 2008, s. 185-195.
18. Bielecki W., Beletska A., San Pietro P.,Extracting Coarse-Grained Parallelism in Program Loops with the Slicing Framework, In Proceedings of ISPDC, IEEE, 2007.
19. Bielecki W., Pałkowski M., Beletska A., Extracting sources of synchronization-free slices by means of the enlarged Omega Library, artykuł wysłany na konferencję IPDPS09, IEEE Symposium on Parallel and Distributed Processing.
20. Bielecki W., Siedlecki K., Extracting synchronization-free threads in perfectly nested loops using the Omega project software. WSEAS SEPADS, SALZBURG 2005.
21. Bielecki W., Siedlecki K.. Finding Free Schedules for Non-Uniform Loops. Electronic Modeling 2004.
22. Bielecki W., K. Siedlecki K.. Wyszukiwanie początków niezależnych wątków w dowolnie zagnieżdżonych pętlach programowych. PS, M IX SNI, 2004.
23. Bielecki W., Siedlecki K.. Wyznaczanie niezależnych wątków w pętlach idealnie zagnieżdżonych. PS, M VII SNI, 2002.
24. Bielecki W., Pałkowski M. : Extracting coarse-grained parallelism in computer simulation applications, ACS 2006, 13th International Multi-Conference, Vol. II, Szczecin 2006, s. 237-246.

25. Bielecki W. Finding Coarse Grained Parallelism in Computational Geometry Algorithms, Computational Science and Its Applications — ICCSA 2003.
26. Bielecki W. Finding Synchronization-Free Parallelism for Non-uniform Loops, Faculty of Computer Science, In Proceedings of the Computational Science – ICCS 2003, Lecture Notes in Computer Science, Springer, volume 2658, s. 925-934.
27. Bielecki W., Pałkowski M. Using message passing for developing coarse-grained applications in OpenMP, Proceedings of Third International Conference on Software and Data - ICSOFT 2008, Porto, Portugalia 2008, s. 145-153.
28. Bielecki W., Pałkowski M., Siedlecki K.. Badania efektywności metod wyszukiwania gruboziarnistej równoległości w pętlach programowych. PS, M XI SNI, 2006.
29. 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.
30. Butenhof D. R., Programming with POSIX Threads, Addison-Wesley, 1997.
31. Chapman B., Jost G., Pas R. Using OpenMP, Portable Shared Memory Parallel Programming, MIT Press, 2008.
32. Collard J. F., Feautrier P., Risset T., Construction of do loops from systems of affine constraints, Parallel Processing Letters, Tom. 5, 1995, s. 421-436.
33. Crichlow J. M., An introduction to distributed and parallel computing. New York, Prentice Hall, 1988.
34. Dagum L., Menon R. OpenMP: An Industry-Standard API for Shared-Memory Programming. SILICON GRAPHICS INC.
35. Darte A., Robert Y., Vivien F. Scheduling and Automatic Parallelization, Birkhäuser Boston, 2000.
36. 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.
37. El-Rewini H., Abd-El-Barr M. Advanced computer architecture and parallel processing., a john wiley & sons, inc publication, 2005.

38. Feautrier P. Some efficient solutions to the affine scheduling problem, part i, one dimensional time. International Journal of Parallel Programming 21. 1992, s. 313-348.
39. Feautrier P. Some efficient solutions to the affine scheduling problem, part ii, multidimensional time. International Journal of Parallel Programming 21. 1992, s. 389- 420.
40. Foster I. Designing and Building Parallel Programs. http://www-unix.mcs.anl.gov/dbpp/.
41. Gavalda R., Ayguade E., Torres J. Obtaining Synchronization-Free Code with Maximum Parallelism, 1996.
42. GCC, the GNU Compiler Collection, http://gcc.gnu.org/.
43. Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R., Sunderam V.S., PVM: Parallel Virtual Machine Users' Guide and Tutorial for Networked Parallel Computing, MIT Press, 1994.
44. Geist G. A., Kohl J. A., Papadopoulos P. M. PVM and MPI: a Comparison of Features. 1996.
45. Grama A., Gupta A., Karypis G., Kumar V. An Introduction to Parallel Computing.Addison Wesley, 2003.
46. 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.
47. Gustafson J. Reevaluation Amdahl's law. Communications of the ACM. 1988, Tom 31, 5.
48. http://www.nas.nasa.gov/Resources/Software/npb.html
49. Heroux M. A., Raghavan P., Simon H. D., Parallel Processing for Scientific Computing, Cambridge University Press, 2007.
50. Jin H., Frumkin M., Yan J. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. October 1999. NAS Technical Report NAS-99-011.
51. Juhasz Z., Kacsuk P., Kranzlmuller D. Distributed and Parallel Systems - cluster and grid computing. Springer Science, Business Media, Inc. 2005.

52. 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.
53. Kelly W., Pugh W. A Framework for Unifying Reordering Transformations. Univ. Of Maryland, 1993.
54. 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.
55. Kelly W., Maslov V., Pugh W., Rosser E., Shpeisman T., Wonnacott D. The omega library interface guide. University of Maryland, 1995. Technical Report CS-TR-3445.
56. Kelly W., Maslov V., Pugh W., Rosser E., Shpeisman T., Wonnacott D. The Omega Library, Version 1.1.0, Interface Guide. November 18, 1996.
57. 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, s. 579-598
58. Kumar V., Introduction to parallel computing : design and analysis of algorithms, Redwood City, Benjamin/Cummings, 1994.
59. Kumar V., Gupta A. Analyzing scalability of parallel algorithms and architectures. Journal of Parallel and Distributed Computing. 22(3), 379-391, 1994.
60. Kunz Robert C, Performance bottlenecks on large-scale shared-memory multiprocessors, December 2004.
61. Lee C., UTDSP Benchmark Suite, http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html.
62. Lewis B., Berg D. J., PThreads Primer, A Guide to Multithreaded Programming, Sun Microsystem Press, 1996.
63. Lewis T. G. Foundations of Parallel Programming, A Machine-Independent Approach. Los Alamitos, California : IEEE Computer Society Press, 1994.
64. Lilja D. J. A Multiprocessor Architecture Combining Fine-Grained and Coarse-Grained Parallelism Strategies, Parallel Computing, Vol. 20, No.5, 1994, s. 729-751.
65. 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.

66. 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.
67. 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.
68. Moldovan D., Parallel Processing, From Applications to Systems, Morgan Kaufmann Publishers, Inc., 1993.
69. Morrison R. S. Cluster Computing: Architectures, Operating Systems, Parallel Processing & Programming Languages. 1998 – 2003.
70. Mościcki M., Pałkowski M., Zarządzanie procesem testowania na przykładzie Rational TestManager, PS, M XI SNI, 2006.
71. Muchnick S. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997.
72. Nichols B., Pthreads Programming. s.l. : O'Reilly and Associates, 1996.
73. OpenMP Application Program Interface, Version 3.0. 2008, http://www.openmp.org.
74. Pacheco P. S. Parallel Programming with MPI. Morgan Kaufmann, 1997.
75. Pałkowski M., Implementacja prezentowanych w niniejszej pracy algorytmów - narzędzie, kod źródłowy, dokumentacja. http://detox.wi.ps.pl/SFS_Project .
76. Pałkowski M., Rozszerzenie biblioteki omega calculator do celów zrównoleglania pętli programowych, Metody Informatyki Stosowanej II/2007, Kwartalnik Komisji Informatyki PAN Oddział Gdańsk, 2008.
77. Pałkowski M., Upraszczanie relacji zależności i zbiorów iteracji w celu podniesienia wydajności generowanego kodu równoległego. Publikacja planowana do Kwartalnika Politechniki Szczecińskiej, 2008.
78. Park I. Parallel programming methodology and environment for the shared memory programming model, Purdue University, 2000. Thesis.
79. Peng, Sean Hsien-en. UTDSP: A VLIW Programmable DSP Processor. Department of Electrical and Computer Engineering, University of Toronto, 1999.
80. Pugh W., Rosser E. Iteration Space Slicing and Its Application to Communication Optimization. Proceedings of the International Conference on Supercomputing. 1997, s. 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. Rajasekaran S., Reif J., Handbook of Parallel Computing Models, Algorithms and Applications, Chapman & Hall/CRC, 2007.
84. Sahnin S., Thanvantri V., Parallel Computing, Performance Metrics and Models. Computer & Information Sciences Department University of Florida, 1995.
85. Siedlecki K., Algorytmy wyszukiwania drobno- i gruboziarnistej równoległości w pętlach programowych z zależnościami afinicznymi - praca doktorska. Szczecin, Wydział Informatyki, Politechnika Szczecińska, 2008.
86. Silberschatz A., Galvin P. B., Podstawy systemów operacyjnych, Wydawnictwo Naukowo-Techniczne, Warszawa 2000.
87. 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.
88. Van Roy P., Haridi S., Programowanie. Koncepcje, techniki i modele, Helion 2005.
89. Vasilache, N., Bastoul, C., and Cohen A. Polyhedral code generation in the real world. In Proceedings of the International Conference on Compiler Construction (ETAPS CC'06) LNCS, pp 185-201, Springer-Verlag, Vienna, Austria. 2006.
90. Waheed A., Yan J. Parallelization of NAS Benchmarks for Shared Memory Multiprocessors. March 1998. NAS Technical Report NAS-98-010.
91. Wolf M. E. High performance compilers for parallel computing, Redwood City, Addison-Wesley, 1996.
92. Wolf M. E. Improving locality and parallelism in nested loops. 1992. Ph.D. Dissertation CSL-TR-92-538.
93. Wolf M. E., Lam M. S. A loop transformation theory and an algorithm to maximize parallelism. Transactions on Parallel and Distributed Systems,Tom 2(4), 1994, s. 452-470.
94. Wolfram Research, Pakiet Mathematica, http://www.wolfram.com/.

95. Xue J. Unimodular Transformations of Non-Perfectly Nested Loops. Department of Mathematics, Statistics and Computing Science, University of New England, 1997.