Marek PAŁKOWSKI, Włodzimierz BIELECKI
KATEDRA INŻYNIERII OPROGRAMOWANIA, WYDZIAŁ INFORMATYKI, ZACHODNIOPOMORSKI UNIWESYTET TECHNOLOGICZNY
ul. Żołnierska 49, 71-210 Szczecin

Prywatyzacja zmiennych skalarnych dla wyznaczania równoległości w pętlach programowych

Dr inż. Marek PAŁKOWSKI

Stopień doktora uzyskał na Zachodniopomorskim Uniwersytecie Technologicznym z zakresu automatycznego zrównoleglania pętli programowych w oparciu o podział na fragmenty kodu. W badaniach koncentruje się na przetwarzaniu równoległym i jego zastosowaniach w środowiskach sprzętowo-programowym.

e-mail: mpalkowski@wi.zut.edu.pl

Prof. dr hab. inż. Włodzimierz BIELECKI

Jest kierownikiem Katedry Inżynierii Oprogramowania na Zachodniopomorskim Uniwersytecie Technologicznym. W badaniach koncertuje się na przetwarzaniu równoległym i rozproszonym, kompilatorów optymalizujących, ekstrakcji grubo- i drobnoziarnistej równoległości zawartej w pętlach programowych.

e-mail: wbielecki@wi.zut.edu.pl

Streszczenie

Wynikiem automatycznego zrównoleglenia pętli programowych jest kod tożsamy z sekwencyjnym odpowiednikiem, który honoruje wszystkie zależności. Zastosowanie prywatyzacji określonych zmiennych programu pozwala na wyeliminowanie zależności danych, co zmniejsza złożoność obliczeniową wyznaczania równoległości w pętlach programowych a także pozwala zwiększyć stopień wyznaczanej równoległości. W artykule zaprezentowano zastosowanie technikę prywatyzacji zmiennych operującej na zależnościach zapisanych w postaci relacji. Zbadano stosowalność proponowanego rozwiązania na pętlach programowych z benchmarku NAS.

Słowa kluczowe: prywatyzacja, równoległość gruboziarnista, zrównoleglanie pętli programowych.

Automatic privatization of scalar variables for coarse-grained parallelism extraction

Abstract

Extracting synchronization-free slices allows us to automatically generate parallel loops. The code can be executed on multi-processors machines in a reduced period of time. Privatization of data is an important technique used by compilers to parallelize loops by eliminating storage-related dependences. A scalar variable defined within a loop is said to be privatizable with respect to that loop if and only if every path from the beginning of the loop body to a use of *X* within that body must pass through a definition of *X* before reaching that use. In this paper there is presented an approach to automatic privatization of variables involved in data dependences that permits for extracting loop parallelism. The algorithm input is a set of relation dependences, the output is a parallel loop when appropriate. The scope of the approach applicability is illustrated by means of the NAS Parallel Benchmark suite. The obtained results are compared with another automatic parallelizer and locality optimizer for multicores – Pluto. The future work is considered.

Keywords: privatization, coarse-grained parallelism, parallelizing loops.

1. Wstęp

W celu wygenerowania programu równoległego tożsamego z sekwencyjnym odpowiednikiem niezbędne są techniki przetwarzania kodu, a zwłaszcza zrównoleglanie pętli programowych, w których zawarta jest zdecydowana większość obliczeń aplikacji. Istnieje wiele metod zrównoleglania pętli, w tym technika zrównoleglania pętli w oparciu o podział jej przestrzeni iteracji na niezależne fragmenty kodu w sposób automatyczny, zaproponowanej we wcześniejszej pracy autorów [5]. Skuteczność techniki zależna jest od algorytmów do wstępnego przetwarzania eliminujących zbędne zależności. Im mniej jest zależności, tym większa jest wyznaczana równoległość.

W niniejszym artykule zaproponowano algorytm realizujący prywatyzację zmiennych skalarnych. Prywatyzacja zmiennej w pętli programowej eliminuje zależności i poszerza stosowalność dalszych algorytmów zrównoleglenia. Istnieje wiele technik [1], które określają możliwość prywatyzacji zmiennych za pomocą teorii grafów lub rozwiązywania układów równań. Proponowane podejście pozwala na automatyczną prywatyzację zmiennych skalarnych w pętlach programowych korzystając z zapisu zależności obecnych w pętli otrzymanych z narzędzia Petit [2] oraz dedykowane jest do przetwarzania wstępnego poprzedzającego ekstrakcję równoległości gruboziarnistej za pomocą niezależnych fragmentów kodu.

2. Znajdowanie fragmentów kodu

Zrównoleglenie pętli programowej polega na podziale jej zbioru iteracji na części i rozdystrybuowanie ich do poszczególnych wątków aplikacji, które są następnie przydzielane do różnych jednostek obliczeniowych systemu. Pomiędzy iteracjami pętli jednakże mogą zachodzić **zależności danych**, które nie pozwalają na dowolny podział obliczeń pętli.

Dwie iteracje I i J są zależne, jeżeli odwołują się do tej samej komórki w pamięci i przynajmniej jedna z nich dokonuje operacji zapisu. I i J są określane odpowiednio początkiem i końcem zależności oraz I jest leksykograficznie mniejsza od J, $I \prec J$ (oznacza to, że I musi zostać wykonana przed J).

W obrębie zależności danych istnieją trzy podstawowe typy zależności [1, 3]:

1. Zależność przepływu danych zwana również **zależnością prostą** (ang. *Data - Flow Dependence, True Dependence*) [1, 3] - 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 δ S2 (odczyt S2 uzależniony jest od S1).
2. **Zależność odwrotna** (ang. *Antidependece*) [1, 3] - 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 δ^{-1} S2 (lub S1 δ^{-}S2).
3. **Zależność po wyjściu** (ang. *Output Dependence*) [1, 2, 3] - 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*) [3].

W zrównoleglaniu pętli programowych za pomocą fragmentów kodu (ang. *slices*) [5], wykorzystuje się dokładną analizę zależności opracowaną przez Pugh i Wonnacott'a [4], w której zależności reprezentowane są przez relacje między krotkami wejściowymi i wyjściowymi oraz ograniczeń zawierających formuły Presburgera:

{[*krotki wejściowe*] → [*krotki wyjściowe*] *: ograniczenia*};

Za pomocą zbiorów natomiast można określić przestrzeń i podprzestrzenie iteracji pętli programowej.

Do analizy zależności może być wykorzystane narzędzie *Petit* [2], które akceptuje na wejściu pętlę zapisaną w formacie zbliżonym do języka FORTRAN. W pętli dozwolone są podstawowe operacje arytmetyczne, tablice oraz instrukcje warunkowe.

Ograniczenia w relacjach i zbiorach określane są przez afiniczne równości i nierówności. Taka reprezentacja zależności jest elastyczna i pozwala także na definiowanie parametryzowanych zbiorów instrukcji.

Prywatyzacja zmiennych pozwala na znaczne wyeliminowanie zależności w procesie zrównoleglenia pętli programowych. Poniższy kod przykładowej pętli:

```
DO I = 1, N
S1       T = A(I)
S2       A(I) = B(I)
S3       B(I) = T
ENDDO
```

przekształcony na następujący:

```
PARALLEL DO I = 1, N
PRIVATE t
S1       t = A(I)
S2       A(I) = B(I)
S3       B(I) = t
END PARALLEL DO
```

pozwala na wyeliminowanie wszystkich zależności przenoszonych pętlą i zrównoleglenie poprzez prywatyzację zmiennej T, szczegóły są przedstawione w publikacji [1].

Zmienna skalarna *X* zdefiniowana w ciele pętli jest możliwa do **prywatyzacji** z honorowaniem zależności, wtedy i tylko wtedy, gdy każda ścieżka od początku ciała pętli do użycia zmiennej *X* w ciele pętli przechodzi przez jej definicję, innymi słowy zapis do zmiennej poprzedza jego użycie [1].

Idea zaprezentowanego dalej algorytmu oparta jest na powyższej definicji. W pracach [1, 13, 14] przedstawiono szczegóły technik prywatyzacji w oparciu o analizę grafu zależności i przepływu danych.

3. Algorytm do automatycznej prywatyzacji zmiennej skalarnej

Proponowany algorytm realizuje prywatyzację zmiennych operując na danych wejściowych uzyskanych z narzędzia Petit i jest możliwy do zaimplementowania w dowolnym środowisku.

Wejście:
Lista zmiennych skalarnych, zbiór *S* relacji zależności

Wyjście:
Zmodyfikowany zbiór *S* relacji zależności; ewentualnie kod równoległy, gdy zbiór jest pusty

Metoda:
1: Dla każdej zmiennej skalarnej *X*
  1.1 *y* = MIN(NumLines(*X*, *S*))
  1.2 Dla każdej zależności *R* ze zbioru *S*
    If Type(*R*) = anti and *R*(sour) = *X* and *R*(dest) = *X* and
    and LinSour(*R*) <= *y* then
    Private(*X*) = False , GOTO :1
  1.3 Private(*X*) = True
  1.4 Usuń *R* : *R*{sour} = *X* and *R*{dest} = *X*} w zbiorze *S*
2: Jeżeli zbiór *S* jest pusty to dokonaj zrównoleglenia pętli, w przeciwnym razie zastosuj inną technikę, np. ekstrakcję niezależnych fragmentów kodu [5].

Oznaczenia:
NumLines(*X*, *S*) – zbiór wartości numerów linii w kodzie pętli, gdzie zmienna *X* jest początkiem lub końcem wszystkich zależności ze zbioru *S,*
R(sour / dest) – zmienna, do której odnosi się początek / koniec zależności reprezentowanej przez relacje *R,*
Type(*R*) – typ zależności reprezentowany przez relację *R,*
LineSour(R) – numer linii kodu, do której odnosi się początek zależności reprezentowanej przez relację R

Jeśli zbiór *S* jest pusty, to wszystkie gniazda pętli można poczynić równoległymi. Jeśli zrównoleglona pętla będzie najbardziej zewnętrzna, to będziemy mieli do czynienia z równoległością gruboziarnistą [5], natomiast, jeśli najbardziej zewnętrzna pętla będzie sekwencyjna, a pozostałe równoległe, to będziemy mieli do czynienia z drobno-ziarnistą równoległością.

Zilustrujemy przedstawiony algorytm za pomocą przykładu niżej:

```
for iel=1 to N1 do
 ntemp=lx1*lx1*lx1*(iel-1)
 for j=1 to N2 do
  for i=1 to N3 do
   idel(i,j,1,iel)=ntemp+(i-1)*lx1 +
                   (j-1)*lx1*lx1+lx1
   idel(i,j,2,iel)=ntemp+(i-1)*lx1 +
                   (j-1)*lx1*lx1+1
   idel(i,j,3,iel)=ntemp+(i-1)*1 +
                   (j-1)*lx1*lx1+lx1*(lx1-1)+1
   idel(i,j,4,iel)=ntemp+(i-1)*1 +
                  (j-1)*lx1*lx1+1
   idel(i,j,5,iel)=ntemp+(i-1)*1 +
                   (j-1)*lx1+lx1*lx1*(lx1-1)+1
   idel(i,j,6,iel)=ntemp+(i-1)*1 + (j-1)*lx1+1
  endfor
 endfor
endfor
```

Wynik analizy zależności za pomocą narzędzia Petit [2] zamieszczono poniżej. W jego pierwszej kolumnie zapisano rodzaj zależności: *flow* – prosta, *anti* – odwrotna, *output* – po wyjściu. W drugiej kolumnie podano numer linii początku zależności. W trzeciej kolumnie podano nazwę zmiennej, do której odnosi się początek zależności. W kolumnie czwartej i piątej podano numer linii końca zależności oraz zmienną, do której się odnosi. W następnej kolumnie podano czy zależność zachodzi w tej samej iteracji pętli (0) czy jest przenoszona między iteracjami (+). W ostatniej kolumnie podano informacje dodatkowe o zależnościach. Symbol *M* oznacza, że początek i końce zależności odnoszą się do tego samego miejsca w pamięci. Symbol *o* określa, że zależność jest cykliczna. Dla przykładowej pętli wszystkie zależności posiadają obie wyżej wymienione cechy.

```
flow   23: ntemp -->  29: ntemp  (0)   [ Mo]
flow   23: ntemp -->  29: ntemp  (+)   [ Mo]
flow   23: ntemp -->  31: ntemp  (0)   [ Mo]
flow   23: ntemp -->  31: ntemp  (+)   [ Mo]
flow   23: ntemp -->  33: ntemp  (0)   [ Mo]
flow   23: ntemp -->  33: ntemp  (+)   [ Mo]
flow   23: ntemp -->  35: ntemp  (0)   [ Mo]
flow   23: ntemp -->  35: ntemp  (+)   [ Mo]
```

```
flow   23: ntemp -->  37: ntemp  (0)    [ Mo]
flow   23: ntemp -->  37: ntemp  (+)    [ Mo]
flow   23: ntemp -->  39: ntemp  (0)    [ Mo]
flow   23: ntemp -->  39: ntemp  (+)    [ Mo]
output 23: ntemp -->  23: ntemp  (+)    [ Mo]
anti   29: ntemp -->  23: ntemp  (+)    [ Mo]
anti   31: ntemp -->  23: ntemp  (+)    [ Mo]
anti   33: ntemp -->  23: ntemp  (+)    [ Mo]
anti   35: ntemp -->  23: ntemp  (+)    [ Mo]
anti   37: ntemp -->  23: ntemp  (+)    [ Mo]
anti   39: ntemp -->  23: ntemp  (+)    [ Mo]
```

Wyniki zastosowania algorytmu są następujące.

1. Dla zmiennej *ntemp*
   1.1 NumLines(*ntemp, S*) = {23,29,31,35,37,39}
   y = 23
   1.2 Dla wszystkich zależności odwrotnych, wartości LinSour(R) tworzą zbiór{29,31,33,35,37,39}. Żadna z tych wartości nie jest mniejsza od y=23, zatem warunek LinSour(R) <= y nie będzie nigdy spełniony.
   1.3 Private(ntemp) = True
   1.4 S = NULL
2. Pętla zewnętrzna może zostać zrównoleglona. Uzyskana równoległość jest gruboziarnista i pozbawiona synchronizacji.

```
#pragma omp parallel for private(ntemp)
for iel=1 to N1 do
 ntemp=lx1*lx1*lx1*(iel-1)
 for j=1 to N2 do
  for i=1 to N3 do
   idel(i,j,1,iel)=ntemp+(i-1)*lx1 +
                   (j-1)*lx1*lx1+lx1
   idel(i,j,2,iel)=ntemp+(i-1)*lx1 +
                   (j-1)*lx1*lx1+1
   idel(i,j,3,iel)=ntemp+(i-1)*1 +
                   (j-1)*lx1*lx1+lx1*(lx1-1)+1
   idel(i,j,4,iel)=ntemp+(i-1)*1 +
                  (j-1)*lx1*lx1+1
   idel(i,j,5,iel)=ntemp+(i-1)*1 +
                   (j-1)*lx1+lx1*lx1*(lx1-1)+1
   idel(i,j,6,iel)=ntemp+(i-1)*1 + (j-1)*lx1+1
  endfor
 endfor
endfor
```

Proponowany algorytm może być stosowany do przetwarzania wstępnego (ang. *pre-processing*) przed stosowaniem innych technik ekstrakcji równoległości w pętlach programowych [5, 6, 7, 8]. Korzystając z wyników jego działania możliwe jest automatyzowanie dalszego procesu redukcji zależności. Zaletą algorytmu jest jego niska złożoność obliczeniowa oraz prostota implementacji. Może on operować na dużych zbiorach zależności.

Proponowane podejście można rozszerzyć na zmienne tablicowe. Jeżeli zbiór zależności przenoszonych pętlą odwołuje się do tych samych referencji zmiennych indeksowych, to można potraktować zmienną jak skalar i zastosować wcześniej podany algorytm.

Dla przykładu niżej

```
#pragma omp parallel for private(a)
 for j=1 to N do
  for i=1 to N do
    a(i) = b(i,j);
    c(i,j) = a(i);
```

Petit zwraca następującą relacjcję zależności:

```
flow     1:  a(i)  -->   2:  a(i)    (0)      [ Mo]
```

Zależność odnosi się do zmiennej tablicowej a(i). Indeksy tablic są takie same. Nie ma zależności odwrotnej; warunek 1.2 algorytmu nigdy nie będzie spełniony. Zatem zmienna a może zostać sprywatyzowana. Po usunięciu jedynej zależności zbiór S będzie pusty. Można dokonać zrównoleglenia pętli. Dla zachowania poprawności semantycznej w tablicy a muszą zostać zapisane dane z iteracji J=N, I=N.

4. Badania eksperymentalne

Z zestawu testowego NAS Parallel Benchmark [9] wytypowano następujące 25 pętli, które zaprezentowano w tabeli 3. We wszystkich przypadkach pętla zewnętrzna została zrównoleglona. W pierwszej kolumnie podano nazwę pętli, w drugiej liczbę zależności. W trzeciej kolumnie podano stopień zagnieżdżenia oraz czy pętla jest idealnie zagnieżdżona (t) lub nieidealnie (n). W następnej kolumnie podano zmienne, które zostały sprywatyzowane. W ostatniej kolumnie podano liczbę niezależnych wątków obliczeń, na które składa się uzyskany kod równoległy. Wszystkie pętle są sparametryzowane.

Tab. 3. Pętle z benchmarku NAS
Tab. 3. Loops of NAS Parallel Benchmark

Pętla	Liczba zależności	Zagnieżdżenie pętli / idealnie	Sprywatyzowane zmienne	Liczba wątków
BT_initialize.f2p_2	42	3 / n	eta, zeta, temp	N1+1
BT_initialize.f2p_3	42	3 / n	eta, zeta, temp	N1+1
BT_initialize.f2p_4	42	3 / n	xi, zeta ,temp	N1+1
BT_initialize.f2p_5	42	3 / n	xi, zeta ,temp	N1+1
BT_initialize.f2p_6	42	3 / n	xi, eta ,temp	N1+1
BT_initialize.f2p_7	42	3 / n	xi, eta ,temp	N1+1
BT_rhs.f2p_1	46	3 / t	rho_inv	N1+1
BT_rhs.f2p_5	128	3 / t	wijk, wp1, wm1	N1
FT_auxfnct.f2p_1.t	1	1 / t	dummy	N1 - 1
LU_erhs.f2p_2	66	4 / n	xi, eta, zeta	N1
LU_HP_erhs.f2p_2	66	4 / n	xi, eta, zeta	N1
LU_HP_pintgr.f2p_2	109	2 / t	k	N2-N1
LU_pintgr.f2p_2	109	2 / t	k	N2-N1
SP_initialize.f2p_2	42	3 / n	eta, zeta, temp	N1+1
SP_initialize.f2p_3	42	3 / n	eta, zeta, temp	N1+1
SP_initialize.f2p_4	42	3 / n	xi, zeta ,temp	N1+1
SP_initialize.f2p_5	42	3 / n	xi, zeta ,temp	N1+1
SP_initialize.f2p_6	42	3 / n	xi, eta ,temp	N1+1
SP_initialize.f2p_7	42	3 / n	xi, eta ,temp	N1+1
SP_rhs.f2p_1	46	3 / t	rho_inv	N1+1
SP_rhs.f2p_5	128	3 / t	wijk, wp1, wm1	N1
SP_txinvr.f2p_1	271	3 / t	ru1, uu, vv, ww, a2cinv, r1, r2, r3, r4, r5, t1,t2, t3	N1
SP_tzetar.f2p_1	288	3 / t	xvel, yvel, zvel, ac, a2cu, r1, r2, r3, r4, r5, t1, t2, t3, uzik1, btuz	N1
UA_setup.f2p_14	31	4 / n	temp, dtemp, temp1, temp2	N1
UA_setup.f2p_15	19	3 / n	ntemp	N1

Badany zestaw pętli programowych spróbowano zrównoleglić korzystając z innego rozwiązania: Pluto (ang. *An automatic parallelizer and locality optimizer for multicores*). Jest to narzędzie, które dokonuje automatycznego zrównoleglenia pętli w oparciu o model wielościenny (ang. *polyhedral model*) [10]. Głównym aparatem do przekształcania pętli są transformacje afiniczne [11, 12]. Pluto dedykowany jest do programów w języku C dla ekstrakcji równoległości drobno- grubo-ziarnistej z jednoczesnym zwiększeniem lokalności kodu. Pętle równoległe zapisywane są automatycznie za pomocą pragm OpenMP. Dla wszystkich badanych pętli narzędzie Pluto nie znalazło równoległości (sprawdzono dla najnowszej wersji narzędzia 0.7). Rezultatem działania narzędzia jest kod sekwencyjny, identyczny z oryginalnym.

5. Prace pokrewne

Popularne techniki prywatyzacji zmiennych oparte są na analizie grafu i przepływu danych lub wyszukiwania określonych mapowań [1, 13, 14]. W proponowanym rozwiązaniu wykorzystywany jest tylko zapis zależności w postaci relacji uzyskanych z narzędzia Petit. Implementacja ogranicza się do analizy danych tekstowych.

W prywatyzację zmiennych wyposażony jest także sam analizator zależności Petit [2].

Jednakże narzędzie Petit:

- umożliwia tylko prywatyzację zmiennych tablicowych,
- niektóre funkcje w tym prywatyzacji nie działają poprawnie w najnowszej wersji oraz projekt nie jest od dłuższego czasu rozwijany,
- wynik nie jest wyznaczeniem równoległości i nie jest generowany kod równoległy.

Powyższe czynniki były motywacją do opracowania i zaimplementowania własnego rozwiązania w celu poszerzenia spektrum zrównoleglenia pętli programowych.

6. Podsumowanie

Proponowane podejście zwiększa spektrum pętli programowych, które można zrównoleglić stosując algorytmy ekstrakcji niezależnych fragmentów kodu [1]. Pozwala ono na eliminację zależności oraz ekstrakcję drobno- i grubo-ziarnistej równoległości dla przypadków, dla których inne narzędzia, jak Pluto, zawodzą.

Proponowane rozwiązanie zawiera podstawową technikę prywatyzacji zmiennych. Zastosowanie bardziej zaawansowanych technik prywatyzacji do przetwarzania wstępnego w ekstrakcji równoległości gruboziarnistej będzie tematem przyszłych badań.

7. Literatura

[1] Allen R., Kennedy K.: Optimizing compilers for modern architectures: A Dependence based Approach, Morgan Kaufmann Publish., Inc, 2001.

[2] Kelly W., Pugh W., Maslov V., Rosser E., Shpeisman T., Wonnacott D.: New User Interface for Petit and Other Extensions. User Guide, 1996.

[3] Moldovan D.: Parallel Processing: From Applications to Systems, Morgan Kaufmann Publishers, Inc, 1993.

[4] Pugh W., Wonnacott D.: An exact method for analysis of value-based array data dependences. In In Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing. Springer-Verlag, 1993.

[5] Beletska A., Bielecki W., Cohen A., Palkowski M., Siedlecki K.: Coarse-grained loop parallelization: Iteration space slicing vs affine transformations. Parallel Computing, 37, s. 479–497, 2011.

[6] Pugh W., Rosser E.: Iteration Space Slicing and Its Application to Communication Optimization. Proceedings of the International Conference on Supercomputing. s. 221-228, 1997.

[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, s. 185-195, 2008.

[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, s. 145-153, 2008.

[9] NAS Parallel Benchmarks v.3.2, 2010. http://www.nas.nasa.gov/Resources/Software/npb.html

[10] PLUTO - An automatic parallelizer and locality optimizer for multicores, http://pluto-compiler.sourceforge.net/, 2011.

[11] Lim A. W., Lam M., Cheong G.: An affine partitioning algorithm to maximize parallelism and minimize communication. In ICS'99, s. 228-237, ACM Press, 1999.

[12] Feautrier P.: Some efficient solutions to the affine scheduling problem, part i, ii, one dimensional time, International Journal of Parallel Programming 21, s. 313-348, 389- 420, 1992.

[13] Gupta M.: On Privatization of Variables for Data-Parallel Execution, In Proceedings of the 11th International Parallel Processing Symposium, 1997.

[14] Padua D., Peng T.: Automatic Array Privatization, Proc. 6th Workshop on languages and compilers for Parallel Computing, 1993.

otrzymano / received: 06.12.2011
przyjęto do druku / accepted: 03.01.2012 *artykuł recenzowany / revised paper*

INFORMACJE

Informacja redakcji dotycząca artykułów współautorskich

W miesięczniku PAK od numeru 06/2010 w nagłówkach artykułów współautorskich wskazywany jest autor korespondujący (Corresponding Author), tj. ten z którym redakcja prowadzi wszelkie uzgodnienia na etapie przygotowania artykułu do publikacji. Jego nazwisko jest wyróżnione drukiem pogrubionym. Takie oznaczenie nie odnosi się do faktycznego udziału współautora w opracowaniu artykułu. Ponadto w nagłówku artykułu podawane są adresy korespondencyjne wszystkich współautorów.

Wprowadzona procedura wynika z międzynarodowych standardów wydawniczych.

Redakcja