A REVIEW ON DEPENDENCE ANALYSIS AND PARALLELIZATION TECHNIQUES

Vijaya Balpande
Computer Science and Engg.
Priyadarshini J.L. College of Engineering

Pradnya Borkar
Computer Science and Engg.
Priyadarshini J.L. College of Engineering

Abstract—With the new era of multicore processors it is required to improve the execution speed of the program. Consequently, developing parallel programs is the primary concern of the multicore era. The main challenge is to convert the sequential algorithm to parallel. Conversion of the sequential algorithm to parallel program without changing the output requires analyzing the set of constraints called dependency. Dependency specifies the basis for powerful transformation systems that enhance the implicit parallelism present within the program. The applications that are implemented with data structures like trees, queues and graphs is the most difficult problem for parallelization as these data structure deals with pointer. Graph Isomorphism is a technique of matching the structure of one graph with another and if only subpart of a graph is matched it is known as subgraph isomorphism. Graph and subgraph Isomorphism is one of the most highly-studied problems in the various field of computer science. In this paper we have studied the data dependency, control dependency, different parallelization techniques and the graph isomorphism problem.

Keywords—Data dependency, Control dependency, Graph Isomorphism, Subgraph Isomorphism, Parallelization

I. INTRODUCTION

With the evolution of parallel computers, the optimization process becomes too complex. The principal optimization becomes uncovering the parallelism in a sequential algorithm and tailoring the parallelism to the target machine. The basis of this approach is a dependence analysis. Dependence represents the set of constraints such as data dependence and control dependence. Data dependence constraint must ensure that the data is produced and consumed in correct order. In control dependence the order of execution of statements must be preserved depending on the condition specified in the program. For transformation, dependency analysis is a tool determining whether it is safe to transform the program that will preserve the same output of the original program. As specified in Allen et.al.(2002), a program transformation that changes the order of execution of statement without adding or deleting any statement in a program is known as reordering transformation. Reordering transformation thus preserves the relative execution order of source and sink of that dependence. For the two statements S and T, if the instance of statement T(i) depends on instance S(i) then S(i) is called source and T(i) is called sink. For loop transformation, the loop iterations are needed to be standardized and the process of standardizing the loop iteration is loop normalization. In loop normalization, the index space is transform to iteration space to have unit stride. By loop normalization the non-uniform index space is transformed to uniform iteration space. The relation between the source and sink of dependence in the iteration space can be characterized by distance and direction vectors. For loop reordering transformation, direction vector can be used to specify the relationship between the index vector of the source and sink of dependence. The direction vector specifies in which direction the loop iteration are moving and distance vector specifies that the index variable value is increasing uniformly. For checking whether the parallelization of sequential program is possible or not dependence testing is required. For determining whether two references to the same variable in a given set of loop access the same memory location, dependence testing is required. Techniques for performing dependence testing are GCD test and Banerjee Inequality test as given in Allen et.al.(2002). Dependence testing is carried out by the transformation techniques like loop normalization, constant propagation, and induction-variable. Loop normalization is performed to have uniform iteration space. For vectorization the loop interchange and wavefront method for parallelization are described in the papers of Lamport L. et al.(1974, 1976,1981). Loop skewing which is the practical way of implementing the Lamport Wavefront method was introduced by Wolfe M.J (1986). The performance of parallelized code not only depend on parallelism found in the code but also on code being packed as fine grain and coarse grain granularity. Loop distribution is performed by node splitting which breaks the data dependency cycle. Loop Fusion is
performed by merging two loops into a single loop. To apply loop fusion both loops must have same structure, loop depth, loop bounds and iteration direction. For optimization of array, Loop Methods combining loop distribution and Loop Fusion were discussed in Allen et. al.( 1987).

An application like chemical compounds, geographic maps, computer networks, software systems architectures, social networks requires the information to be represented in the form of graphs. For example, for finding the relationship of a person in a social network, requires to process huge database which can be represented in the form of graph. Comparison between graphs can be formulated as a graph or subgraph isomorphism

Parallelizing the pointer based data structure requires analyzing the dependence structure in the sequential algorithm. Dependency analysis includes data dependence, control dependence, flow dependence. For exploiting parallelism of sequential algorithm the loop structure requires rigorous analysis. Dependence in loop structure influences the parallelization of sequential algorithm.

For parallelizing the numerical and scientific applications DOALL and DOACROSS techniques Allen et. al.(2002) are used for loop parallelization and performs well for on very regular and analyzable structure that has predictable array accesses but not suitable for unpredictable data access pattern.

In this paper section II describes the dependence analysis and its properties, section III describes the existing techniques of parallelization, section IV describes graph isomorphism problem and section V includes the conclusion.

II. DEPENDENCE AND ITS PROPERTIES

A. Data Dependence:
Data dependence arises due to use of same data in more than one statement and accesses the same memory location and at least one of the statement stores data into it.
For following program code 1:

\[
\begin{align*}
S_1: & \quad r = 3.0 \\
S_2: & \quad \pi = 3.14 \\
S_3: & \quad \text{peri} = 2 * \pi * r
\end{align*}
\]

No execution constraint exist between statement \(S_1\) and \(S_2\) as execution order \(S_1, S_2, S_3\) and \(S_2, S_1, S_3\) will produce the same result.

B. Control Dependence:
A dependence that arises due to control flow is called control flow dependence.
For following program code 2:

\[
\begin{align*}
S_1: & \quad \text{if} (y! = 0) \quad \text{goto} \\
S_2: & \quad x = x / y \\
S_3: & \quad \text{continue}
\end{align*}
\]

Executing \(S_2\) before \(S_1\) could cause a divide-by-zero exception. Therefore \(S_2\) cannot be executed before \(S_1\) as \(S_2\) is conditionally dependent on \(S_1\).

C. True Dependence or Flow Dependence:
For following program code 3:

\[
\begin{align*}
S_1: & \quad y = a + b \\
S_2: & \quad z = x + y
\end{align*}
\]

The value \(y\) computed by \(S_1\) is used by \(S_2\) Dependence of \(S_2\) on \(S_1\) is called true dependence or flow dependence represented by \(S_1 \delta^{-1} S_2\). In \(S_1\), \(y\) is used as an output variable and in \(S_2\), \(y\) is used as an input variable. Flow dependence is same as read after write (RAW) hazard.

D. Antidependence:
For following program code 4:

\[
\begin{align*}
S_1: & \quad p = y + b \\
S_2: & \quad y = x + 3
\end{align*}
\]

The value \(y\) computed by \(S_2\) is read by \(S_1\). This prevents the interchange of \(S_1\) to \(S_2\). This type of dependence is called antidependence represented by \(S_1 \delta^{-1} S_2\) and is equivalent to write after read (WAR) hazard.

E. Output Dependence:
For following program code 5:

\[
\begin{align*}
S_1: & \quad y = a + b \\
S_2: & \quad y = x + z
\end{align*}
\]

Both the statements \(S_1\) and \(S_2\) uses \(y\) as an output variable and stores the value in the same variable. Statements \(S_1\) and \(S_2\) thus write into the same location. This type of dependence is called output dependence represented by \(S_1 \delta^{-1} S_2\) and is equivalent to write after write (WAW) hazard.

F. Dependence in Loops:
For following program code 6:

\[
\begin{align*}
& \text{DO } I = 1, \ n \\
S_1: & \quad A(I+1) = A(I) + B(I) \quad \text{ENDDO}
\end{align*}
\]

In statement \(S_1\), instance \(A(3)\) uses the value of \(A\)(computed in the previous iteration any loop iteration depends on the instance of itself executed in previous iteration. The iteration vector is denoted as

\[
i = \{i_1, i_2,...,i_n\}
\]

(1)

where \(i_k. 1 \leq k \leq n\), represents the iteration number for the loop at nesting level \(k\).
For the following program code 7:

\[
\begin{align*}
& \text{DO } I = 1, 2
\end{align*}
\]
DO J = 1, 2  
S1  
ENDDO

The set of all possible iteration vectors for a statement $S_i$ is an iteration space. The iteration space of $S_1$ is $\{(1,1),(1,2),(2,1),(2,2)\}$.

G. Reordering Transformation

For following program code 7:

$$S_1: r = 3.0 \pi $$
$$S_1: \text{peri} = 2 \pi r$$

Reordering the statement $S_1$, $S_2$, $S_3$ as $S_2$, $S_1$, $S_3$ will produce the same result. A transformation is valid if it preserves all dependences in the program. A reordering transformation changes the order of execution of the statement without adding or deleting any executions of any statements and preserves the source and sink of transformation in Allen et. al.(2002).

H. Distance and Direction Vectors

For following program code 8:

L1:  
    do $I_1 = 10, 100, 3$
L2:  
    do $I_2 = 50, 5, -2$

S:  
    X ($2I_1 - 1, I_1 + I_2$) = $Y (I_1 + I_2)$

T:  
    Z ($I_1 + I_2$) = $X (3I_2 + 1, 2I_1 + 2)$

By formulating the equation for statement S and T we get, $I_1 = i_1$, $I_2 = i_2$;

$S_1$:  
    $X (2i_1 - 1, i_1 + i_2)$

$I_1 = j_1$, $I_2 = j_2$;

$T$:  
    $X (3j_2 + 1, 2j_1 + 2)$

$$2i_1 - 1 = 3j_2 + 1$$

(2)

$$i_1 + i_2 = 2j_1 + 2$$

(3)

By rearranging the above equation 2 and 3 we get,

$$2i_1 - 3j_2 = 2$$

(4)

Constraints on $i_1$, $j_1$, $i_2$, $j_2$ will be

$I_1 = \{10, 13 \ldots 100\}$
$I_2 = \{50, 48 \ldots 6\}$

In above program segment the index space of $I_1$ and $I_2$ have an arbitrary stride so to standardize the index space to have an unit stride so that the index variable to be increase in sequential order as 0,1,2, … loop normalization is carried out. Loop normalization is performed by introducing a new variable called iteration variable and new iteration space having unit stride is found. Dependence between statement instances, $S(i)$ an instance of statement $S$ is determined by an index point $i$, and $T(j)$ the instance of statement $T$ is determined by an index point $j$ and the distance from $S(i)$ to $T(j)$ is written as distance between the source $S(i)$ and sink $T(j)$ of dependence in the iteration space of the loop nest containing the statement involved in the dependence. For loop normalization the distance and direction vector are needed to be found.

Dependence of $T$ on $S$ for the set of all pairs $(S(i), T(j))$ that satisfies the condition $i < j$ such that iteration $H(j)$ dependence on iteration $H(i)$ and the distance vector is $d$, direction vector $\sigma$ and dependence level $l$ is calculated as stated in equation 7, 8 and 9.For $i < j$ the distance vector $d(i,j)$ is defined as in Allen et. al.(2002).

$$d(i,j) = j - i$$

(7)

Since $i \leq j$, distance vector must always be lexicographically non-negative. No legal dependence can have negative distance because this would indicate that the source of the dependence was executed before sink. This means the source and sink are reversed and they are antidependent.

The direction vector $\sigma(i,j)$ is defined as

$$\sigma(i,j) = \text{sign}(d)$$

(8)

It is also specifies as $(<, =,>)$ depending on the relative values of iteration vectors $i$ and $j$.The arrow points to the loop iteration that occurs first pair of iteration vector $s$ for the source and sink of the dependence.

Direction vectors can be used as a basis for understanding loop reordering transformations because they summarize the relationship between the index vectors at the source and sink.

The dependence level $l$ is specified as

$$l = \text{lev}(d)$$

(9)

Dependence level are determined based on the value of $i$ and $j$.If $i=1$ and $j=1$ level is 1,if $i=0$ and $j=1$ then level is 2 and if both $i = j = 0$ then level is 3.At level 3 loops are totally independent. If $S_2$ depends on $S_1$ at level 1 for $(1 \leq l \leq m)$, we say that dependence of $S_2$ on $S_1$ is carried by $L_i$. Computation of distance vector, direction vector and dependence level for the program code 8 is specified.
For following program code 9:

L1:   DO I = 0, 4, 1
      DO J = 0, 4, 1
  S1:  A (I + 1, J) = B (I, J) + C
  S2:  B (I, J+1) = A (I, J+1) + 1
  S3:  D (I, J) = B (I, J+1)
  J+1) -2 ENDDO
ENDDO

For array A dependence between S1 and S2 can be specified as

\[ d(2,2) = (1,1) \]

For array B dependence between S1 and S2 can be specified as,

\[ d(2,2) = (1,2) \]

dependence coming in the node and the recurrence is broken and the code can be rewritten as

DO I = 1, N
  B (I, J+1) = B (I, J) + 1 ENDDO
ENDDO

For the coarse-grained parallelization, a parallel loop is moved to the outer most position to increase granularity and decrease synchronization overhead in Allen et. al.(2002). In loop interchange dependency between the statements changes. For the loop interchange the dependence vector must remain positive.

1. Node splitting:

Node splitting is the loop transformation technique which breaks the data dependency cycle and helps to parallelize the loop.

For following program code 11:

DO I = 1, N
  B (I) = B (I+1) + B (I)
  S1:  A (I) = B (I+1) + B (I)
  S2:  B (I+1) = C (I)
  M DO I = 1ENDDO
ENDDO

in the above loop there are two different references to B in the statement S1 and there exist anti dependence S1 \( \delta_{ij} S_2 \) and recurrence. This is removed by the technique called node splitting. Node splitting creates a copy of node from which an anti dependence emanates, if there are no dependence coming in the node and the recurrence is broken and the code can be rewritten as

DO I = 1, N
  A (I) = B (I+1)
  B (I) = B (I+1)
  S1:  A (I) = B (I+1) + B (I)
  S2:  B (I+1) = C (I)
  M DO I = 1 ENDDO
ENDDO

2. Loop Skewing:

Loop skewing is a transformation that reshapes an iteration space and expresses the existing parallelism with conventional parallel loops.

Loops L_1 and L_2 carry no dependence so they can be merged together to increase the granularity and the code can be rewritten as

M. Unimodular transformation:

Loop interchange, loop skewing and loop reversal are examples of general set of transformation known as unimodular transformation. Unimodular transformation is implemented for only perfectly nested loops. It supports goal directed parallelization strategies.
III. PARALLEZATION TECHNIQUES

For extraction of parallelism for running it on multicore processor to improve the performance of algorithm the different parallelization techniques are tiling technique given in Tan et.al. (2007) and Wolf M.E.et.al. (1991), speculative parallelization explained in Raman Arun (2012), decoupled software pipelining Cintra et.al R(2005) and speculative decoupled software pipelining Cintra et.al R(2003).Tiling technique is used to exploit the parallelism in loop by analyzing the dependencies. For implementing the tiling loop skewing, loop transformation techniques are used. Tiling increases the granularity of computation and decreases the amount of communication incurred between the processors. It improves the data locality and data reusability which result in better utilization of cache locality. As the data remain in cache for the entire iteration, moving in and out of data required for computation is reduced thereby improving the overall performance. It also improves the register reuse.

Speculative Parallelization is implemented by analyzing the values of data dependencies between the different tasks and parallelizing the sequential code.

Decoupled software pipelining exploits the fine grained level parallelism as compared to DOACROSS parallelism specified in Allen et al.(2002). In DOACROSS parallelism loops consist of dependencies among the iteration of the loop. DOACROSS parallelism is characterized by the parallel execution of parts of each loop iteration across multiple cores. Dependences are handled by forwarding values from core to core by some way, often through memory with synchronization. In DSWP, there are no restrictions like control flow should be simple, to operate only on arrays and should have regular memory access pattern that can be seen in DOACROSS.DSWP partition the loop code and particular piece of code across all iteration is executed on each core for which the core is responsible. It results into long communication latencies between threads.

The problem of dependence recurrence present in DSWP can be handled by speculative decoupled software pipelining Cintra et.al R(2003). Speculative DSWP combines the speculation and pipeline parallelism and improves the significant speedup in the presence of long inter-core communication latency.

IV. GRAPH ISOMORPHISM

Graph Isomorphism is a technique of matching the structure of one graph with another and if only subpart of a graph is matched it is known as subgraph isomorphism. Graph and subgraph Isomorphism is one of the most highly-studied problems in the various field of computer science. Graph is used for representing information in computer networks, social networking, data mining, chemical compounds, etc. For enhancing the performance of sequential algorithm and exploiting the resources of multicore conversion of sequential program to parallel program is required. Subgraph Isomorphism is found to be solved in polynomial time but graph isomorphism is found not to be NP-complete. Despite of much effort no polynomial-time algorithm for graph isomorphism has been found. Several subproblems of graph isomorphism are known to have polynomial algorithms.

The graph isomorphism is expressed as in Deo Narasingh et al.(1995) : Given two graphs G=(V1,E1) and H=(V2,E2) ,if there exist one to one mapping function f from V1 to V2 such that (i, j) ∈ E1, if and only if (f(i), f(j)) ∈ E2 . The function f is called an isomorphism from G to H. If the two graphs isomorphic to each other, it is denoted by G ≅ H.

Exact Matching and Inexact Matching are the two ways for matching the graphs. Exact graph matching is characterized by mapping between the two nodes of two graphs if there is an edge between the two nodes in the first graph, they are mapped to two nodes in the second graph that are linked by an edge by preserving the edge. The different form of exact matching is graph isomorphism, subgraph isomorphism, monomorphism or Automorphism and Maximum common subgraph (MCS). For exact matching the different techniques exist like tree search based algorithms and canonical labeling. Ullmann Algorithm in Ullmann J. R et. al.(1997) , Messmer B.T. and Bunke H.(1995) , D.G.Corneil and Gotelib(1970) and Nauty Algorithm in McKay Brendan D et. al.(1981,2004) and in are the algorithm based on tree search method and canonical labeling. In McKay Brendan D et. al.(1981,2004) algorithm based on canonical labeling is described. Canonical labeling is practically available algorithm on site of the author specified in McKay Brendan D(2004).

In Qiu et. al.(2010), for testing the runtime input graph with the model database graph vertex invariant and decision tree concept is implemented. The vertex invariants are used to partition the matrix of the graph before graph isomorphism detection. The vertex invariant property of graph, the size of decision tree is reduced as compared to Messmer et.al.(1995). The technique is similar to breadth pruning technique which reduces the size of decision tree remarkably still the time complexity is almost equivalent.

Decision tree is the most widely used method for inductive conclusion and simple method for knowledge representation.

Graph isomorphism problem can be solved by decision tree method using the vertex invariant. Parallelization of sequential algorithm of graph isomorphism can be done as it has wide variety of application.

V. CONCLUSION

With the evolution of parallel computers, the optimization process becomes too complex. The principal optimization becomes uncovering the parallelism in a sequential algorithm and tailoring the parallelism to the target machine. Mostly today’s algorithms are sequential, which basically perform operations in a sequential fashion. As the speed at which multicore processors operate has been improving at an exponential rate it is necessary to design an algorithm that specifies multiple operations. In order to solve a problem efficiently on a parallel machine, dependence analysis plays an important role in parallelizing the sequential code. Dependence analysis helps in implementing parallelization techniques.

VI. REFERENCES

Compilers for Modern Architectures: A Dependence-Based Approach, Morgan Kaufmann Publisher Inc.


