Szczecin, dnia 25 stycznia 2018r.

dr inż. Marek Pałkowski
Katedra Inżynierii Oprogramowania
Wydział Informatyki
Zachodniopomorski Uniwersytet Technologiczny w Szczecinie
tel. 692 079 195
e-mail: mpalkowski@wi.zut.edu.pl

# AUTOREFERAT

1. Imię i Nazwisko.

**Marek Pałkowski**

2. Posiadane stopnie naukowe:

6 lipca 2004 – tytuł *magistra inżyniera* uzyskany na Wydziale Informatyki Politechniki Szczecińskiej, kierunek informatyka, w zakresie programowanie równoległe i rozproszone

20 stycznia 2009 – *stopień naukowy doktora* nauk technicznych w dyscyplinie informatyka nadany uchwałą Rady Wydziału Informatyki Zachodniopomorskiego Uniwersytetu Technologicznego w Szczecinie, tytuł rozprawy „*Algorytmy zwiększające ekstrakcję równoległości w pętlach programowych*"

3. Dotychczasowe zatrudnienie w jednostkach naukowych

od 01.04.2009 do 31.12.2009 – *asystent* w Katedrze Inżynierii Oprogramowania na Wydziale Informatyki Informatyki Zachodniopomorskiego Uniwersytetu Technologicznego

od 01.01.2010 – *adiunkt* w Katedrze Inżynierii Oprogramowania na Wydziale Informatyki Informatyki Zachodniopomorskiego Uniwersytetu Technologicznego. Od 01.10.2014 jestem kierownikiem Zakładu Technik Programowania.

4. Wskazanie osiągnięcia wynikającego z art. 16 ust. 2 ustawy z dnia 14 marca 2003 r. o stopniach naukowych i tytule naukowym oraz o stopniach i tytule w zakresie sztuki (Dz. U. 2016 r. poz. 882 ze zm. w Dz. U. z 2016 r. poz. 1311.):

a) Tytuł osiągnięcia naukowego/artystycznego,

**„Techniki tworzenia kompilatorów automatycznie optymalizujących kod programowy w oparciu o tranzytywne domknięcie grafu zależności"**

*(jednotematyczny cykl publikacji)*

b) Publikacje wchodzące w skład osiągnięcia naukowego (w kolejności chronologicznej)

1. Anna Beletska, Włodzimierz Bielecki, Albert Cohen, Marek Pałkowski, Krzysztof Siedlecki, *Coarse-grained loop parallelization: Iteration Space Slicing vs affine transformations,* 2011, Parallel Computing, 37(8), s. 479-497. **IF:** 1.311, **Punkty MNiSW:** 30, **Udział:** 30%.
2. Włodzimierz Bielecki, Marek Pałkowski, Tomasz Klimek, *Free scheduling for statement instances of parameterized arbitrarily nested affine loops,* 2012, Parallel Computing 38(9), s. 518-532. **IF:** 1.214, **Punkty MNiSW:** 30, **Udział:** 33,3%.
3. Marek Pałkowski, Włodzimierz Bielecki *TRACO: Source-to-Source Parallelizing Compiler*, 2016, Computing and Informatics 35(6), s. 1277-1306. **IF:** 0,488, **Punkty MNiSW:** 15, **Udział:** 70%.
4. Włodzimierz Bielecki, Marek Pałkowski, *Tiling arbitrarily nested loops by means of the transitive closure of dependence graphs*, 2016, International Journal of Applied Mathematics and Computer Science 26(4), s. 919-939. **IF:** 1.420, **Punkty MNiSW:** 25, **Udział:** 50%.
5. Tomasz Klimek, Marek Pałkowski, Włodzimierz Bielecki, 2016, *Synchronization-Free Automatic Parallelization for Arbitrarily Nested Affine Loops*, wydawca: IEEE, International Symposium: Computer Architecture and High Performance Computing Workshops (SBAC-PADW), Los Angeles, CA, USA. **Punkty MNiSW:** 15 (Web of Science), **Udział:** 33,3%.



Pałki

6. Marek Pałkowski, Włodzimierz Bielecki, 2016, *An Iteration Space Visualizer for Polyhedral Loop Transformations in Numerical Programming,* Annals of Computer Science and Information Systems, Vol. 8, str. 705-708, wydawca: IEEE, (FedCSiS 2016), Gdańsk. **Punkty MNiSW:** 15 (Web of Science), **Udział:** 70%.
7. Marek Pałkowski, Włodzimierz Bielecki *Parallel tiled code generation with loop permutation within tiles,* 2017, Computing and Informatics 36(6), s. 1261-1282. **IF:** 0,488, **Punkty MNiSW:** 15, **Udział:** 60%.
8. Marek Pałkowski, Włodzimierz Bielecki, *Parallel tiled Nussinov RNA folding loop nest generated using both dependence graph transitive closure and loop skewing*, 2017, BMC Bioinformatics 18:290, s. 1-10 (open-access). **IF:** 2,435, **Punkty MNiSW:** 35, **Udział:** 70%.
9. Włodzimierz Bielecki, Marek Pałkowski, Piotr Skotnicki *Generation of parallel synchronization-free tiled code*, 2017, Computing, Springer Vienna, ISSN=1436-5057, DOI: 10.1007/s00607-017-0576-3. **IF:** 1,589, **Punkty MNiSW:** 25, **Udział:** 33,3%
10. Marek Pałkowski, Włodzimierz Bielecki, *Tuning Iteration Space Slicing based tiled multi-core code implementing Nussinov's RNA folding*, 2018, BMC Bioinformatics, 19:12, s. 1-12 (open-access) **IF:** 2,435, **Punkty MNiSW:** 35, **Udział:** 70%.

c) Omówienie celów naukowych oraz osiągniętych wyników wraz z omówieniem ich ewentualnego wykorzystania.

Niemal dwie dekady temu rozwój prędkości procesorów komputerowych został znacznie zahamowany poprzez ograniczenia fizyczne takie jak wykładniczy wzrost liczby tranzystorów, problemy z emisją ciepła i ciągła miniaturyzacja. Sprawiło to, że producenci sprzętu zintensyfikowali wysiłek w celu konstruowania wydajnych procesorów wielordzeniowych. Wkrótce potem pojawiły się kolejne komputery równoległe takie jak karty graficzne do ogólnych zadań obliczeniowych i wielordzeniowe koprocesory. Powstało wiele ośrodków obliczeniowych zajmujących się przetwarzaniem równoległym różnorodnych symulacji komputerowych, w tym krajowe centra badawcze takie jak w Krakowie (Cyfronet), w Poznaniu (PCSS) czy w Gdańsku (CI Task)[1]. W wyniku tego duże zainteresowanie badaczy i inżynierów skupione zostało na konstruowaniu optymalnego kodu programowego wykorzystującego dostępną moc maszyny. Pod pojęciem optymalizacji kodu rozumiane jest jako jego zrównoleglenie oraz poprawy jego lokalności. Szczególnym zadaniem stało się

[1] Wymienione ośrodki znajdują się w pierwszej dwusetce listy top500.org.



Pałki

zbudowanie kompilatorów automatycznie przekładających oryginalny kod na szybszy i zgodny z oryginałem bez znaczącego udziału programisty.

Najwięcej obliczeń znajduje się w konstrukcjach iteracyjnych, stąd w takich kompilatorach zaszytych jest wiele sprawnych algorytmów przetwarzających pętle programowe. Automatyczne transformacje pętli nie są zadaniami trywialnymi ze względu na różnorodność kodu programowego. W technikach tych wykorzystywane są różne aparaty matematyczne, a ich skuteczność ocenia się poprzez takie mierniki jak zakres stosowania, przyspieszenie (skrócenie czasu obliczeń), skalowalność (zachowanie przyspieszenia wobec zmiany rozmiaru problemów lub liczby wątków) czy lokalność kodu (większe wykorzystanie pamięci lokalnej np. kieszeniowej wobec operacyjnej).

Najpopularniejszym i bardzo skutecznym rozwiązaniem jest platforma transformacji afinicznych (ATF – ang. *affine transformation framework*) stosowana do przekształceń pętli programowych. Afiniczne transformacje są obecnie najbardziej rozwiniętą techniką ekstrakcji równoległości [Feautrier92_1, Feautrier92_2, Lim94, Bondhugula08, Verdoolaege13, Bondhugula16]. Łączą one dużą klasę transformacji, np. przeplot, odwrócenie, łączenie, permutację pętli oraz pozwalają na uzyskanie kodu równoległego dla maszyn z pamięcią dzieloną i systemów rozproszonych. Zwiększają one także lokalność kodu i redukują zapotrzebowanie na pamięć. Główną ideą ATF jest afiniczne przekształcenie pętli w taki sposób, że zależne od siebie instancje instrukcji składają się na tą samą partycję w przestrzeni iteracji lub należą do różnych partycji czasu, z których każda zawiera niezależne instancje. Istotną zaletą ATF jest zatem zastosowanie jednej metody do wielu transformacji.

Klasyczne transformacje ATF nie pozwalają jednak w ogólnym przypadku pętli programowej na ekstrakcję równoległości czy poprawę lokalności, w szczególności, gdy graf zależności jest nieregularny [1,8, Mullapudi14, Wonnacott13][2]. Innymi słowy, brakuje funkcji afinicznej do przedstawienia istniejącej równoległości czy istniejącego podziału na bloki w przestrzeni iteracji pętli programowej, pomimo tego, że istnieje równoległość w takich kodach.

W kompilatorach transformujących pętle programowe dominujący jest model wielościenny (ang. *polyhedral model*). Przestrzeń iteracji, granice pętli, odwołanie do tabel czy wyrażenia warunkowe opisane są poprzez wyrażenia afiniczne. Model wielościenny stosowany pozwala na realizację 3 elementów:

- analiza zależności danych w pętlach programowych,
- transformacje pętli programowych,
- generowanie kodu.

Niniejszy cykl publikacji przedstawia szereg autorskich algorytmów przekształcających pętle programowe za pomocą wyznaczenia tranzytywnego domknięcia w grafach zależności bez konieczności znalezienia funkcji afinicznych. Algorytmy te są rozwinięciem teorii podziału przestrzeni iteracji na fragmenty kodu, poszerzają one dotychczasowy zakres stosowalności oraz zostały zaimplementowane w autorskim kompilatorze. Narzędzie to realizuje wszystkie trzy elementy modelu wielościennego. Analiza zależności i generowanie kodu wykonywane są za pomocą takich samych modułów jak w pokrewnych kompilatorach. Różnicą i innowacją są zaproponowane techniki przekształcania (transformacje) pętli programowych. Charakterystyczną i łączącą cechą wszystkich

[2] Odwołania do pozycji z cyklu publikacji oznaczone są liczbowo, z kolei odwołania z bibliografii umieszczonej na końcu autoreferatu oznaczone są nazwiskiem pierwszego autora i rokiem.

algorytmów jest wyznaczanie tranzytywnego domknięcia relacji opisujących wszystkie zależności w kodzie. Nadrzędnym celem badań jest przekształcenie kodu i wykazania jego możliwości przyspieszenia na nowoczesnych platformach wielordzeniowych w przypadkach, gdy pokrewne rozwiązania, w tym techniki ATF zawodzą lub generują kod wykazujący mniejszą akceleracje obliczeń. Przedstawiony cykl publikacji stanowi kontynuację i znaczący rozwój badań rozpoczętych w mojej dysertacji doktorskiej nad algorytmami zrównoleglającymi kod programowy.

Znaczący wkład autorski w dyscyplinie Informatyka polega na rozwoju teorii tworzenia kompilatorów optymalizujących z zastosowaniem tranzytywnego domknięcia grafu zależności pozwalających zwiększyć zakres stosowalności istniejących pokrewnych narzędzi, a także implementacja zaproponowanych rozwiązań w ogólnie dostępnym kompilatorze TRACO oraz wykazanie skuteczności zaimplementowanych algorytmów dla aplikacji rzeczywistych i benchmarków realizujących obliczenia numeryczne i symulacje w takich dziedzinach jak dynamiczne programowanie, bioinformatyka, obliczeniowa mechanika płynów czy algebra liniowa.

*Reprezentacja zależności i tranzytywne domknięcie grafu zależności*

W prezentowanych algorytmach podstawowym aparatem matematycznym jest arytmetyka Presburgera, która jest układem aksjomatycznym liczb naturalnych z dodawaniem, znanym także jako arytmetyka Peano bez mnożenia. Język arytmetyki zawiera binarne wartości 0 i 1 oraz dodawanie określane przez binarną funkcję + [Fisch74].

W opisywanych publikacjach przyjęto reprezentację zależności poprzez relacje, których ograniczenia zbudowano z formuł Presburgera zaproponowaną w pracy [Pugh93]. Relacje określające zależności łączą pary zbiorów instrukcji pętli pomiędzy sobą za pomocą symbolu $\rightarrow$. Zbiory krotek opisują przestrzeń iteracji, zbiory instrukcji lub bloki (ang. tiles). Taka reprezentacja jest matematycznie abstrakcyjna i niezależna od docelowej architektury czy języka programowania. Ponadto, relacje służą jako reprezentacja sparametryzowanych grafów zależności (pokrewne rozwiązania stosują tożsamą postać macierzową).

W celu analizy i transformacji programów używane są zbiory i relacje, których liniowe ograniczenia nad zmiennymi całkowitymi połączone są za pomocą logicznych operatorów negacji ($\neg$), koniunkcji ($\wedge$) i alternatywy ($\vee$) lub projekcji kwantyfikatora szczegółowego ($\exists$). W poszczególnych etapach zaproponowanych algorytmów wykonywane są operacje na zbiorach i relacjach typu część wspólna ($\cap$), unia ($\cup$), różnica(-), dziedzina, zakres, kompozycja ($\circ$), aplikacja relacji na zbiorze. Fundamentalną operacją jest tranzytywne domknięcie relacji zależności w pętlach programowych.

Domknięcie przechodnie (tranzytywne) jest operacją wywodzącą się z teorii grafów i wyznacza wszystkie tranzytywne ścieżki pomiędzy wierzchołkami w grafie skierowanym. Relacja jest matematyczną reprezentacją skierowanego grafu, zatem domkniecie relacji opisuje domknięcie grafu i odwrotnie. Dokładne tranzytywne domknięcie relacji jest definiowane jako nieskończona (sparametryzowana) liczba unii potęgi relacji [Kelly96]. Potęga stopnia $k$ relacji jest to $k$-krotne złożenie (kompozycja) tej relacji [Verdoolaege10]. Istotną cechą jest wyznaczenie ścieżek dla relacji z parametrami, co oznacza w praktyce transformacje dla pętli programowej, której zmienne parametryczne nie są znane w czasie kompilacji.



Pallli

Zespół profesora Williama Pugh (University of Maryland) jako pierwszy zaproponował algorytm do wyznaczania domknięcia dla sparametryzowanego grafu zależności [Kelly96]. Nie istnieje jednak wciąż uniwersalny algorytm obliczania domknięcia przechodniego w sposób dokładny dla dowolnej relacji sparametryzowanej. Niemniej obecnie zaproponowane algorytmy [Verdoolaege10, Bielecki10, Bielecki14] pozwalają na wyznaczenie dokładnego domknięcia grafu zależności dla szerokiego zakresu pętli programowych, a co za tym idzie na transformacje pętli programowych bez zastosowania funkcji afinicznych. Ponadto badania nad udoskonaleniem wyznaczenia domknięcia lub jego aproksymacji są wciąż aktualnym tematem [Verdoolaege11, Feautrier12, Feautrier15]. Algorytmy tego zadania są poza zakresem niniejszego cyklu publikacji.

*Ograniczenia transformacji afinicznych*

Techniki ATF są obecnie uznawane za najbardziej efektywne metody do wyznaczania automatycznej optymalizacji kodu oraz zaimplementowane są w takich skutecznych i pokrewnych kompilatorach jak Pluto [Bondhugula08], Pluto+ [Bondhugula16], PTile [Baskaran10], POCC [Park11] czy PPCG [Verdoolaege13]. Ten sam aparat matematyczny wykorzystywany jest do wyznaczenia równoległości jak i poprawy lokalności kodu poprzez blokowanie i permutację pętli programowych. Model ATF realizowany jest poprzez złożoną sekwencję transformacji zmieniających kolejność wykonania instancji instrukcji w pętlach programowych. Techniki ATF gwarantują poprawność wykonania kodu poprzez honorowanie zależności, tj. wykonanie obliczeń skojarzonych z początkiem zależności przed obliczeniami skojarzonymi z końcem dla każdej zależności. Wynikiem zastosowania funkcji afinicznego mapowania są informacje o numerze przyporządkowanego procesora, czasie logicznym oraz nowej iteracji dla wszystkich instancji instrukcji, natomiast wyznaczone funkcje afiniczne są danymi wejściowymi do generatora kodu.

Ograniczeniem technik ATF dla ogólnego przypadku jest możliwość braku rozwiązania utworzonego układu afinicznych równości i nierówności, na podstawie którego wyznaczana jest poprawna transformacja afiniczna. Czynnikami ograniczającymi tę metodę są głównie niejednorodne zależności, występowanie negatywnych współczynników w wektorach dystansu, czy też cykle w grafie zależności pomiędzy blokami. Wady klasycznych algorytmów ATF opisywane są w publikacjach między innymi [Feautrier92_1, Feautrier92_2, Bondhugula16, Verdoolaege13, Wonnacott13] oraz w pracach z prezentowanego cyklu publikacji.

W artykule [1] wykazano, że transformacje afiniczne nie dokonują ekstrakcji równoległości w postaci niezależnych wątków dla zadanych pętli programowych. Pokazano też przykłady, gdy ATF nie wykrywa także wszystkich niezależnych fragmentów kodu. Transformacje te nie dokonują także ekstrakcji równoległości bez synchronizacji zawartej w poddomenie dziedziny pętli oraz w przypadku, gdy niezależne fragmenty kodu posiadają wiele początków (punktów niebędącymi końcami zależności).

W artykule [2] wykazane zostało, że transformacje ATF znajdują równoległość z synchronizacją z większą liczbą punktów synchronizacji, tzn. niemożliwa jest ekstrakcja maksymalnej równoległości (w zadanym punkcie czasu brak jest wszystkich niezależnych instrukcji gotowych do wykonania).

W artykule [4] pokazano przykłady, dla których algorytmy ATF nie pozwalają na wyznaczenie sposobu blokowania pętli programowych. W artykule [8] z kolei pokazano

aplikacje dynamicznego programowania (DP), dla których ATF uniemożliwia blokowanie wewnętrznych gniazd pętli programowych kluczowych do uzyskania zadowalającego przyspieszenia obliczeń. Ograniczenia te wiążą się z nieregularnymi zależnościami i brakiem możliwości wygenerowania takich bloków, pomiędzy którymi nie istnieją cykle w aplikacjach dynamicznego programowania.

Porównania ATF z proponowanymi algorytmami przedstawiono w sposób teoretyczny, jak i praktyczny, tj. analizując wygenerowane kody za pomocą autorskiego kompilatora i pokrewnych narzędzi bazujących na ATF.

*Wyznaczanie równoległości za pomocą tranzytywnych ścieżek w grafie zależności*

Podział przestrzeni iteracji (ang. *iteration space slicing*) został przedstawiony przez Pugh i Rossera [Pugh97] jako rozszerzenie ekstrakcji fragmentów kodu (ang. slicing) zaproponowanej przez badacza Weisera [Weiser84]. Podział ten wyodrębnia niezależne fragmenty kodu lub wymagających stosunkowo rzadkiej synchronizacji.

Pugh i Rosser zastosowali podział przestrzeni iteracji do optymalizacji komunikacji międzyprocesowej. Pokazali w szczególności jak ekstrakcja fragmentów pozwala na łączenie (fuzję) pętli programowych, tolerancję opóźnień w komunikacji i łączenie komunikatów. Jednakże autorzy nie zaprezentowali podziału przestrzeni iteracji w sposób automatyczny wraz z blokowaniem pętli programowych za pomocą tranzytywnego domknięcia. Przy opracowywaniu algorytmów zaprezentowanych w niniejszym cyklu skorzystano z analizy zależności zaproponowanej przez badaczy Pugh i Wonnacota oraz zaimplementowanego w narzędziu Petit [Pugh93], gdzie zależności są prezentowane przez relacje. Relacje zależności są danymi wejściowymi do prezentowanych algorytmów podziału przestrzeni iteracji i za ich pomocą można wyznaczyć transformację i/lub sprawdzić jej poprawność.

Kluczową operacją jest obliczenie tranzytywnego domknięcia unii wszystkich relacji zależności. Jeżeli nie jest możliwe znalezienie dokładnego domknięcia z wszystkimi tranzytywnymi ścieżkami, można dokonać jeszcze aproksymacji. Oznacza to dodanie fałszywych ścieżek (zależności) i znalezienie prawidłowego, lecz mniej optymalnego rozwiązania [Verdoolaege10].

Niezależny fragment kodu (ang. *slice*) to maksymalny podgraf w grafie zależności, którego nie łączy żadna nieskierowana ścieżka z innym fragmentem. Ponieważ graf zależności jest grafem skierowanym, dokonuje się inwersji wszystkich relacji i połączenia tych dwóch zbiorów za pomocą unii. W ten sposób otrzymujemy nieskierowany graf zależności [Pugh97].

Topologią fragmentu może być łańcuch, drzewo lub graf ogólny. Pugh i Rosser nie zaprezentowali jak automatycznie wyznaczyć punkty reprezentatywne (ich leksykograficznie pierwsze instancje instrukcji) fragmentu i ich wszystkie tranzytywnie zależne instancje instrukcji oraz sposobu generowania kodu na podstawie wyznaczonych zbiorów. Pomimo tego, ich zespół zaproponował wiele użytecznych modeli do wyznaczania transformacji i optymalizacji kodu za pomocą arytmetyki Presburgera i biblioteki Omega Calculator [Kelly95].

W artykule [1] zaprezentowano jak za pomocą operacji na relacjach i zbiorach wyznaczyć fragmenty kodu i wygenerować równoległy kod bez synchronizacji.

Jeśli graf zależności pętli reprezentuje tylko jeden niezależny fragment kodu, wówczas możliwe jest obliczenie harmonogramowania iteracji w czasie (ang. *schedule*), np. za pomocą

transformacji afinicznych. Badacz Uday Bounghdula (Ohio State University) opracował kompilator Pluto, który dokonuje zrównoleglenia z synchronizacją dla szerokiego spektrum pętli programowych za pomocą transformacji afinicznych, potokowości (ang. *pipelining*) oraz podstawowych transformacji unimodularnych takie jak przekoszenie pętli. W klasycznych algorytmach zaimplementowanych w Pluto istnieją ograniczenia w wyznaczaniu harmonogramowania swobodnego [Darte00] (ang. *free-schedule*), które pozwala na znajdowanie maksymalnej drobnoziarnistej równoległości w pętlach programowych. Techniki oparte na transformacjach linowych lub afinicznych [Allen01, Wolfe95, Bacon93, Banerjee93, Feautrier92_1, Feautrier92_2] nie pozwalają na znajdowanie takiego harmonogramowania swobodnego w ogólnym przypadku [2].

Wonnacott w pracy [Wonnacott13] opisał, że brakuje maksymalnej równoległości podczas uruchomienia kodu z optymalizacją pamięci podręcznej w rozwiązaniach Bounghduli. W ostatnim czasie stosowalność algorytmów kompilatora Pluto zostały poszerzone dla regularnych programów typu „*stencil*" [Bondhugula17].

W pracy [Bielecki03] wykazano, że istnieje szeroka gama pętli programowych, w których za pomocą arytmetyki Presburgera można wyznaczyć równoległość z synchronizacją, pomimo braku możliwości wyznaczenia afinicznego mapowania. Ograniczeniem algorytmów przedstawionych w tej pracy jest jednak to, że zakres ich stosowalności dotyczy tylko pętli niesparametryzowanych.

W artykule [2] niniejszego cyklu zaproponowana została metoda wyznaczenia harmonogramowania swobodnego dla pętli sparametryzowanych z zależnościami regularnymi i nieregularnymi. Z wykorzystaniem kompozycji oraz potęgi relacji zależności pokazano, że możliwe jest wyznaczenie maksymalnej równoległości (jednoczesnego uruchomienia równoległych instrukcji, gdy tylko ich wszystkie operandy są dostępne) w pętlach programowych dla których klasyczne transformacje zawodzą.

*Zwiększanie lokalności kodu poprzez blokowanie pętli*

Blokowanie pętli programowych (ang. *tiling* - kafelkowanie) to grupowanie zbioru instancji instrukcji w mniejsze bloki w celu zwiększenia ponownego użycia pamięci kieszeniowej. Jest to jedna z podstawowych transformacji poprawiająca lokalność programu. W przetwarzaniu równoległym bloki są niepodzielnymi makro-instrukcjami i zwiększają gruboziarnistość kodu. Początkowe prace nad blokowaniem [Irigoin88, Wolf91] oraz nowe i zautomatyzowane metody [Bondhugula08, Griebl04, Wonnacott13] oparte są o techniki klasyczne w modelu wielościennym (Pluto) oraz inne (modele rzadkie [Strout04], nieliniowe modele wielościenne [Kim09] czy o podział przestrzeni iteracji [Pugh97]). Najbardziej zaawansowane metody klasyczne oparte są o transformacje afiniczne oraz zaimplementowane w kompilatorze Pluto. W narzędziu Pluto+ [Bondhugula16], połączono ATF z technikami podziału zbioru indeksów [Griebl00], aby poszerzyć stosowalność o pętle z cykliczną zmiennością dziedziny.

Jednakże klasyczne transformacje afiniczne nie są bez wad. W pracach [Mullapudi14, Wonnacott15] wskazano, że nie dokonują one blokowania w taki sposób ze brak jest cykli pomiędzy nimi (takie cykle uniemożliwiają znalezienie szeregowania bloków [Feautrier92_1, Feautrier92_2]) oraz istnieją problemy ze skalowalnością obliczeń (brak równoległości lub mały stopień równoległości dla pierwszych partycji czasu). Drugie ograniczenie zostało rozwiązane poprzez zmianę kształtu bloku z prostokątnego na diamentowy czy trapezoidalny

[Grosser14]. Techniki te przetestowano na licznych benchmarkach, w których większość należy do typu „stencil". Są to programy, które do obliczenia elementu tablicy w zadanym czasie (iteracji) wymagana jest tylko stała liczba sąsiednich elementów np. algorytmy Gauss-Seidel'a czy Jacobi'ego z algebry liniowej [Pouchet15]. Rozwiązania te są jednak bezużyteczne, gdy występuje nieregularność w grafie zależności np., w których do obliczenia określonego elementu potrzebny jest cały wiersz i kolumna poprzednio obliczonych wartości elementów tej tablicy. Pojawiają się wówczas zależności niejednorodne oraz brak jest możliwości wygenerowania takich bloków pomiędzy którymi nie istnieją cykle zależności, co ogranicza takie podejścia jak blokowanie diamentowe bazujące na ATF [Bondhugula17]. Przykładem takich aplikacji są algorytmy dynamicznego programowania z zakresu zagadnień optymalizacji [8].

W pracy [4] pokazane zostało jak skonstruować blokowanie pętli programowych za pomocą tranzytywnego domknięcia grafów zależności. Jeżeli tylko możliwe jest obliczenie domknięcia unii relacji zależności, kompilator wygeneruje kod, w którym nie będzie cykli w grafie zależności pomiędzy blokami docelowymi. W tym celu poszczególne instrukcje prostokątnych bloków zostaje odpowiednio przesunięte do późniejszych leksykograficznie bloków (zwanych też kaflami). W pracy [4] udowodniono poprawność tego podejścia, natomiast w pracy [Palkowski15] zaprezentowano jak zrównoleglić zablokowany kod za pomocą technik opartych na podziale przestrzeni iteracji [1,2] oraz za pomocą innych klasycznych technik jak transformacje unimodularne i afiniczne [8].

Ponadto pokazano efektywność metody [4], gdy niemożliwe jest wyznaczenie funkcji afinicznej dokonującej blokowania pętli lub blokowania wszystkich gniazd pętli (np. w aplikacjach dynamicznego programowania), nie można zastosować afinicznego blokowania z kaflami o kształcie rombu, trapezu i sześciokąta oraz niemożliwe jest zastosowanie podstawowych technik, np. permutacji pętli. Zaprezentowana technika łączy w sobie cechy modelu wielościennego i podziału przestrzeni iteracji. Algorytmy zostały zaimplementowane w autorskim kompilatorze [3].

*Kontrybucje techniczne*

Implementacja algorytmów została zrealizowana w autorskim kompilatorze TRACO (akronim od słowa TRAnsitive ClOsure) [Traco2017]. Bez automatycznego narzędzia niemożliwe jest analiza i wygenerowanie kodu równoległego z blokowaniem już dla niewielkich programów. Automatyzacja transformacji jest zatem konieczna ze względu na duży stopień skomplikowania przekształceń sparametryzowanych i dowolnie zagnieżdżonych pętli programowych.

TRACO (licencja GPL) jest kompilatorem automatycznie przekładających kompilowalne źródła wejściowe na zoptymalizowane źródła wyjściowe (ang, source-to-source). Stąd w kompilatorze znajdują się odpowiednie moduły analizy strukturalnej kodu, czyli ekstrakcji pętli programowych (przetwarzania wstępnego) i przetwarzania końcowego przystosowującego kod do kompilowalnej postaci. Pomiędzy nimi znajdują się bloki kompilacji definiowane przez etapy modelu wielościennego, tj. analiza zależności, transformacje pętli i wygenerowanie kodu. Taka modułowa budowa występuje także w pokrewnych kompilatorach optymalizujących, np. Pluto i Pluto+.

Do analizy zależności stosowane są narzędzia Petit [Pugh93] i Pet [Verdoolaege12]. W celu wykonania operacji na zbiorach i relacjach z zakresu arytmetyki Presburgera



Palki

wykorzystano także nowoczesną i ciągle rozwijaną bibliotekę Integer Set Library (ISL) autora Svena Verdoolaege'a [Verdoolaege10]. Do generowania kodu wykorzystywany jest także ISL oraz narzędzie Cedrica Bastoula, Cloog [Bastoul04]. Te same biblioteki używane są w pokrewnych kompilatorach np. Pluto, Pluto+ czy PCGG. Wyróżniającym elementem TRACO jest zatem moduł transformacji.

TRACO ma ponad 100 tysięcy linii kodu (bez komponentów trzecich i licznych przykładów), które zostały napisane w językach C/C++ i Python w zdecydowanej większości przez autora niniejszego autoreferatu. TRACO dedykowany jest do systemów Linux. Struktura i budowa kompilatora została dokładniej opisana w pracy [3]. TRACO jest narzędziem ogólnodostępnym (traco.sourceforge.net) wraz z kodami źródłowymi.

Najbardziej porównywalnym narzędziem do TRACO, pod względem możliwości optymalizacji kodu, jest kompilator Pluto, który stosuje w module transformacji ATF. Inne narzędzia jak Cetus i Par4All nie oferują blokowania pętli. Spośród komercyjnych kompilatorów wymienia się narzędzie Intela, ICC, który jednak skutecznością w transformacjach pętli ustępuje Pluto i TRACO. Generuje on jednak szybki binarny kod wraz z optymalizacją sprzętową i wektoryzacją, który można skompilować także na podstawie źródeł wygenerowanych poprzez TRACO [3].

TRACO transformuje pętle programowe zgodne z modelem wielościennym i składnią C/C++. Kompilator został przećwiczony na wielu programach z następujących dziedzin: symulacji fizyki, obliczeniowej mechaniki płynów, zestawach testowych z algebry liniowej, przetwarzania obrazów i sygnałów, dynamicznego programowania (kombinatoryczna optymalizacja) i innych. W fazie wstępnej projektowania algorytmów używano kodów sztucznych z docelowym grafem zależności. W samym jednak testowaniu zastosowano kody programów rzeczywistych (ang, real-life codes) jak i zestawy testowe (benchmarki), których statyczny przepływ sterowania (ang. static control flow) jest zaszyty w wielu aplikacjach rzeczywistych.

*Miejsce uzyskanych wyników w klasyfikacji ACM*

W klasyfikacji ACM uzyskane wyniki należą do następujących dziedzin i poddziedzin[3]:

- *Software and its engineering* → *Compilers* (High)
- *Software and its engineering* → *Parallel programming languages* (High)
- *Software and its engineering* → *Automatic programming* (High)
- *Computing methodologies* → *Parallel computing methodologies* (High)
- *Theory of computation* → *Parallel computing models* (High)
- *Theory of computation* → *Scheduling algorithms* (High)
- *Theory of computation* → *Parallel algorithms* (High)
- *Mathematics of computing* → *Graph theory* (High)
- *Theory of computation* → *Dynamic programming* (High)
- *Applied computing* → *Bioinformatics;* (High)
- *Applied computing* → *Computational Biology* (High)

[3] W nawiasie podano stopień powiązania osiągnięć naukowych z daną klasyfikacją ACM.

- *Computing methodologies* → *Massively parallel and high-performance simulations* (Medium)
- *Software and its engineering* → *Multithreading* (Medium)
- *Software and its engineering* → *Massively parallel systems* (Medium)
- *Computing methodologies* → *Modeling and simulation* (Medium)
- *Computing methodologies* → *Linear algebra algorithms* (Low)
- *Hardware* → *Digital signal processing* (Low)

*Podsumowanie*

Do głównych osiągnięć zaprezentowanych w jednotematycznym cyklu publikacji zaliczam:

- rozwinięcie teorii podziału przestrzeni iteracji pętli programowych [1,2,4,5,7-10],
- sformalizowanie algorytmu ekstrakcji niezależnych fragmentów kodu o dowolnym grafie zależności [1,5,9],
- zastosowanie potęgi relacji zależności do wyznaczenia harmonogramowania swobodnego [2],
- algorytm blokowania pętli programowych w oparciu o tranzytywne domknięcie grafu zależności [4,7-10],
- wykazanie większych możliwości zaproponowanych technik do blokowania pętli z zakresu dynamicznego programowania [8,10],
- lepsza i automatyczna optymalizacja kodu Nussinowej do predykcji struktur RNA w porównaniu do pokrewnych rozwiązań [8,10],
- połączenie algorytmów blokowania opartych o tranzytywne domknięcie z klasycznymi algorytmami zrównoleglania [4,7,8,10],
- wskazanie ograniczeń transformacji afinicznych w zrównoleglaniu i poprawianiu lokalności oraz prezentacja możliwych rozwiązań opartych o sparametryzowane grafy zależności [1,2,4,5,8-10],
- przeprowadzenie licznych badań eksperymentalnych za pomocą TRACO i wykazanie skuteczności zaproponowanych rozwiązań [1-10],
- implementacja kompilatora TRACO [1-10].

W zrealizowanym projekcie kompilatora przeanalizowano tematy, które do tej pory były w niewielkim stopniu eksploatowane przez środowisko naukowe związane z automatycznym programowaniem i podziałem przestrzeni iteracji pętli programowych.

Optymalizacja kodu w oparciu o tranzytywne domknięcie grafu zależności jest zagadnieniem, które zapoczątkowała współpraca z badaczami z Paryskiego instytutu Inria oraz Politechniki Mediolańskiej. Mam nadzieję na dalsze zainteresowanie międzynarodowego środowiska naukowego najnowszymi tematami automatycznej optymalizacji pętli programowych z nieregularnym grafem zależności [4, 9], np. występujących w aplikacjach dynamicznego programowania [8] oraz modelami znajdującymi najlepsze rozmiary bloków [10]. Omówione artykuły zostały zacytowane przez uznanych specjalistów z dziedziny kompilatorów optymalizujących, m. in. Michelle Mills Strout (Univ. of Arizona), David Wonnacott (Havefort College, Philadephia), Paul Kelly (Imperial College London), Sven



Paulk

Verdoolaege (KU Leuven, Belgia), Albert Cohen (Inria), Louis-Noel Pouchet (Univ. of California) oraz Sanjay Rajopadhye (Colorado State University).

Przedstawiony cykl publikacji zawiera nowatorskie rozwiązania i zastosowania z zakresu informatyki, a dokładniej z teorii kompilatorów (automatyczna optymalizacja i generowanie kodu), przetwarzania równoległego oraz technik programowania i teorii grafów.

Cykl publikacji uzupełnia jednotematyczna lista przedstawiona w punktach II A) i E) wykazu dorobku naukowego (43 publikacje łącznie z omawianym cyklem). Łączna punktacja ministerialna to 537, z czego udział własny autora autoreferatu to 282,3. W ramach tej listy są przede wszystkim artykuły konferencyjne oraz jedna monografia [Bielecki11].

W dalszej części opracowania streszczony zostanie wkład naukowy i techniczny przedstawiony w poszczególnych artykułach cyklu publikacji.

## 1. *Coarse-grained loop parallelization: Iteration Space Slicing vs affine transformations*

Artykuł przedstawia algorytm ekstrakcji niezależnych fragmentów tj. równoległości pozbawionej synchronizacji (zwanej też gruboziarnistą). Taki kod równoległy jest efektywny na architekturach z wysokimi kosztami synchronizacji na platformach wieloprocesorowych lub kosztami komunikacji na platformach rozproszonych.

Przedstawione algorytmy podziału przestrzeni iteracji wyodrębniają w grafie zależności $D$ niezależne fragmenty kodu (ang. *slice*) w postaci luźno spójnie składowej (ang. *weakly connected component*) grafu $D$. Innymi słowy fragment kodu jest maksymalnym podgrafem grafu $D$, w którym istnieje nieskierowana ścieżka pomiędzy każdą parą wierzchołków.

Początki krańcowe (ang. *ulitmate dependence source*) zależności to instrukcje będące początkami zależności i niebędące jednocześnie jej końcami.

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

Przed wyznaczeniem fragmentów budowana jest unia wszystkich relacji zależności poddanych wstępnemu przetwarzaniu w przypadku pętli nieidealnie zagnieżdżonych.

Aby wyznaczyć równoległość pozbawioną synchronizacji należy wykonać 2 kroki:

1. wyznaczyć zbiór reprezentatywnych punktów fragmentów kodu,
2. zrekonstruować fragmenty kodu z punktów reprezentatywnych i wygenerować kod przebierający niezależne fragmenty.

W artykule podano metodę rozróżnienia topologii grafu na łańcuch, drzewa i grafy ogólne. W łańcuchach i drzewach wszystkie początki krańcowe są punktami reprezentatywnymi. W grafie ogólnym należy odtworzyć zależność pomiędzy punktem reprezentatywnym i punktami krańcowymi. Punkty reprezentatywne nie należą do punktów krańcowych połączonych nieskierowaną ścieżką, dla których istnieje inny mniejszy leksykograficznie początek krańcowy. Możliwe jest też wyznaczanie tych samych punktów reprezentatywnych za pomocą nieskierowanego grafu zależności oraz tranzytywnego domknięcia relacji prostych i odwrotnych [5].

Rekonstruowanie wszystkich instrukcji fragmentów kodu odbywa się za pomocą tranzytywnych ścieżek od początków reprezentatywnych. Generowanie kodu odbywa się w

dwóch etapach. Najpierw wygenerowana jest równoległa pętla przebierające reprezentatywne punkty fragmentów kodu. Następnie dołączana jest pętla sekwencyjna przebierająca pozostałe instrukcje fragmentów. W artykule podano dwa algorytmy wyznaczające instrukcje fragmentów dla postaci afinicznej za pomocą pętli licznikowej i dla postaci nieafinicznej za pomocą pętli warunkowej.

W pracy opisano szczegółowe porównanie możliwości podziału przestrzeni iteracji i tranzytywnego domknięcia grafu zależności z transformacjami afinicznymi.

W części eksperymentalnej przeprowadzono badania nad pętlami z zestawu testowego NASA Parallel Benchmark (obliczeniowa mechanika płynów) [NPB15] oraz UTDSP (cyfrowe przetwarzanie obrazów i sygnałów) [Sean99]. Zbadano czasy wykonywania algorytmów oraz efektywność i przyspieszenie obliczeń dla ośmiu procesorów czterordzeniowych oraz 96-rdzeniowej karty graficznej. Porównano także efektywność kodów przebierających instrukcje niezależnych fragmentów za pomocą pętli licznikowej i warunkowej.

Artykuł powstał przy współpracy z instytutem naukowym Inria (autorzy Cohen i Beletska). Artykuły [4, 9, Palkowski15] prezentują połączenie opisanej techniki wyznaczania niezależnych fragmentów kodu z blokowaniem pętli programowych.

## 2. *Free scheduling for statement instances of parameterized arbitrarily nested affine loops*

Artykuł [2] prezentuje algorytm do wyznaczania harmonogramowania swobodnego (ang. free-scheduling) instancji instrukcji afinicznych pętli programowych. Harmonogramowanie swobodne wykonuje instrukcje, gdy tylko wszystkie ich operandy są ustalone i jest najszybszym partycjonowaniem instrukcji w czasie (z najmniejszą liczbą punktów czasu). To pozwala na ekstrakcję maksymalnej drobnoziarnistej równoległości.

Do pokrewnych rozwiązań zalicza się znane prace Feautrier'a [Feautrier92_1, Feautrier92_2], Darte'go i Vivien'a [Darte94, Darte96] oraz rozwiązania zaimplementowane w kompilatorze Pluto. Pokrewne rozwiązania bazują na liniowych i afinicznych harmonogramach, które jednak nie gwarantują uzyskania harmonogramu swobodnego w ogólnych przypadkach pętli programowych.

Opisywany algorytm bazuje na obliczeniu $k$-tej potęgi $R^k$ relacji reprezentującej wszystkie zależności w pętli programowej. Relacja $R^k$ aplikowana jest na zbiorze początków krańcowych (zwanych też właściwymi, które nie są końcami innych zależności). Wówczas parametr $k$ symbolizuje zadany punkt w czasie, a początki krańcowe wykonywane są w pierwszej partycji czasu $k$=0. Poprzez kolejne kompozycje wyznaczane są kolejne zbiory instancji instrukcji dla $k$=1,2,3… .

Ponieważ w zadanym punkcie czasowym harmonogramu instancje muszą być niezależne (równoległe), należy opóźnić instancje, które są końcami zależności. W grafie zależności wszystkie instancje mają odległość do początków krańcowych $k$, lecz mogą istnieć zależności między samymi tymi punktami. Za pomocą tranzytywnego domknięcia i kompozycji instancje te przesuwane są do późniejszych punktów czasu.

Wynikiem algorytmu jest zbiór reprezentującym *k-ty* czas oraz krotki iteracji pętli, który jest wejściem do generatora kodu. Po wygenerowaniu kodu powstaje pętla programowa, której zewnętrzna pętla jest sekwencyjna (przebiera punkty czasowe), a dalsze są już tylko równoległe.

Warunkiem poprawności jest dokładne obliczenie potęgi $k$ relacji $R$. Skorzystano z autorskiej implementacji w bibliotece Omega. Warto dodać, że wyznaczenie dokładnego $R^k$ gwarantuje obliczenie dokładnego tranzytywnego domknięcia $R^+$ [Verdoolaege11].

W przypadku braku dokładnego wyniku zaprezentowano dwa alternatywne rozwiązania: ograniczenie się do aproksymacji, której wynikiem już jest zwykłe harmonogramowanie (nieswobodne) oraz powtórzenie procedury dla podprzestrzeni iteracji pętli programowej reprezentowanej przez pętle wewnętrzne (zewnętrzne są wykonywane sekwencyjnie).

W części badawczej artykułu zmierzono czasy obliczania $R^k$ jak i innych etapów kompilacji na zbiorze testowym obliczeniowej mechaniki płynów, NASA Parallel Benchmark. Zmierzono czasy wykonania, przyspieszenie i efektywność benchmarków na 96-rdzeniowej karcie graficznej z wykorzystaniem interfejsu programistycznego CUDA. W praktyce pokazano, że transfer danych pomiędzy pamięcią operacyjną i karty jest akceptowalnym kosztem.

Artykuł [Bielecki15] przedstawia połączenie wyżej opisanej techniki z blokowaniem pętli programowych [4].

## 3. *TRACO: Source-to-Source Parallelizing Compiler*

Niniejsza publikacja opisuje implementację autorskich algorytmów wzbogaconych o popularne transformacje pętli programowych. Kompilator TRACO zastąpił dotychczas rozwijaną bibliotekę ISSF (ang. *Iteration Space Slicing Framework*). Na wzór coraz nowocześniejszej implementacji kompilatora Pluto wprowadzono wiele mechanizmów automatyzacji.

Po pierwsze, algorytmy zostały przeniesione z nierozwijanej biblioteki Omega [Kelly95] do nowocześniejszej i dotychczas rozwijanej biblioteki ISL. W niej znajdują się wszystkie operacje arytmetyki Presburgera, w tym efektywniejsze algorytmy wyznaczania tranzytywnego domknięcia i $k$-tej potęgi relacji [Verdoolaege10].

Następnie, implementacja stała się narzędziem typu source-to-source, tzn. wejściem i wyjściem jest program lub jego fragment zawierający pętle programową zgodnie z składnią C/C++. To pozwoliło na przetwarzanie skomplikowanych kodów reprezentowanych przez modele wielościenne, przyspieszenie prac i zmniejszyło ryzyko pomyłek. Kompilator przetwarza wielościenny kod z generatora na kompilowalny kod zgodnie ze standardami OpenMP (dla procesorów wielordzeniowych i koprocesorów) [OpenMP17] oraz OpenACC dla procesorów graficznych [OpenACC17].

Algorytmy zaimplementowane są w języku Python wraz z biblioteką *islpy* [Klöckner15], która pozwala na korzystanie z biblioteki ISL. Dzięki temu prototypowanie nowych algorytmów stało się szybsze niż w języku C/C++. W późniejszym czasie, oprócz analizatora zależności Petit dodano analizator zależności Pet, analizator składni kodu Clan [Bastoul08] oraz nowsze generatory *cloog* [Bastoul04] i ISL AST [Verdoolaege12].

Algorytmy ekstrakcji niezależnych fragmentów kodu oraz równoległości z synchronizacją wzbogacone zostały o transformację prywatyzacji zmiennych i równoległą redukcję. Rozszerzyło to stosowalność kompilatora i zmniejszało zbiór relacji zależności na wejściu.

W części eksperymentalnej pokazano możliwości kompilatora za pomocą zestawów testowych NASA Parallel Benchmark i PolyBench [Pouchet15]. Badania przeprowadzono na

procesorach wielordzeniowych i kartach graficznych typu Tesla. W wynikach eksperymentów przedstawiono dane statystyczne stosowalności kompilatora w zestawach testowych, czasy wykonania wraz z przyspieszeniem i efektywnością oraz kosztami komunikacji.

W publikacji zawarto analizę porównawczą z innymi pokrewnymi narzędziami tj. Intel C++ Compiler (ICC), Pluto, Par4All, Cetus i PIT. Część eksperymentalna zawiera porównanie efektywności wygenerowanego kodu przez kompilator TRACO z wyżej wymienionymi narzędziami.

TRACO jest wielomodułową platformą rozwijaną do dziś. Jest oprogramowaniem wolnym, a jego źródła są publicznie dostępne (traco.sourceforge.net) dla badaczy, studentów i programistów. Kompilator usprawnił projektowanie i implementację nowych algorytmów wraz z testowaniem wygenerowanych kodów. Platforma pozwoliła także w późniejszych badaniach zbadać inne klasyczne oraz trudne w ręcznej analizie transformacje, a przede wszystkim blokowanie pętli w celu poprawienia lokalności kodu poprzez zwiększenie trafności odwołań do pamięci kieszeniowej.

## 4. *Tiling arbitrarily nested loops by means of the transitive closure of dependence graphs*

Publikacja przedstawia innowacyjne podejście blokowania dowolnie zagnieżdżonych pętli programowych za pomocą tranzytywnego domknięcia grafu zależności. Jest połączeniem modelu wielościennego i podziału przestrzeni iteracji oraz nie wymaga obliczenia funkcji afinicznych. Rozwiązanie nie jest zależne także od pełnej permutacji pętli programowych.

Przestrzeń iteracji dzielona jest początkowo na prostokątne bloki. Następnie za pomocą tranzytywnego domknięcia instrukcje stanowiące końce zależności są usuwane z bloków, jeżeli odpowiednie początki należą do późniejszych leksykograficznie bloków. Takie instrukcje są przesuwane do najstarszego leksykograficznie bloku, jeżeli instrukcja jest końcem wielu zależności. Artykuł zawiera dowód formalny poprawności tego podejścia. Warto też zwrócić uwagę, że docelowy kształt bloku ulega zmianie. Modyfikacja taka jest konieczna, jeśli w wektorach zależności występują współczynniki ujemne. W skrajnych przypadkach kod staje się sekwencyjny. Blokowane są instrukcje, które są w obrębie przynajmniej dwóch pętli programowych.

O skuteczności algorytmu decyduje możliwość wyznaczenia dokładnego tranzytywnego domknięcia grafu zależności. Jednakże, wyznaczenie aproksymacji tranzytywnego domknięcia wciąż jest poprawne kosztem optymalności (pod postacią mniejszej równoległości i lokalności). Do grafu dojdą tylko nieistniejące zależności, które ograniczą lokalność i równoległość wciąż poprawnego kodu.

Wygenerowany kod składa się z podwójnej liczby pętli w porównaniu do oryginalnego odpowiednika. Pierwsze pętle przebierają identyfikatory bloków, drugi zbiór przebiera instrukcje wewnątrz bloku. Instrukcje bloku wykonywane są sekwencyjnie, zatem zależności między nimi nie muszą być brane pod uwagę. Istotne są tylko zależności między blokami. Warto zwrócić uwagę, że taki graf jest pozbawiony cykli. W celu zrównoleglenia można zastosować dowolne techniki, w tym unimodularne, afiniczne [7,8] i z wykorzystaniem tranzytywnego domknięcia [9, Palkowski15] lub *k*-tej potęgi relacji zależności [Bielecki15, Palkowski15]. W artykule przedyskutowano jak nanieść równoległość



Palli

na zblokowany kod, tj. zamiast instancji instrukcji brane są pod uwagę identyfikatory bloków i zależności między nimi.

W publikacji zaprezentowano proces przetwarzania wstępnego relacji zależności jak i identyfikatorów bloków sprowadzając wszystkie instrukcje pętli nieidealnie zagnieżdżonej do tych samych wymiarów. Na przykładzie programu *k6* z zestawu testowego Livermoore Loops rozwiązującego ogólną postać rekurencji liniowej (ang. *general linear recurrence*) przedstawiono możliwość zblokowania pętli programowej, gdy niemożliwe jest wyznaczenie funkcji afinicznej i takie narzędzia jak Pluto zawodzą. Ponadto dokonano ręcznej transformacji w celu zrównoleglenia tej pętli i wykazano przyspieszenie na maszynie wieloprocesorowej (wykorzystano w tym celu właściwości asocjacji i komunikacji redukcji oraz usunięto część zależności).

Część eksperymentalną uzupełniono o badania z zestawu NASA Parallel Benchmark i PolyBench Suite na maszynie z 48 jednostkami obliczeniowymi. Odnotowano przyspieszenie i skalowalność obliczeń realizowanych przez wygenerowane kody za pomocą kompilatora TRACO.

### 5. *Synchronization-Free Automatic Parallelization for Arbitrarily Nested Affine Loops*

Artykuł opisuje ekstrakcję niezależnych fragmentów kodu w dowolnie zagnieżdżonych pętlach programowych. W porównaniu do podejścia [1] technika ta nie wymaga obliczenia tranzytywnego domknięcia unii wszystkich relacji zależności dla wszystkich gniazd dla każdego przypadku pętli programowej. Zamiast tego, algorytm w oparciu o graf SCC[4] oblicza tranzytywne domkniecie dla każdej instrukcji pętli osobno oraz dla każdej pary instrukcji oblicza tranzytywne domkniecie dla podgrafu reprezentującego zależności pomiędzy instancjami tej pary instrukcji. Pozwala to znacząco zredukować złożoność obliczeniową algorytmu znajdującego niezależne fragmenty.

W tym celu do zbioru relacji zależności dodawany jest zbiór ich odwróconych postaci tj. relacji po operacji inwersji, gdzie dziedzina staje się zakresem i odwrotnie. Warto zwrócić uwagę, że takie relacje nie są zgodne z porządkiem leksykograficznym. Dalej sposób obliczania punktów reprezentatywnych jest następujący, dla i-tego gniazda obliczana jest dziedzina relacji których początki znajdują się w i-tym gnieździe i odejmowany od niej zakres relacji, których zakresem są instrukcje z i-tego gniazda.

Algorytm jest zintegrowany z wyznaczaniem iteracyjnego tranzytywnego domknięcia przedstawionego w pracy [Bielecki14] i zintegrowany z kompilatorem TRACO. Tranzytywne domknięcie obliczane jest za pomocą zmodyfikowanego algorytmu Floyda-Warshalla. Za pomocą nieskierowanego grafu wyznaczane są pozostałe instrukcje należące do fragmentów kodu z poszczególnych gniazd i tranzytywnie zależne od znalezionych początków. Proces generowania kodu został uproszczony do pojedynczego etapu w porównaniu z rozwiązaniem [1]. Na wejściu do generatora kodu podawany jest zbiór przebierający punkty reprezentatywne i zależne instrukcje fragmentów kodu. Taka technika upraszcza ekstrakcję fragmentów kodów zwłaszcza dla dowolnie zagnieżdżonych pętli z wieloma gniazdami.

---

[4] Silna spójnie składowa grafu zależności (SCC) to graf, w którym węzły reprezentują instrukcje pętli natomiast krawędzie wskazują na zależności pomiędzy instancjami instrukcji.

W pracy pokazano na przykładach aplikacji symulacji z NAS Parallel Benchmark sposób działania podejścia, wyznaczanie fragmentów kodu dla poszczególnych gniazd oraz możliwość wygenerowania kodu przebierającego niezależne fragmenty, gdy transformacje afiniczne zawodzą (na przykładzie kompilatora Pluto). W części eksperymentalnej wykazano znaczne przyspieszenie dla testowanych kodów za pomocą koprocesora obliczeniowego Intel Xeon Phi. Dodatkowo porównano czasy obliczeń tranzytywnego domknięcia z bibliotekami Omega Calculator i ISL wykazując znacznie krótsze czasy iteracyjnego podejścia dla wielu pętli programowych z zestawu NAS Parallel Benchmark.

Podsumowując, zaprezentowane podejście wykazuje możliwość wykorzystania domknięcia do ekstrakcji niezależnych fragmentów kodu, gdy obliczenie dokładnego domknięcia unii relacji zależności jest niewykonywalne w zadanym czasie. Artykuł poszerza możliwość stosowania idei zaprezentowanej w pracy [1] z wykorzystaniem nieskierowanego grafu zależności i iteracyjnego wyznaczenia domknięcia.

### 6. *An Iteration Space Visualizer for Polyhedral Loop Transformations in Numerical Programming*

Artykuł opisuje narzędzie do wizualizacji modelu wielościennego w pętlach programowych. Moduł ten jest zintegrowany z kompilatorem TRACO. Wizualizuje on w przestrzeni dwuwymiarowej i trójwymiarowej zależności pętli programowych pomiędzy instancjami instrukcji. Pobiera on informacje z TRACO o dostępnej równoległości oraz kształtach bloków w zadanej przestrzeni iteracji. Udostępnione są różne funkcje m. in. obracanie, przybliżanie, kolorowanie, filtrowanie, przejrzystość bloków. Narzędzie pomaga przy projektowaniu i walidacji algorytmów do transformacji pętli programowych.

Moduł jest napisany w języku Python i oparty na bibliotece matplotlib. Informacje o zależnościach i zbiorach pobierane są z biblioteki islpy. Jednolite środowisko programistyczne znacznie ułatwiło stworzenie tego narzędzia.

Wizualizacja zintegrowana jest z algorytmami wykorzystującymi tranzytywne domknięcie grafu zależności, tj. ekstrakcji niezależnych fragmentów kodu, harmonogramowaniem swobodnym na poziomie instancji instrukcji jak i samych bloków. W artykule przedstawiono przykłady pętli programowych z wizualizacją. Ciekawymi opcjami są rozdzielenie gniazd instrukcji na podprzestrzenie bloków wraz z zależnościami oraz pokazanie równoległości zblokowanego kodu zademonstrowanego w artykule na przykładzie rozkładu Plancka z zestawu pętli testowych *Livermoore*.

W pracy omówiono narzędzia pokrewne. Najbliższym rozwiązaniem jest *islplot*, którego ograniczone możliwości 3D skłoniły do napisania własnego narzędzia. Ponadto istnieją inne moduły do prezentacji graficznej, które są jednak zintegrowane tylko z afinicznymi transformacjami. Autorska wizualizacja pobiera dane generowane przez kompilator w postaci relacji, zbiorów i krotek. Dzięki temu możliwe jest graficzne przedstawienie poszczególnych kroków algorytmów zaimplementowanych w TRACO.

### 7. *Parallel tiled code generation with loop permutation within tiles*

W niniejszej publikacji omówiono rozszerzenie techniki blokowania pętli programowych w oparciu o tranzytywne domkniecie grafu zależności [4]. Zwiększenie wydajności uzyskano poprzez wprowadzenie permutacji zblokowanego kodu.



Pall

W trakcie optymalizacji pętli programowej warte jest sprawdzenie czy jest możliwość przeprowadzenie techniki zamiany (przeplotu) kolejności pętli (ang. *loop interchange*). Jest to bazowa technika transformacji afinicznych. Zamiana kolejności pętli albowiem może zmienić odczyt kolumnami w odczyt wierszami, co dla programów napisanych w języku C/C++ i komputerów z pamięcią dzieloną oznacza poważny wzrost trafień do pamięci kieszeniowej procesora.

Jeżeli w pętli programowej nie ma zależności to dowolny przeplot jest poprawny. Permutacja jest uogólnieniem przeplotu i dotyczy każdej pętli, nie tylko wewnętrznych. W pracy wykazano, że warto poprawić lokalność kodu za pomocą obu technik permutacji i blokowania. Zmienne indeksowe pętli wewnętrznych powinny być ostatnimi zmiennymi w tablicach używanych w danej instrukcji. Kolejność pętli programowych jest określona także przez zmienne wejściowe ustalone przez eksperta (np. doświadczonego programistę).

Do wyznaczenia permutacji stosowane jest także tranzytywne domknięcie. Technika blokowania pętli [4] nanoszona jest na podbloki (ang. *sub-tiles*). Artykuł zawiera dowód poprawności tego podejścia. Warto zaznaczyć, że nie jest to klasyczny przeplot, lecz rozwiązania bardzo do niego zbliżone. Aby wygenerować podbloki definiowane są zbiory poprzednich i następnych podbloków, na wzór algorytmu prezentowanego w [4]. Jest to tylko konieczne, jeżeli w wektorach dystansu występują ujemne składowe. W przypadku nieujemnych elementów w wektorze otrzymamy rozwiązanie tożsame z permutacją pętli programowej.

Wiele pokrewnych rozwiązań, w tym prace Irigoin i Trioleta [Irigoin88] oraz Xue [Xue97, Xue12] mogą być stosowane tylko dla nieujemnych wektorów zależności (między instancjami instrukcji lub blokami) na wejściu. Mullaupadi i Bondhugula [Mullapudi14] wskazali na zbytnią „konserwatywność" tych ograniczeń. Budowanie schematów blokowania statycznego, chociaż nie jest łatwe, prowadzi do dużej wydajności.

W części eksperymentalnej zastosowano implementacje omawianego podejścia w kompilatorze TRACO. Wyniki badań zilustrowano za pomocą rzeczywistych kodów w zestawie testowym NASA Parallel Benchmark 3.3. Badania przeprowadzono na 32 wątkach. Do zrównoleglenia kodu zastosowano algorytmy ekstrakcji równoległości pozbawionej synchronizacji [1], aczkolwiek każda inna transformacja zrównoleglenia może zostać zastosowana, w której wejściem są zależności między blokami. Porównano czasy podejścia [4] z permutacją i bez. Opisano także typy zagnieżdżeń pętli, dla których permutacja nie zmieniła przyspieszenia zablokowanego kodu oraz typy, dla których omawiana modyfikacja umożliwia znaczne przyspieszenie, w tym superliniowe.

### *8. Parallel tiled Nussinov RNA folding loop nest generated using both dependence graph transitive closure and loop skewing*

Artykuł prezentuje automatyczne zrównoleglenie i blokowanie pętli programowej implementującej predykcję struktur RNA Nussinowej. Te zadanie obliczeniowe należy do zadań z bioinformatyki i dokonuje predykcji za pomocą maksymalizacji par bazowych w podsekwencjach RNA. Tranzytywne domknięcie jest użyte do blokowania kodu, natomiast zrównoleglenie jest uzyskane poprzez klasyczną transformację przekoszenia pętli programowej. Jest to pierwsza próba wygenerowania statycznego i równoległego kodu z zablokowaniem wszystkich trzech gniazd pętli programowych w predykcji Nussinowej.

Predykcja struktur RNA jest obliczeniowo złożonym zadaniem i należy do współczesnych zadań biologii obliczeniowej. Kody implementujące algorytmy, takie jak predykcja Nussinowej można przedstawić za pomocą modelu wielościennego, gdzie wyrażenie w pętlach programowych są afiniczne. Jednakże, klasyczne transformacje nie pozwalają na uzyskanie efektywnego kodu.

Ograniczenia te powoduje przynależność zadań predykcji do skomplikowanej klasy dynamicznego programowania NPDP (ang. *nonserial polyadic dynamic programming*), gdzie graf jest pełen niejednorodnych zależności (wektor zależności nie jest reprezentowany tylko przez stałe), które dodatkowo tworzą cykle w przestrzeni zależności między blokami.

Mullaupadi i Bondhugula [Mullaupadi14] opisali ograniczenia metod ATF zaimplementowanych w kompilatorze Pluto i zaproponowali dynamiczne harmonogramowanie bloków za pomocą łańcuchów redukcji. Rezultatem ich badań nie jest jednak statyczny kod dla NPDP. Ponadto, zaprezentowano tylko blokowanie najbardziej zagnieżdżonych instancji instrukcji w algorytmie Nussinowej. Zespół Wonnacotta opisał blokowanie wszystkich trzech gniazd pętli poprzez podzielenie iteracji na dwie zewnętrzne pętle i opóźnienie wykonania instancji instrukcji z trzeciego gniazda w algorytmie Nussinowej. Nie opisano jednak metody zrównoleglenia takiego kodu. W kompilatorze Pochoir [Tang11] opisano blokowanie o kształcie rombu dla algorytmu Gotoha, lecz nie zaprezentowano blokowania dla wszystkich gniazd pętli programowej. W popularnej ręcznej i równoległej implementacji predykcji RNA, GTFold (gtfold.sourceforge.net) nie wzięto pod uwagę technik poprawienia lokalności kodu.

Ograniczenia w pokrewnych rozwiązaniach i możliwość wyznaczenia dokładnego domknięcia dla algorytmu Nussinowej stało się motywacją niniejszego artykułu. Za pomocą domknięcia unii relacji możliwe jest zablokowanie wszystkich gniazd pętli programowych. Bloki są dzielone na poprawne i niepoprawne (w bloku niepoprawnym końce zależności znajdują się w leksykograficznie wcześniejszych blokach niż początki zależności). Niepoprawne końce zależności są zatem przenoszone w obrębie tego samego lub następnego gniazda instrukcji w pętli zgodnie z podejściem opublikowanym w [4] (algorytm Nussinowej składa się z dwóch gniazd instrukcji). Taki podział bloków jest nowym rozwiązaniem i nie występuje w pokrewnych rozwiązaniach i narzędziach. W pracy pokazano jak automatycznie wygenerować kod równoległy i udowodniono poprawność zblokowanego kodu po przekoszeniu pętli programowej.

W części eksperymentalnej predykcję przeprowadzono na sekwencjach o długości 2200 (średniej) oraz 5000 (najdłuższych) dla mRNA *homo sapiens* pobranych z bazy NCBI [NCBI17]. Przyspieszenie jest jednak niezależne od zawartości sekwencji RNA (tylko od jego długości) i wykazano podobne wyniki dla losowych wartości tablic RNA.

Następnie zaprezentowano znaczne przyspieszenie obliczeń dla 64 wątków dwóch współczesnych procesorów wielordzeniowych Intel Xeon E5-2699 v.3 i 244 wątków czterech koprocesorów obliczeniowych Intel Xeon Phi 7120P. W badaniach wykazano skalowalność obliczeń oraz lepszą wydajność i lokalność kodu w porównaniu do kodu opartego o dwuwymiarowe blokowanie wygenerowanego przez kompilator Pluto+.

Warto podkreślić, że w niniejszej pracy zaprezentowano możliwość połączenia metod blokowania w oparciu o tranzytywne domknięcie grafu zależności z metodami afinicznymi. Ponadto wykazano konkretną klasę programów rzeczywistych NPDP, dla których algorytmy zaimplementowane w kompilatorze TRACO są efektywniejsze niż popularne metody kompilatora Pluto i innych prac pokrewnych. Do takich zadań należą inne algorytmy z

bioinformatyki np. sekwencjonowanie DNA przy użyciu technik Smitha-Watermana oraz Neddlemana-Wuncha. Również dla energetycznego modelu predykcji RNA Zukera wykazano możliwość zblokowania wszystkich gniazd [Palkowski17], podczas gdy afiniczne metody zawodzą. Poza tym istnieją inne klasyczne problemy NPDP optymalizacji za pomocą dynamicznego programowania, takie jak optymalna triangulacja wielokąta, problem nawiasowania ciągu macierzy, przeszukiwanie drzew binarnych czy wyszukiwanie najdłuższej podsewkencji w łańcuchach znaków [Cormen09], dla których prezentowane techniki umożliwiają blokowanie wszystkich gniazd pętli programowych.

Skuteczność metod klasycznych ATF wykazano na licznych benchmarkach wielościennych, a przede wszystkim na zadaniach typu "stencil", gdzie komórka w tablicy periodycznie obliczana jest za pomocą wartości sąsiednich. Natomiast w zadaniach NPDP obliczenie każdej komórki wymaga przeglądu wszystkich poprzedników w danym wierszu i kolumnie (gdzie zazwyczaj złożoność zadaniowa należy do typu $O(n^3)$ lub $O(n^4)$, a pamięciowa $O(n^2)$). Jest to spowodowane tym, że implementacja obliczeń wykonywana jest za pomocą dynamicznego programowania, który jest tabelaryczną metodą rozwiązywania problemów zamiast niewydajnej rekurencji sprawdzającej te same podproblemy. Wypełnianie tablicy kosztów prowadzi do nieregularności w grafie zależności kodu DP, które ograniczają skuteczność ATF i wymagana jest analiza tego grafu w oparciu o jego tranzytywne domknięcie w celu osiągnięcia lepszej optymalizacji kodu.

**9. *Generation of parallel synchronization-free tiled code***

Artykuł przedstawia rozwiązanie wolnej od synchronizacji równoległego blokowania dowolnie zagnieżdżonych pętli programowych w oparciu o tranzytywne domknięcie grafu zależności. Blokowanie pętli programowych jest oparte o technikę zaprezentowanej w publikacji [4]. Wprowadzona równoległość na poziomie bloków przedstawiona jest jako niezależne fragmenty kodu, które są blokami lub podblokami. Wykazano, bowiem, że równoległość na poziomie instancji instrukcji zawiera więcej wątków i sam poziom niezależnych bloków obliczonych algorytmem z pracy [4] nie pozwala na uzyskanie maksymalnej liczby wątków dla kodu zblokowanego.

Przedstawiona technika składa się z następujących etapów: ekstrakcja niezależnych fragmentów kodu [1] na poziomie instancji instrukcji, zablokowanie pętli poprzez docelowe i prawidłowe względem porządku leksykograficznego bloki, aplikację części wspólnej niezależnych fragmentów kodu i bloków (co odróżnia od poprzednich rozwiązań [4, Palkowski15] formujących fragmenty z gotowych bloków), wygenerowanie kodu.

Bloki wynikowe dzielą się na następujące kategorie: bez punktów reprezentatywnych, z jednym punktem i wieloma punktami reprezentatywnymi. W ostatnim przypadku można wprowadzić kolejny podział bloków na części, z których każdy zawiera przynajmniej jeden punkt reprezentatywny. Algorytm pozwala także na wprowadzeniu ręcznej relacji mapującej punkty reprezentatywne wewnątrz pojedynczego bloku przez eksperta w celu uzyskania jeszcze lepszego wyniku.

Implementacja algorytmów została wykonana w kompilatorze TC [TC2017], który jest nowszą odsłoną kompilatora TRACO, dokonującego transformacji pętli z wykorzystaniem tranzytywnego domknięcia grafu zależności. Narzędzie te bazuje w całości na bibliotece ISL i zostało napisane tylko w języku C++.



Palli

W części eksperymentalnej zbadano osiem aplikacji z zestawu testowego PolyBench 4.1 [Pouchet15], w których można wydzielić niezależne fragmenty kodu. Przeanalizowano przyspieszenie kodu przebierającego niezależne fragmenty kodu składające się z całych i podzielonych bloków. Dla dwóch aplikacji uzyskano przyspieszenie superliniowe, które wartość kilkakrotnie przekracza liczbę wątków. Wykazano, że przyspieszenia dla innych kodów są porównywalne lub lepsze od przyspieszenia kodów wygenerowanych za pomocą kompilatora Pluto bazującego na transformacjach afinicznych.

**10. *Tuning Iteration Space Slicing based tiled multi-core code implementing Nussinov's RNA folding***

Artykuł porusza problem doboru odpowiedniego rozmiaru bloków w celu zwiększenia lokalności równoległego kodu wygenerowanego przez autorski kompilator za pomocą techniki TSS (ang. *tile size selection*). Aby skorzystać z tej techniki, zastosowano model opisujący wymiary bloków poprzez parametry obecne w zblokowanym kodzie.

W artykule przedstawiono metodę odtworzenia takiego kodu na podstawie niesparametryzowanych kodów. W miejsce stałych rozmiarów wstawiane są wyrażenia zależne od parametrów, które niekoniecznie muszą być afiniczne. Za pomocą kodu sparametryzowanego możliwe jest automatyczne zeskanowanie przestrzeni szukania i wyznaczenie dobrych rozmiarów (ang. *good tile sizes*).

Technika ta jest zastosowana do zblokowanego kodu równoległego Nussinowej omówionego w publikacji [8]. Dla problemów typu NPDP techniki afiniczne nie dokonują blokowania z udziałem trzeciej wewnętrznej pętli tego algorytmu predykcji RNA. Niestety, znane narzędzia do wyznaczania parametrycznego kodu blokowania bazują tylko na ATF. W pracy pokazano z kolei, że zblokowanie wszystkich pętli pozwala na szersze poszukiwanie odpowiednich rozmiarów bloków w przestrzeni trójwymiarowej.

W celu określenia wartości parametrów eksperymentalnie przeskanowano trójwymiarową przestrzeń, której każdy wymiar posiada po 20 rozmiarów, czyli $20^3$=8000 zestawów na krótkich sekwencjach RNA o długości 2200. Badania wysunęły następujące wnioski. Po pierwsze, blokowanie zewnętrznej pętli nie jest uzasadnione, ponieważ bloki 3D zwykle nie mieszą się w pamięci kieszeniowej lub większość bloków jest już nieprostokątna w wyniku korekcji bloków oryginalnych. Po drugie, bardzo ważne jest blokowanie drugiej i trzeciej pętli. Pozwala to na zachowanie oryginalnego prostokątnego kształtu bloku. Jednakże wielkość drugiego parametru musi być o rząd większa niż trzeci, ponieważ zbyt duże wielkości trzeciego rozmiaru rozsuwają od siebie linie pamięci kieszeniowej, co prowadzi do pogorszenia lokalności. Wyznaczone wielkości bloku B=[1,b2,b3], gdzie b2 > b3 można określić jako "złoty środek" i sprawdzają się również przy dłuższych sekwencjach RNA.

W części eksperymentalnej porównano wyniki z kodami afinicznego kompilatora Pluto, który generuje kod o znacznie gorszej lokalności, co już wykazano w pracy [8]. Poszerzono zatem porównanie o metody ręczne Changa [Chang10] i Li [Li14]. Chang zmodyfikował ręcznie rekurencję Nussinowej, aby poprawić lokalność kodu poprzez obliczanie komórek po przekątnych wypełnianej tablicy parowania sekwencji. Li z kolei ulepszył tą modyfikację poprzez wykorzystanie dolnej część tablicy (jest ona nieużywana w rekurencji Nussinowej) i umieszczenie w nich kopii komórek. W ten sposób zastąpił on czasochłonne czytanie komórek kolumnami, na odczyt wierszami w dolnej części tablicy.

Wyniki czasowe Li znacznie przewyższają wyniki zblokowanego kodu poprzez kompilator Pluto.

Zblokowanie drugiego i trzeciego gniazda dla wyznaczonych rozmiarów bloków umożliwiło poprawienie wyniku czasowego, który okazał się lepszy niż implementacja Li dla sekwencji dłuższych niż 2500. Dla zblokowanego kodu sekwencyjnego (wykonanego na 1 wątku) odnotowano przyspieszenie rzędu 3.7 w stosunku do oryginalnego kodu, natomiast dla kodu równoległego wykonanego przez 32 wątki uzyskano superliniowe przyspieszenie 112.9. Korzyści czasowe z pracy [8] zostały poprawione o około 30~40%. Zastosowanie modelu szacowania jakości kodu dla różnych wielkości bloków jest zatem zdecydowanie lepsze niż empiryczne i ręczne poszukiwanie odpowiednich rozmiarów, ponieważ automatyczne przeszukiwanie przestrzeni dostarcza więcej informacji z ogromnej ilości przeprowadzonych prób czasowych oraz pozwala na uzyskanie jeszcze większej wydajności prezentowanych kodów.

## Bibliografia


**[Allen01]** R. Allen, K. Kennedy, *Optimizing compilers for modern architectures: A Dependence-based Approach*, Morgan Kaufmann Publishers, Inc., 2001.

**[Bacon93]** D. F. Bacon, S. L. Graham, O. J. Sharp, *Compiler Transformations for High-Performance Computing*, ACM Computing Surveys 26, 1993.

**[Banerjee93]** U. Banerjee. *Loop Transformations for Restructuring Compilers*. pages 328, Kluwer Academic, 1993.

**[Baskaran10]** M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, P. Sadayappan: *Parameterized tiling revisited*. In:Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization. CGO '10, pp. 200–209. ACM,New York, NY, USA, 2010.

**[Bastoul04]** C. Bastoul, *Code generation in the polyhedral model is easier than you think*, PACT'13, IEEE International Conference on Parallel Architecture and Compilation Techniques, Juan-les-Pins, France, pp. 7–16, 2004.

**[Bastoul08]** C. Bastoul. *Extracting polyhedral representation from high level programs*. Technical Report, LRI, Paris-Sud University, 2008. Related to the Clan tool.

**[Beletskyy03]** V. Beletskyy, K. Siedlecki, *Finding free schedules for non-uniform loops*, in: Euro-Par 2003 Parallel Processing, Lecture Notes in Computer Science, vol. 2790/2003,, pp. 297–302., 2003.

**[Bielecki10]** W. Bielecki, T. Klimek, M. Pałkowski, A. Beletska, 2010, *An iterative algorithm of Computing the Transitive Closure of a Union of Parameterized Affine Integer Tuple Relations*, Lecture Notes in Computer Science, Springer, Volume 6508/2010 str. 1611-3349

**[Bielecki11]** W. Bielecki, M. Pałkowski, 2011, *Ekstrakcja drobno- i gruboziarnistej równoległości w pętlach programowych*, Wydawnictwo ZUT Szczecin, ISBN 9788376630977, liczba stron 260, (monografia).

**[Bielecki14]** W. Bielecki K. Kraska, T. Klimek, *Using basis dependence distance vectors to calculate the transitive closure of dependence relations by means of the Floyd-Warshall algorithm*. Journal of Combinatorial Optimization, 30(2), s. 253-275, 2014.

**[Bielecki15]** Włodzimierz Bielecki, Marek Pałkowski, Tomasz Klimek, 2015, Free Scheduling of Tiles Based on the Transitive Closure of Dependence Graphs., Lecture Notes in Computer Science, Springer, Vol. 9574, 11th Int. Conference on Parallel Processing and Applied Mathematics (PPAM'15), Kraków, str. 133-142.

**[Bondhugula08]** U. Bondhugula, A. Hartono, J. Ramanujam, P. Sadayappan, *A practical automatic polyhedral parallelizer and locality optimizer,* in: Conference on Programming Language Design and Implementation, ACM, pp. 101–113, 2008.

**[Bondhugula16]** U. Bondhugula, A. Acharya, A. Cohen *The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests*, ACM Transactions on Programming Languages and Systems (TOPLAS), vol 38, issue 3, Apr 2016.

**[Bondhugula17]** U Bondhugula, V Bandishti, I. Pananilath *Diamond Tiling: Tiling Techniques to Maximize Parallelism for Stencil Computations*, IEEE Transactions on Parallel and Distributed Systems (TPDS), pg 1285-1298, Vol 28, Issue 5, May 2017.




Pałłki

**[Chang10]** D. Chang, C. Kimmer, M. Ouyang, *Accelerating the Nussinov RNA folding algorithm with CUDA/GPU*. In: The 10th IEEE International Symposium on Signal Processing and Information Technology, pp. 120–125, doi:10.1109/ISSPIT.2010.5711746, 2010.
**[Cormen09]** T. H. Cormen, C. Stein, R. L. Rivest, C. E. Leiserson, Introduction to Algorithms, 3rd ed. The MIT Press, ISBN 0262033844, 9780262033848, 2009..
**[Darte94]** A. Darte, Y. Robert, *Constructive methods for scheduling uniform loop nests*, IEEE Trans. Parallel Distrib. Syst. 5, 814–822, 1994.
**[Darte96]** A. Darte, F. Vivien, *Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs*, in: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques, PACT '96, IEEE Computer Society, Washington, DC, USA, pp. 281–291, 1996.
**[Darte00]** A. Darte, Y. Robert, F. Vivien, *Scheduling and Automatic Parallelization*, Birkhäuser Boston, 2000.
**[Feautrier92_1 ]**P. Feautrier, *Some efficient solutions to the affine scheduling problem: I. one-dimensional time*, Int. J. Parallel Prog. 21 (5) (1992) 313–348.
**[Feautrier92_2]** P. Feautrier, *Some efficient solutions to the affine scheduling problem: II. multi-dimensional time*, Int. J. Parallel Prog. 21 (5) (1992) 389–420.
**[Feautrier12]** P. Feautrier. Approximating the transitive closure of a boolean-affine relation. In U. Bondhugula and V. Loechner, editors, IMPACT 2012, 2012.
**[Feautrier15]** P. Feautrier. The power of polynomials, In U. Bondhugula and V. Loechner, editors, IMPACT 2013, 2015.
**[Fisch74]** M. J. Fischer, M. O. Rabin, *Super-exponential complexity of Presburger arithmetic,* Proceedings of the SIAM-AMS Symposium in Applied Mathematics Vol. 7: 27–41, 1974.
**[Griebl00]** M. Griebl, P. Feautrier, and C. Lengauer. *Index set splitting*. International Journal of Parallel Programming, 28(6):607–631, 2000.
**[Griebl04]** M. Griebl, *Automatic Parallelization of Loop Programs for Distributed Memory Architectures,* D.Sc. thesis, University of Passau, Passau, 2004
**[Grosser14]** T. Grosser, S. Verdoolaege, A. Cohen, P. Sadayappan, *The relation between diamond tiling and hexagonal tiling,* Parallel Processing Letters 24(03): 1441,002, 2014.
**[Grosser15]** T. Grosser, S. Verdoolaege, A. Cohen, *Polyhedral ast generation is more than scanning polyhedra*, ACM Trans Program Lang Syst 37(4):12:1–12:50, 2015.
**[Irigoin88]** F. Irigoin, R. Triolet, *Supernode partitioning,* Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'88, San Diego, CA, USA, pp. 319–329, 1988.
**[Kelly95]** W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, D. Wonnacott, *The omega library interface guide,* Tech. rep., College Park, MD, USA, 1995.
**[Kelly96]** W. Kelly, W. Pugh, E. Rosser, T. Shpeisman, *Transitive closure of infinite graphs and its applications,* Languages and Compilers for Parallel Computing , LNCS, vol. 1033, pp. 126-140, 1996.
**[Kim09]** D. Kim, S.V. Rajopadhye, *Parameterized tiling for imperfectly nested loops,* Technical Report CS-09-101, Colorado State University, Fort Collins, CO, 2009.
**[Klöckner15]** A. Klöckner *islpy, a Python wrapper around Sven Verdoolaege's isl,* https://documen.tician.de/islpy, 2015.
**[Li14]** J. Li, S. Ranka, S. Sahni, *Multicore and GPU algorithms for Nussinov RNA folding,* BMC Bioinformatics 15(8), 1,. doi:10.1186/1471-2105-15-S8-S1, 2014.
**[Lim94]** A.W. Lim, M.S. Lam, M.S. *Communication-free parallelization via affine transformations*, in K. Pingali etal. (Eds.), 24th ACM Symposium on Principles of Programming Languages, Springer-Verlag, Berlin/Heidelberg, pp. 92–106, 1994.
**[Liv10]** Livermoore Loops Benchmark, http://www.netlib.org/benchmark/livermorec, 2010.
**[Mullapudi14]** R. T. Mullapudi, U. Bondhugula. *Tiling for dynamic scheduling*. In IMPACT 2014: 4rd International Workshop on Polyhedral Compilation Techniques, 2014.
**[NCBI17]** National Center for Biotechnology Information, *https://www.ncbi.nlm.nih.gov.*
**[NPB15]** *NAS Parallel Benchmarks suite,* http://www.nas.nasa.gov, 2015.
**[Nussinov78]** R. Nussinov, G. Pieczenik G, J.R. Griggs, D.J. Kleitman, *Algorithms for loop matchings*. SIAM J Appl Math.;35(1):68–82, 1978.
**[OpenACC17]** *The OpenACC 2.5 Application Programming Interface,* https://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5.pdf, 2017.
**[OpenMP17]** OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.5, http://www.openmp.org/wp-content/ uploads/openmp-4.5.pdf, 2017.

**[Palkowski15]** M. Pałkowski, T. Klimek, W. Bielecki, *TRACO: An automatic loop nest parallelizer for numerical applications*, Annals of Computer Science and Information Systems, IEEE Xplore® Digital Library., Vol. 7, str. 681-686 (FedCsis Łódź), 2015.
**[Palkowski17]** M. Palkowski, W. Bielecki:, *A Practical Approach to Tiling Zuker's RNA Folding Using the Transitive Closure of Loop Dependence Graphs*. Advanced in Inteligent Systems, vol 656, ISAT (2), pp. 200-209, 2017.
**[Park11]** E. Park, L.N. Pouchet, J. Cavazos, P. Sadayappan. *Predictive Modeling in a Polyhedral Optimization Space*. In 9th IEEE/ACM International Symposium on Code Generation and Optimization (CGO'11), Chamonix, France, April 2011.
**[Pouchet15]** L.N. Pouchet, T*he polyhedral benchmark suite v.4.1*, http://web.cse.ohio-state.edu/~pouchet/software/polybench, 2015.
**[Pugh93]** W. Pugh, D. Wonnacott, A*n exact method for analysis of value-based array data dependences*, in: Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing, Springer-Verlag, 1993.
**[Pugh97]** W. Pugh, E. Rosser, *Iteration space slicing and its application to communication optimization*, in: International Conference on Supercomputing, pp. 221–228, 1997.
**[Sean99]** P. Sean Hsien-en , *UTDSP: A VLIW Programmable DSP Processor*, 1999.
**[Strout04]** M.M. Strout, L. Carter, J. Ferrante, B. Kreaseck, *Sparse tiling for stationary iterative methods, Internaional Journal of High Performance Computing Applications* 18(1): 2004.
**[Tang11]** Y. Tang, R. A. Chowdhury, B.C. Kuszmaul, C. Luk, and C. E. Leiserson, *The Pochoir Stencil Compiler*. page 117-128, 23rd ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'2011), 2011.
**[TC17]** *TC Optimizing Compiler* http://tc-optimizer.sourceforge.net, 2017.
**[Traco17]** *TRACO Compiler*, traco.sourceforge.net, 2017.
**[Verdoolaege10]** S. Verdoolaege *ISL: an integer set library for the polyhedral model.* In: Mathematical software— 810 ICMS 2010, Lecture notes in computer science. vol 6327. Springer, Berlin, pp 299–302, 2010.
[**Verdoolaege11**] Sven Verdoolaege, Albert Cohen, and Anna Beletska. Transitive closures of affine integer tuple relations and their overapproximations. In SAS, pages 216–232, 2011.
**[Verdoolaege12]** S. Verdoolaege, T. Grosser, *Polyhedral extraction tool*. In: In Proceedings of the 2nd international 819 workshop on polyhedral compilation techniques. Paris, France, 2012.
**[Verdoolaege13]** S. Verdoolaege, J.C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, F. Catthoor, *Polyhedral Parallel Code Generation for CUDA*, Journal of ACM Transactions on Architecture and Code Optimization (TACO), Volume 9 Issue 4, 2013, 54:1-54:23, 2013.
**[Verdoolaege15]** S. Verdoolaege, *Integer set library—manual,* http://www.kotnet.org/~skimo//isl/manual.pdf, 2015.
**[Verdoolaege16]** S. Verdoolaege, *Presburger formulas and polyhedral compilation,* v0.02. Polly Labs and KU 814 Leuven., 2016.
**[Weiser84]** M. Weiser, *Program slicing*, in: IEEE Transactions on Software Engineering, pp. 352–357, 1984.
**[Wolf91]** M.E. Wolf, M.S. Lam, *A data locality optimizing algorithm,* Proceedings of the ACM SIGPLAN, Conference on Programming Language Design and Implementation, Toronto, Canada, pp. 30–44., 1991.
**[Wolfe95]** M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, pages 570, 1995.
**[Wonnacott13]** D. Wonnacott, M. Strout, *On the Scalability of Loop Tiling Techniques*, IMPACT 2013, http://impact.gforge.inria.fr/impact2013/papers/impact2013_on_the_scalability_of_loop_tiling_techniques.pdf
**[Wonnacott15]** D. Wonnacott D, T. Jin T, A. Lake, *Automatic tiling of "mostly-tileable" loop nests*. In: IMPACT 2015: 5th International Workshop on Polyhedral Compilation Techniques. Amsterdam; 2015. http://impact.gforge.inria.fr/ impact2015/papers/impact2015-wonnacott.pdf.
**[Xue97]** J. Xue, *On tiling as a loop transformation*, Parallel Processing Letters 7(4): 409–424, 1997.
**[Xue12]** J. Xue, *Loop Tiling for Parallelism*, Springer Science & Business Media, Springer-Verlag, New York, NY, USA, 2012.



Pałkowski Marek