Włodzimierz BIELECKI, Marek PALKOWSKI, Krzysztof SIEDLECKI
WEST POMERANIAN UNIVERSITY OF TECHNOLOGY, DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY,
uL. Żołnierska 49, 71-210 Szczecin

Using transitive closure and transitive reduction to extract coarse-grained parallelism in program loops

Prof., Ph.D., Ds.C., eng. Wlodzimierz BIELECKI

He is head of the Software Technology Department of the West Pomeranian University of Technology, Szczecin. His research interest include parallel and distributed computing, optimizing compilers, extracting both fine and coarse grained parallelism available in program loops.

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

Ph.D., eng. Marek PAŁKOWSKI

He has graduated and obtained his Ph.D. degree in Computer Science from the Technical University of Szczecin, Poland. The main goal of his research is extracting parallelism from program loops and developing Iteration Space Slicing Framework.

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

Ph.D. eng. Krzysztof SIEDLECKI

He has graduated and obtained his Ph.D. degree in Computer Science from the Technical University of Szczecin, Poland. His research focus on optimizing compilers, parallel and distributed processing, software engineering.

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

Abstract

A technique for extracting coarse-grained parallelism available in loops is presented. It is based on splitting a set of dependence relations into two sets. The first one is to be used for generating code scanning slices while the second one permits us to insert send and receive functions to synchronize the slices execution. The paper presents a way demonstrating how to remove redundant synchronization in generated code by means of the transitive reduction operation. Results of experiments – how many synchronization points can be removed, speed-up and efficiency of examined parallel loops are discussed.

Keywords: synchronization, slices, parallelism, transitive closure and reduction.

Redukcja nadmiarowej synchronizacji w ekstrakcji równoległości gruboziarnistej

Streszczenie

W artykule zaprezentowano technikę ekstrakcji równoległości gruboziarnistej w pętlach programowych. Bazuje ona na podziale relacji zależności na dwa zbiory: na podstawie pierwszego generowany jest kod skanujący niezależne fragmenty, natomiast drugi służy do wstawienia funkcji send i receive (wyślij i odbierz) służących do synchronizacji tych fragmentów. Operacje te zrealizowano za pomocą semaforów, możliwe jest jednak wykorzystanie innej konstrukcji, bardziej wydajnej dla danego środowiska. Algorytm generuje kod z zaznaczonymi punktami synchronizacji, nie narzuca jednak ich implementacji. W artykule przeanalizowano technikę wyszukiwania i eliminacji zbędnych punktów synchronizacji. Ekstrakcja równoległości za pomocą fragmentów kodu bazuje na operacji tranzytywnego domknięcia, znanej także z teorii grafów. Operacja ta jest również wykorzystana do obliczenia tranzytywnej redukcji, za pomocą której eliminowana jest nadmiarowa synchronizacja. Usuwanie zbędnej komunikacji pomiędzy wątkami obliczeń jest istotne, ponieważ ich obsługa zwłaszcza dla komputerów z pamięcią dzieloną, w których ich koszt obsługi jest istotny. Docelowe jest zatem uzyskanie gruboziarnistego kodu równoległego. Zbadano także wyniki przeprowadzonych eksperymentów pod kątem przyspieszenia i efektywności obliczeń.

Słowa kluczowe: fragmenty kodu, synchronizacja, równoległość, tranzytywne domknięcie i redukcja.

1. Introduction

A common approach to executing scientific programs on a parallel machine is to distribute the iterations of a program loop across processors. If there are no dependences between iterations executed on different processors, then the processors can execute the iterations completely independently. Otherwise, the processors have to synchronize at certain points to preserve the original sequential semantics of the program [1].

In our recent work [2-3], we have proposed several algorithms to extract coarse-grained parallelism represented with synchronized slices consisting of the loop statement instances. The necessary synchronization is achieved by placing Send and Receive functions in generated code. We have shown in paper [3] how massage passing can be applied to produce coarse-grained parallel programs. To implement this mechanism, we introduced send and receive functions being based on locks. The number of synchronization points defines the parallel code performance. Reducing this number may considerably improve code performance. So, there is the need to eleminate reduntant synchronization to achieve more efficient parallel code.

The main goal of this paper is to demonstrate how transitive transitive closure and transitive reduction can be used to significantly reduce the number of synchronization points.

2. Background

In this paper, we deal with affine loop nests where i) for given loop indices, lower and upper bounds as well as array subscripts and conditionals are affine functions of surrounding loop indices and possibly of structure parameters (i.e., parameterized loop bounds) and, ii) the loop steps are known positive constants.

A nested loop is called perfectly nested if all its statements are comprised within the innermost nest. Otherwise, the loop is called imperfectly nested. An arbitrarily nested loop can be both perfectly and imperfectly nested.

A statement instance $s(I)$ is a particular execution of a loop statement s for a given iteration I.

Two statement instances $s_1(I)$ and $s_2(J)$ are *dependent* if both access the same memory location, and if at least one access is a Write. $s_1(I)$ and $s_2(J)$ are called the source and destination of a dependence, respectively, provided that $s_1(I)$ is lexicographically smaller than $s_2(J)$ ($s_1(I) \prec s_2(J)$, i.e., $s_1(I)$ is always executed before $s_2(J)$).

To describe and implement our algorithms, we choose the dependence analysis proposed by Pugh and Wonnacott [5] where dependences are represented by dependence relations whose constraints are described in the Presburger arithmetic (built of affine equalities and inequalities, logical and existential operators); the Omega library is used for computations over such relations [6-7].

A dependence relation is a tuple relation of the form

$$\{[\textit{input list}] \rightarrow [\textit{output list}] : \textit{constraints}\}, \qquad (1)$$

where *input list* and *output list* are the lists of variables and/or expressions used to describe input and output tuples and *constraints* is a Presburger formula describing constraints imposed upon *input list* and *output list*.

We use standard operations on relations and sets, such as intersection ($\cap$), union ($\cup$), difference (-), domain of relation (domain(R)), range of relation (range(R)), relation application (given a relation R and set S, $R(S) = \{[e']: \exists e \in S, e \rightarrow e' \in R\}$), transitive closure. These operations are described in detail in [7].

The **transitive closure** of a directed graph $G=(V,E)$ is a graph $H=(V,F)$ with edge (v,w) in H if and only if there is a path from v to w in G. A graph can be represented by an integer tuple relation whose domain consists of integer k-tuples and whose range consists of integer k'- tuples, for some fixed k and k'. It is evident that an integer tuple relation describes the corresponding graph $G=(V,E)$, where the input and output tuples of the relation represent vertices of the graph while affine equalities and inequalities describe the edges of the graph (an edge exists if corresponding affine equalities and inequalities are honored for a given pair of vertices represented with input and output tuples). There exist two relations related to transitive closure: positive transitive closure, R^+, and transitive closure, $R^* = R^+ \cup I$, where I is the identity relation.

Definition: Positive transitive closure is defined as follows:

$$R^+ = \bigcup_{k=1}^{\infty} R^k, \qquad (2)$$

Where $R^k = R \circ R^{k-1}$, $R^{k-1} = R \circ R^{k-2}$, ..., $R^1 = R$, $R^0 = I$.

Definition: Given a dependence graph, D, defined by a set of dependence relations, S, a **slice** is a weakly connected component of graph D, i.e., a maximal subgraph of D such that for each pair of vertices in the subgraph there exists a directed or undirected path.

If there exist two or more slices in D, then taking into account the above definition, we may conclude that all slices are synchronization-free, i.e., there is no dependence between them.

Definition. A **transitive reduction** of a relation R is a minimal relation R' such that the transitive closure of R' is the same as the transitive closure of R.

3. Redundant synchronization removal

The algorithm proposed in [3] gets on input two sets: 1) $S1$ including dependence relations originating at least two synchronization-free slices; 2) $S2$ comprising dependence relations used for inserting synchronization in code being produced on the basis of set $S1$. Using relations in set $S1$ and applying any well-known technique, for example [4], code scanning synchronization-free slices is generated. Next, using set $S2$ *send* and *receive* functions are inserted within code generated (details are presented in paper [3]).

Let us consider the following example:

Example 1:

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

Using Petit [6], we extract the following dependence relations for this loop:

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

For the input of the algorithm presented in [3], we create the following sets of relations: $S1 = \{R_1\}$ and $S2 = \{R_2\}$. Figure 1 illustrates dependences described by relations R_1 and R_2. Using set $S1$, we can extract n synchronization-free slices whose execution requires n^2 points of synchronization. However, it can be noticed that using relation $R'_2 = \{[i,1] \rightarrow [i+1,1]$: `1 <= i <= m-1`$\}$ instead of relation R_2 permits for producing valid code with much fewer points of synchronization. Relation R'_2 represents the transitive reduction of relation R_2.

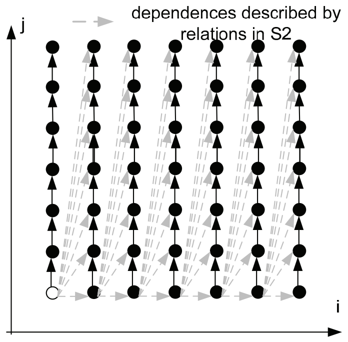

Fig. 1. Dependence graph for Example 1
Rys. 1. Graf zależności dla przykładu 1

To remove redundant synchronization, we calculate relation R as the union of all dependence relations, $R = R_1 \cup R_2$. The transitive closure, R^+, of this relation contains all pairs of iterations that are linked by a chain of synchronization of length one or more. The relation $R^+ \circ R$, which we denote R^{2+}, therefore contains all pairs that are linked by a chain of synchronization of length two or more. There is no need to explicitly synchronize dependences described by relation R^{2+}. So, the dependences that we have to explicitly synchronize are R-R^{2+}. Note, that R-R^{2+} represents the transitive reduction of relation R. This technique is called *simple redundant synchronization removal* and was presented in paper [1].

After applying this technique to the relations of Example 1 , we reduce the number of synchronization points from m^2 to m. Below, there are the results of the transitive reduction calculation.

```
R - R2+ = {[i,j,12] -> [i,j+1,12] : 1 <= i <= m && 1 <= j
          <= n-1} union {[i,1,12] -> [i+1,1,12] : 1 <= i
          <= m-1}
S1 = {R1 := {[i,j,12] -> [i,j+1,12] : 1 <= i <= m && 1 <=
     j <= n-1}}
S2 = {R2 := {[i,1,12] -> [i+1,1,12] : 1 <= i <= m-1}}
```

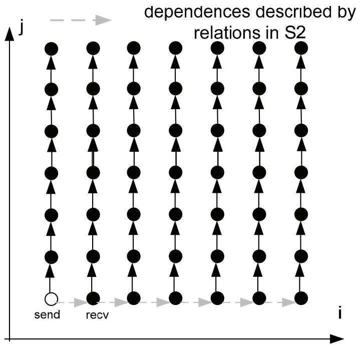

Fig. 2. Dependence graph for Example 1 after redundant dependences removal
Rys. 2. Graf zależności dla przykładu 1 po usunięciu zbytecznych zależności

Applying the algorithm presented in [3] to the sets above, we generate the following parallel code:

```
par for(t1 = 1; t1 <= m; t1++) {
    for(t2 = 1; t2 <= n; t2++) {
      if(t2==1 && 2 <= t1 && t1 <= m)
         recv(t1-1,1);
      s1(t1,t2);
      if(t2 == 1 && 1 <= t1 && t1 <= m-1)
         send(t1,1);
    }
  }
```

where `recv(), send()` are send and receive functions, respectively (their implementation is presented in paper [3]).

Let us consider another example.

Example 2:

```
for k=1 to m do
 for i = 1 to n do
  for j = 1 to n do
   c(k)=c(k) + a(i,j);
  endfor
 endfor
  b = b + c(k);
endfor
```

The loop above exposes the following dependence relations .

```
R1 = {[k,i,j,8] -> [k,i,j',8] : 1 <= j < j' <= n && 1
<= k <= m && 1 <= i <= n}
R2 = {[k,i,j,8] -> [k,i',j',8] : 1 <= i < i' <= n && 1
<= k <= m && 1 <= j <= n && 1 <= j' <= n}
R3 = {[k,i,j,8] -> [k,-1,-1,11] : 1 <= k <= m && 1 <= i
<= n && 1 <= j <= n}
R4 = {[k,-1,-1,11] -> [k',-1,-1,11] : 1 <= k < k' <= m}
```

After removing redundant dependences, applying the technique presented above, we have got the following relations.

```
R1 = {[k,i,j,8] -> [k,i,j+1,8] : 1 <= k <= m && 1 <= i
<= n && 1 <= j < n}
R2 = {[k,i,n,8] -> [k,i+1,1,8] : 1 <= k <= m && 1 <= i < n}
R3 = {[k,n,n,8] -> [k,-1,-1,11] : 1 <= k <= m}
R4 = {[k,-1,-1,11] -> [k+1,-1,-1,11] : 1 <= k < m}
```

Figure 3 presents the dependence graph for the original set of dependence relations and that after redundant dependences removal.

Next, we form the following sets.

```
S1 = {R1,R2,R3} , S2 = {R4}.
```

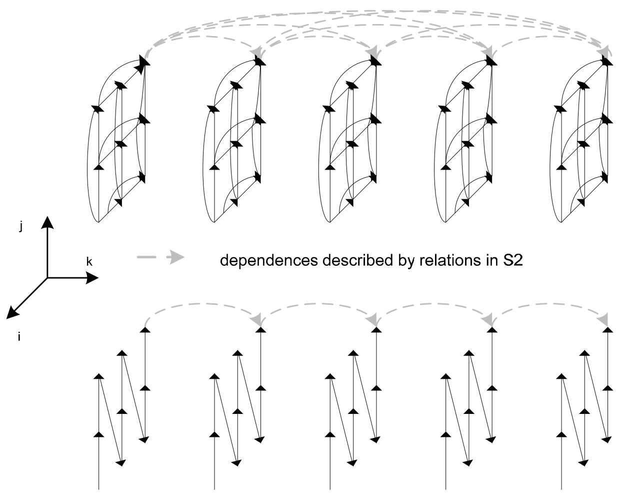

Fig. 3. Dependence graphs for Example 2 before and after redundant dependences removal
Rys. 3. Graf zależności dla przykładu 2 przed i po usunięciu zbytecznych zależności

Applying the technique presented in [3], we may conclude that due to transitive reduction we reduce the number of synchronization points from $\sum_{i=1}^{m-1} i$ to m (see Fig. 3).

Example 3:

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

For this loop, the relations representing dependences are as follows:

```
R1 =  {[i,-1,7] -> [i,1,9] : 1 <= i <= n}
R2 = {[i,j,9] -> [i,j+1,9] : 1 <= i<=n && 1<=j< n}
R3 = {[1,j,9] -> [j,-1,7] : 2 <= j <= n}
R4 = {[1,j-1,9] -> [j,-1,7] : 2 <= j < n}
```

Relations after removing redundant dependences are the following.

```
R1 = {[i,-1,7] -> [i,1,9] : 1 <= i <= n}
R2 = {[i,j,9] -> [i,j+1,9] : 1 <= i <= n && 1 <= j < n}
R3 = {[1,j,9] -> [j,-1,7] : 2 <= j <= n}
S1 = {R1,R2} , S2 = {R3}
```

Figure 4 presents the dependence graph for the original set of dependence relations and that after redundant dependence removal.

Applying transitive reduction reduces the number of synchronization points from $(n-1)^2$ to n (see Fig. 4).

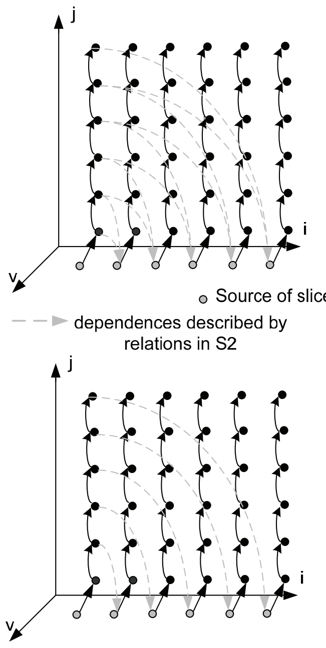

Fig. 4. Dependence graphs for Example 3 before and after redundant dependences removal
Rys. 4. Graf zależności dla przykładu 3 przed i po usunięciu zbytecznych zależności

In Table 1, we present the summary for the above examples. The second column contains the number of extracted synchronized slices. The next column presents the number of synchronization points in the code generated for original dependence relations, and the last column comprises the number of synchronization points in the code after redundant dependences removal.

Tab. 1. Summary for examined examples
Tab. 1. Wyniki dla zbiorów pętli testowych

Example	Number of synchronized slices	Number of synchronization points	
		Without transitive reduction	With transitive reduction
1	m	m^2	m
2	m	$\sum_{i=1}^{m-1} i$	m
3	n	$(n-1)^2$	n

4. Experiments

To evaluate the performance of OpenMP [11] parallel programs being formed by the proposed approach (applying the technique presented in [3] after redundant dependences removal based on transitive reduction), we have examined the three loops presented in Section 3

We have carried experiments on the machine Intel Xeon 1.6 GHz,(8 x 4-core CPU, cache 2 MB), 32 GB RAM, Linux. The results are presented in Table 2, where time is presented in seconds, *S* and *E* denote speed-up and efficiency of parallel programs, respectively. Analyzing the experimental results, we can conclude that the technique presented in this paper permits for producing effective parallel coarse-grained OpenMP programs. The values of speed-up and efficiency depend considerably on the problem size As the problem size increases, the parallel program performance increases as well.

Tab. 2. Results of experiments
Tab. 2. Wyniki dla zbiorów pętli testowych

Loop	Values of parameters	1 CPU	2 CPU			4 CPU			8 CPU		
		Time	Time	*S*	*E*	Time	*S*	*E*	Time	*S*	*E*
1	n=1000, m =100	3,468	2,234	1,552	0,776	1,574	2,203	0,551	1,441	2,407	0,301
	n=512, m=256	1,754	1,040	1,687	0,843	0,792	2,215	0,554	0,617	2,843	0,355
2	n=1000, m=128	0,589	0,311	1,894	0,947	0,232	2,539	0,635	0,213	2,765	0,346
	n=2000, m= 64	0,575	0,340	1,691	0,846	0,229	2,511	0,628	0,227	2,533	0,317
3	n=1000	0,032	0,017	1,882	0,941	0,012	2,667	0,667	0,009	3,404	0,426
	n=2000	0,127	0,079	1,608	0,804	0,036	3,528	0,882	0,026	4,885	0,611

5. Related work

Unimodular loop transformations [9], permitting the outermost loop in a nest of loops to be parallelized, find synchronization-free parallelism. However when there exist dependences between slices, that approach fails to extract coarse-grained parallelism.

The affine transformation framework (polyhedral approach), considered in many papers, for example, in papers [8, 9, 10] unifies a large number of previously proposed loop transformations. It permits for producing affine time and space partitioning and correspondent parallel code. For the presented loop examples, there does not exists any affine space partitioning permitting for extracting coarse-grained parallelism; we are able to find only time partitioning to extract fine-grained parallelism.

To demonstrate this fact, let us consider Example 1. To find an affine transformation for Example 1 we have to construct and resolve the following system of equations [8]

$$C1*(i) + C2*(j) + C == C1*(i) + C2*(j+1) + C$$
$$C1*(i) + C2*(1) + C == C1*(i+1) + C2*(j') + C$$

The solution to this system is $C1=C2=0$, C=any integer, so we can conclude that there does not exists any affine transformation extracting more than one slice. In the same way, we may demonstrate that there is no affine space transformation for Example 2 and Example 3.

Our contribution consists in demonstrating how to generate effective coarse-grained code on the basis of two sets of dependence relations, the first of them permits for extracting slices [4] while the second one is used for inserting send and receive functions to implement slices synchronization. The proposed approach extracts coarse-grained parallelism when in a loop there does not exist synchronization-free slices.

6. Conclusion and future work

We introduced the approach to extract coarse-grained parallelism in loops. Transitive closure and transitive reduction are used to generate effective parallel code. Transitive closure are applied to extract slices while transitive reduction is to considerable reduce the number of synchronization points. Results of experiments demonstrate that this number can be significantly reduced due to applying transitive reduction to a set of original dependence relations. Our future research direction is to derive techniques permitting for automatic splitting a set of dependence relations into two sets, where the first one is to form slices while the second one is to insert synchronization for proper slices execution.

7. References

[1] Kelly W., Pugh W., Rosser E., Shpeisman T.: Transitive Closure of Infinite Graphs and its Applications. International Journal of Parallel Programming. 1996, Tomy 24, n. 6, s. 579-598.
[2] Bielecki W., Pałkowski M.: Extracting coarse-grained parallelism in computer simulation applications, Advanced Computer Systems, 13th International Multi-Conference, Vol. II, Szczecin 2006, s. 237-246.
[3] Bielecki W., Pałkowski M.: Using message passing for developing coarse-grained applications in OpenMP, Proceedings of Third International Conference on Software and Data - ICSOFT 2008, Porto, Portugalia 2008, s. 145-153.
[4] Beletska A., Bielecki W., Siedlecki K., and San Pietro P.: Finding synchronization-free slices of operations in arbitrarily nested loops. In ICCSA (2), volume 5073 of Lecture Notes in Computer Science, pp. 871-886, Springer, 2008.
[5] W. Pugh and D. Wonnacott.: 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.
[6] The Omega project. http://www.cs.umd.edu/projects/omega.
[7] Kelly W., Maslov V., Pugh W., Rosser E., Shpeisman T. and Wonnacott D.: The omega library interface guide. Technical report, USA, 1995.
[8] Lim A. W., Cheong G. I., Lam M. S.: An affine partitioning algorithm to maximize parallelism and minimize communication. In ICS'99, pp.228-237. ACM Press, 1999.
[9] Banerjee U.: 1990. Unimodular transformations of double loops. In Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing, pp. 192-219.
[10] Feautrier P., 1994. Toward automatic distribution. Journal of Parallel Processing Letters 4, pp. 233-244.
[11] www.openmp.org

otrzymano / received: 14.05.2010
przyjęto do druku / accepted: 02.07.2010 *artykuł recenzowany*