Code Optimizations for Parallelization of Programs using Data Dependence Identifier

Kavya Alluru[1]
School of Computer Science & Engineering
Vellore Institute of Technology
Chennai, India

Jeganathan L[2]
Centre for Advanced Data Science
Vellore Institute of Technology
Chennai, India

***Abstract*—In a Parallelizing Compiler, code transformations help to reduce the data dependencies and identify parallelism in a code. In our earlier paper, we proposed a model Data Dependence Identifier (DDI), in which a program P is represented as graph G_P. Using G_P, we could identify data dependencies in a program and also perform transformations like dead code elimination and constant propagation. In this paper, we present algorithms for loop invariant code motion, live range analysis, node splitting and loop fusion code transformations using DDI in polynomial time.**

***Keywords*—Automatic parallelization; parallelizing compilers; code optimizations; data dependence; loop invariant code motion; node splitting; live range analysis; loop fusion**

I. Introduction

Multicore processors have completely replaced single core processors, as a result general purpose computers became parallel systems, this change has thrown lot of challenges to software community in the effective utilization of the former. Though multiprocessing capability of operating systems improves the overall throughput of new hardware still performance of serial programs remains the same even on multicore systems. To enhance the performance of serial programs on multicore systems, instructions in the serial code has to be broken into groups such that each group cab be run in parallel. One way to accomplish this task is by manual conversion which is a tedious job. One more way is to use a tool that converts serial program to parallel.

Automating the process of serial to parallel conversion is called as Automatic Parallelization and the compiler which can perform automatic parallelization is typically referred as Parallelizing Compiler. The general process of serial to parallel program conversion is a three step one: 1) perform code transformations in order to detect potential parallelism; 2) check for data dependencies in the code; 3) generate parallel code.

Two instructions I_1 and I_2 in a program are said to be *data dependent* if both the instructions access same memory location. Presence of data dependencies makes parallelism an impossible task. Code transformations help to eliminate some of the data dependencies thereby giving a scope to detect potential parallelism.

In our earlier paper [1], we proposed a model called Data Dependence Identifier(DDI) which can identify data dependencies in scalars, arrays, and pointers in a program. We also discussed how code optimizations like dead code elimination, constant propagation can be performed using DDI. In this paper, we discussed how code optimizations like loop invariant code motion, live range analysis and node splitting, loop fusion are performed using our model DDI.

II. Related Works

Compiler converts source code to Intermediate Representation (IR) to perform code optimizations. This IR may differ from compiler to compiler. Generally, in traditional compilers for uniprocessor systems, instructions in source code are intermediately represented in three address code format and Directed Acyclic Graph (DAG), code optimizations are performed using this Intermediate Representation [3].

Intermediate Representation is crucial for a parallelizing compiler. Here, we will discuss in brief about some of the parallelizing compilers and their Intermediate Representations.

- SUIF (Stanford University Intermediate Format): SUIF is a source to source parallelizing compiler that takes C or FORTRAN serial code as input and produces parallelized code to be run on a multiprocessor machine. SUIF intermediate representation is a language-independent abstract syntax tree. Data flow analysis, data dependence analysis, scalar and array privatization, reduction variable analysis are performed using IR [4], [5], [6].
- Cetus: Cetus converts serial program written in C to parallel C program by inserting OpenMP annotations to be run on a multicore system. Cetus intermediate representation is a hierarchial tree based structure implemented in Java. Cetus IR includes a set of iterators that traverses through the IR to get the required information about loops, conditional statements, etc. Data dependence analysis - GCD Test [7] and Range Test [8] are used to identify data dependencies in arrays. Transformation techniques like scalar and array privatization, induction variable substitution, reduction variable recognition are performed using IR to eliminate some of the dependencies [9].
- Pluto: Pluto is a source to source compiler that transforms serial C program to OpenMP C [10]. Intermediate Representation of Pluto is based on polyhedral model. Dependence analysis, loop transformations for parallelism and optimized data locality are performed

using IR [11]. Optimizations based on polyhedral model are integrated in compilers like GCC and LLVM. State-of-art in Pluto includes loop fusion transformation using Fusion Conflict Graphs (FCG) [12] and verified code generation [13].

- Intel compiler [16] automatically identify the loops that can be parallelized and partitions the data accordingly.

Using our proposed model DDI, we have shown how data dependence analysis can be performed. We are broadening the scope of our model by showing how code transformations like loop invariant code motion, live range analysis and node splitting, loop fusion can be applied on DDI.

III. Data Dependence Identifier

In this section we discuss in brief about our model Data Dependence Identifier(DDI) which we have proposed in our earlier paper [1]. The main objective of DDI model is to represent a program as graph to identify data dependencies in a program. Though many graphical representation of program exists [14], [15], our representation takes a completely different perspective, we consider variables in the program as nodes and the edges between these variables are drawn based on the mode of access of variables from memory. For this purpose, we have categorized the instructions in a program and parameterized program as discussed in sections A and B.

A. Categorization of Instructions based on Memory Accessibility

We categorized the instructions in a program broadly into **Memory Access Instruction (MAI)** and **Non Memory Access Instruction (NMAI)** based on the way they access the memory. In MAI, instructions access the memory to perform the required operation. Instructions like arithmetic, conditional fall under this category. In NMAI, instructions do not access the memory at all i.e., instructions like jump, break come under this category.

MA Instructions are further classified into three categories: MA-READ, MA-WRITE, MA-READWRITE. In **MA-READWRITE(MARW)**, instructions access the memory for both read as well as write operations. For example, in Arithmetic instruction: '$c = a + b'$, data is read from memory locations a and b and written to a memory location c. In **MA-READ(MAR)**, instructions perform only read operation but no write operation. For example, in conditional instruction: '$if(a > b)'$ data is only read from memory locations a and b but the output is not written to any variable. Generally in these instructions data is read from memory and send to other Hardware Units(HU) in the computer system like processor or output devices. In **MA-WRITE(MAW)**, instructions perform only write operation but no read operation. For example, in assignment instruction '$a = 5'$, a constant value is written to a memory location a. Here we assume, $a = 5$ means that the constant 5 is read from the programmer(PR) and written to the location a.

B. Parameterization of Program

A program P is parameterized with I, V, W, HU, PR, where:

- Set I, finite set of instructions $\{i_1, i_2, ...i_n\}$
- Set V, finite set of memory allocations or variables $\{v_1, v_2, ...v_p\}$
- Set MAI, finite set of MA instructions where $MAI \subseteq I$.
  An instruction $i \in MAI$ is represented as ordered pair $[R, W]$ where $R, W \subseteq V$. R is a set that contains all the variables from which the instruction i reads the data and W is a set with a single variable to which i writes the data. For example, instruction '$c = a + b'$ is written as pair $[\{a, b\}, \{c\}]$ where data is read from variables a , b and output is written to c.
- HU represents the set of hardware units i.e. input devices, output devices, processor and any other hardware unit in the computer system.
- PR is the set of constant values initialized in the program P by the programmer.

Therefore, we write P as $P(I, V \cup \{HU, PR\}, MAI)$.

C. Directed Graph Representation of a Program

In DDI model, we represent a program as graph. Here, we discuss how a program P is transformed to an equivalent directed graph called graph of P written as G_P.

All instructions in a given program P are indexed sequentially with the positive integers $1, 2, ...n$. First instruction in a program is indexed as 1, second instruction as 2 and so on. For $i_n \in I$, we call index of $i_n = n$. Every instruction i_n is written as the pair $[R, W]$. In other words, $index[i_n] = index([R, W]) = n$.

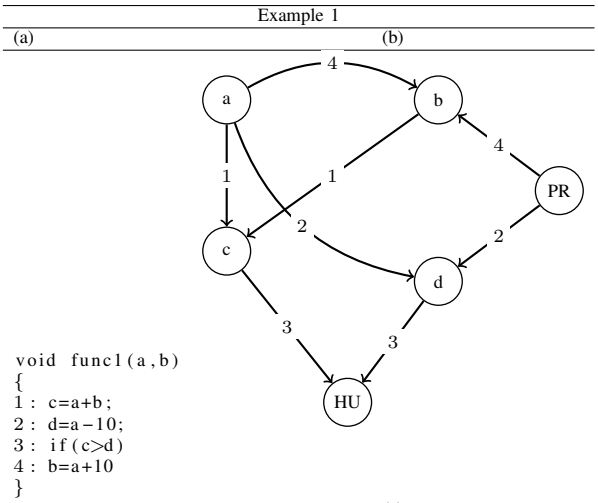
A program $P = (I, V \cup \{PR, HU\}, MAI)$ is transformed into a directed labeled graph $G_P = (V \cup \{PR, HU\}, E, L)$ as follows:

- Set of nodes of G_P are the set of variables $V \cup \{PR, HU\}$.
- For every ordered pair of sets $(R, W) \in MAI$, we include the edges $\{[r, w])|\forall r \in R, w \in W\}$.
- Every edge in G_P is labeled with elements from label set S which contains indices of instructions in I. $L : E \rightarrow \{1, 2, ...n\}$ such that $L((r, w)) = k$ if $index([R, W]) = k$ such that $(r, w) \in E, r \in R, w \in W$.

We use the notation $(.,.)$ to represent the edges of the graph and $[.,.]$ indicates the pair of sets R,W for representing memory access instructions.

In example 1, $I = \{i_1, i_2, i_3, i_4\}$, $V = \{a, b, c, d\}$ and MAI=I as all instructions in program P are memory access instructions. To construct G_P, V acts as nodes N. For instruction $1 : [\{a, b\}, \{c\}]$, we include the edges (a, c) and (b, c) with labels $L((a, c)) = 1$ and $L((b, c)) = 1$ are added to G_P. For instruction $2 : [\{a, PR\}, \{d\}]$, edges (a, d) and

(PR, d) with labels $L((a, d)) = 2$ and $L((PR, d)) = 2$ are added. For instruction $3 : [\{c, d\}, \{HU\}]$, edges with label $L((a, HU)) = 3$ and $L((d, HU)) = 3$ are added. For instruction $4 : [\{a, PR\}, \{b\}]$, edges with label $L((a, b)) = 4$ and $L((PR, b)) = 4$ are added to G_P. The adjacency matrix



(c)

	a	b	c	d	PR	HU
a		4	1	2		
b			1			
c						3
d						3
PR		4		2		
HU						

of graph G_P is shown in example 1(c), rows gives the ***read*** information about the variables and columns gives the ***write*** information. Scanning column c of the matrix tells that variable c is accessed for '$write'$ in instruction 1 and row of c shows variable c is accessed for '$Read'$ in instruction 3.

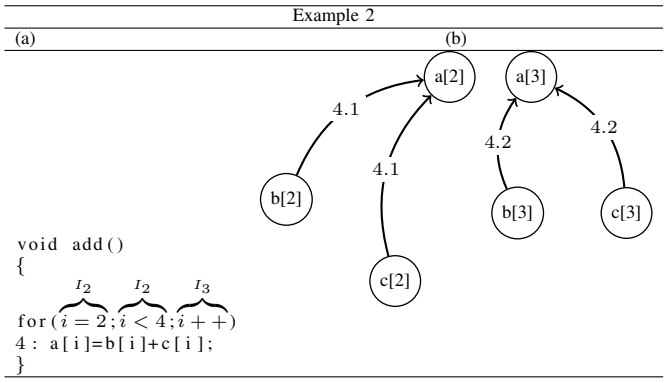
The procedure by which we convert $P(I, V \cup \{PR, HU\}, W)$ into a simple edge labeled graph $G_p(N \cup \{PR, HU\}, E, L)$ is discussed in algorithm 1.

Algorithm 1 Convert Program P to Directed edge-labeled graph G_P

```
1: procedure PROGRAM TO GRAPH
   Input: Program P(I, V ∪ {PR, HU})
   Output: Graph G_p(N ∪ {PR, HU}, E, L)
2:     for each instruction [R, W] ∈ I, index[R, W] = k do
3:         if MAI-verification() then
4:             for every r ∈ R and w ∈ W do
5:                 E = E ∪ {(r, w)}
6:                 L([r, w]) = k
```

Loop representation in DDI:
The statements within the loop are denoted as $i.k$, where $i.k$ represents instruction i when the loop is executed k^{th} time.

Nested loop representation in DDI:
Consider nested loops L_1 and L_2, where L_1 is the outer loop and L_2 is the inner loop. The statements within the nested loop are denoted as $m.x.p$, where $m.x.p$ represents m^{th} instruction when the loop is executed x^{th} iteration in L_1 loop and p^{th} iteration in L_2 loop.



Consider the nested sequence of loops $L_1, L_2, L_3, ...L_n$, where L_1 is the outermost loop and L_n is the innermost loop. The statements within this nested loop are denoted as $m.x_1.x_2....x_n$, where m represents the instruction number and x_1 represents the instance of iteration of outermost loop L_1, x_2 represents the instance of iteration of loop L_2, and x_n represents the instance of execution of innermost loop L_n.

IV. Applications of DDI

In our earlier paper [1], we proposed how compiler optimizations like constant propagation, dead code elimination and induction variable detection can be performed using our DDI model. In this section, we will discuss how optimizations like loop invariant code motion, live range analysis, loop fusion, scalar privatization can be performed using our DDI model.

A. Loop Invariant Code Motion

A set of statement(s) within a loop is called as Loop Invariant Code if the semantics of the program is not affected when the statements are moved out of the loop. Identifying and removing invariant code loop reduces the number of statements within the loop, thereby enhancing the performance of the parallel loop. Code Motion is the process of moving the loop invariant code outside the loop. In the program given in example 3(a), value of x in instruction 5 remains unchanged through out the execution of loop. Even if instruction 5 is moved above the loop, value of x remains the same.

Following observation is made to identify loop invariant code:

- For an instruction $i : [R, W]$, the value of the variable 'W' will get updated during the loop iteration if atleast one of the values of variables in 'R' is changing during the execution of the loop. In example 3(a), in instruction $5 : [\{t, PR\}, \{x\}]$, input variables are $\{t, PR\}$, t is the only input variable here as PR is a constant value and variable t never gets updated in the loop. As t value never changes during the execution of the loop, consequently there is no change in x. We call statement 5 as 'Loop Invariant Code'.

Theorem 1. *Given a loop l with statements $\{i_1, i_2, ...i_s\}$ and G_l be the graph that corresponds to the loop l. A statement $l_k \in l$ is said to be **Loop Invariant Code** if*

1. There exists a node $u \in N.G_l$ such that $L((u', u)) = [i_k.1, i_k.2, ...i_k.m]$ where m is the number of iterations of the loop.

2. There is no edge (v, u') for every $u \in N.G_l$.

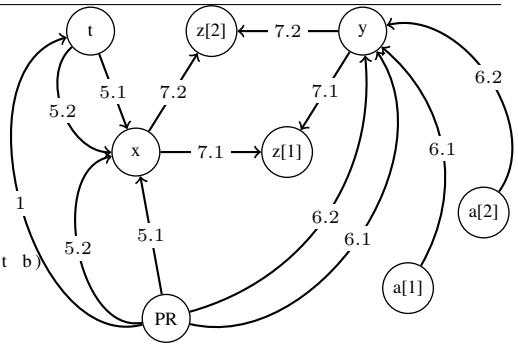
Proof: i_k is 'Loop Invariant Code'

$\implies$ If $i_k : [R, W]$, then $\exists\ u \in W$ where the value of u does not change through out the loop.

$\implies$ $u \in W$ implies that there exists $u_k \in R$ such that value of u_k is written in u and there is no change in u_k through out the loop.

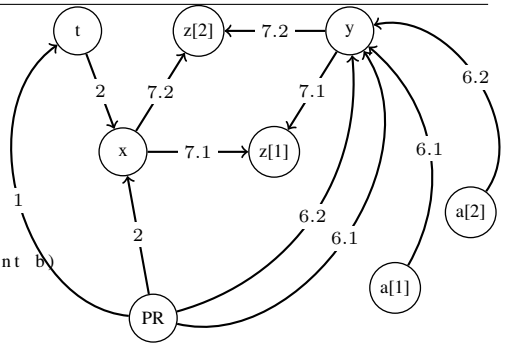
$\implies$ By algorithm, G_l will have the edges (u_k, u) and there will not be any edge of the form (v, u_k) where $u, u_k \in N.G_l$, since the nodes of G_l pertains to the variable inside the loop. ■

Example 3(a)
Before Code Motion




```
void func1(int a,int b)
{
1: t=10
      2     3     4
for(i=1;i<3;i++)
5: x=t*10;
6: y=20+a[i];
7: z[i]=x+y;
}
```

Example 3(b)
After Code Motion




```
void func1(int a,int b)
{
1: t=10
2: x=t*10;
      3     4     5
for(i=1;i<3;i++)
6: a=20+a[i];
7: z[i]=x+y;
}
```

In example 4(a), Loop Statements(LS)={5,6,7} for node x, $L((t, x)) = [5.1, 5.2]$ and $L((PR, x)) = [5.1, 5.2]$, there exists no other edges to node x with label 5. Only source of input to node x with label 5 is from nodes t and PR. As PR is constant value, only input is node t. Incoming edge to node t is $L((PR, t)) = 1$, $1 \notin LS$. Therefore, we conclude statement 5 is loop invariant code.

Algorithm illustrates the process of loop invariant code detection. Line 3-6 of the algorithm examines if node u have any incoming edges from nodes $u, u_1, u_2, ..u_k$ with labels matching the pattern $[n.1, n.2, ...n.m]$. If so, then line 8-10 checks if there are any incoming edges to nodes $u, u_1, u_2, ..u_k$ with label l where $l \in LS$, LS contains loop statement labels.If no such edges exist, instruction n is considered as loop invariant code, move instruction n above the loop in the program. Line 8-10, perform code motion. Example 3(b) shows the program and graph after code motion.

Algorithm 2 Loop invariant code detection

```
1:  procedure LOOP INVARIANT CODE DETECTION
     Input: Graph G_p(N, E, L)
2:     edgelabels=FALSE,invariant=TRUE
3:     LS={i_1, i_2, ..i_s}
4:     for every u ∈ N.G do
5:         if L((u_i, u)) == [n.1, n.2, ...n.m] then
6:             if L((v_i, u_i)) ∈ LS then
7:                 invariant=FALSE
8:         if invariant==TRUE then ▷ perform Code Motion
9:             delete edges L((u_i, u)) = [n.1, n.2, ...n.m]
10:            add edge L((u_i, u)) = p, p < l_1
```

B. Live Range Analysis

For parallelizing a program, statements in the program has to be grouped in such a way that the statements in these groups can be executed in parallel and gives the same output as sequential execution. one way to accomplish this task is using the live range information of variables in the program. We define the *live range* of a variable in a program as follows:

Definition IV.1. A variable u is said to $live$ in statement k of program P if either $Read$ or $Write$ operation is performed on u.

Consider the program in example 5, y is *live* in statement 2 and *not live* in statements 1,3,4.

Definition IV.2. Live Range Analysis of a program P is a description which provides an information on the nature of the variable, whether live or not, in each of the instruction of program P.

In example 4(a), x is live in statements 1,2,3,4. y is live in statement 2. k is live in statement 1. a is live in statement 4. This information is represented in the form: $x : \{1, 2, 3, 4\}$, $y : \{2\}$, $k : \{1\}$, $z : \{3\}$, $a : \{4\}$, which is usually referred as live range analysis of P.

Now, we propose a method to compute live range of variables in a program using our DDI model.

Theorem 2. *Given a program P and the corresponding graph G_P. If node $u \in G_P$ have either incoming and outgoing edges with labels i_k then variable u is said to be* ***live*** *in statements i_k of program P.*

Proof: u is live in instruction i_k.

$\implies$ By definition IV.1, either $Read$ or $Write$ operation is performed on u in i_k. $\implies$ By algorithm 1, there will be

an outgoing edge with label i_k from u (if Write operation is performed over u in i_k) or there will be an incoming edge with label i_k to u (if Read operation is performed over u in i_k).

$\Longrightarrow$ There is an edge incident on u with label i_k. ■

Based on the above theorem, we propose an algorithm to compute live range of variables in a program.

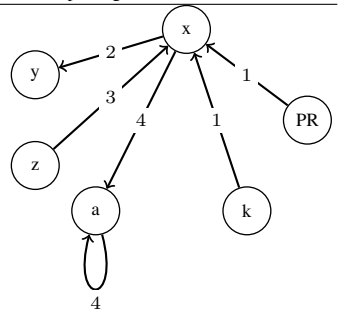
In Line 2 of algorithm 3, A is initialized as an empty two

Algorithm 3 Live Range Analysis

```
1: procedure LIVE RANGE ANALYSIS
   Input: Graph G_p(N, E, L)
2:     A=[ ]
3:     i=0
4:     for every u ∈ N.G do
5:         A[i].append(u, L((v, u)), L((u, u')))
6:         i=i+1
7:     return A
```

dimensional array. The for loop in lines 4-6 examines all the incoming and outgoing edge labels of each node and assigns this information to A. Each row of array A have node label u, the incoming and outgoing edge labels of node u. With assumption the graph is represented using adjacency matrix, the running time of this algorithm is $O(n^2)$, where n is the number of nodes in the graph. The for loop requires scanning the row and column of each and every node, therefore the complexity $O(n^2)$.

Example 4(a)
Before Node Splitting

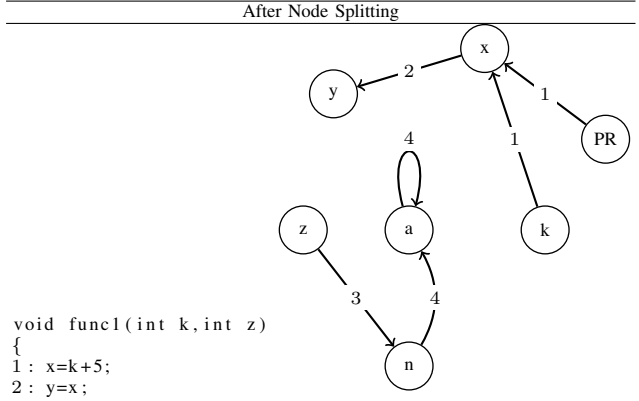



```
void func1(int k,int z)
{
1: x=k+5;
2: y=x;
3: x=z;
4: a=a+x;
}
```

C. Node Splitting

If a variable is *live* through out the program means there exists data dependence among the statements. The data dependence has to be broken in order to group the statements such that each group can execute in parallel. One approach to break the data dependence cycle is using **Node Splitting**. Node splitting creates one more copy of a node(duplicate node) in the graph and divides the edges between two nodes to produce an analogous graph. This transformation limits the live range of a variable to a section in the code hence producing a code more feasible for parallelization. Consider the program in example 4, variable x is live in instructions $\{1, 2, 3, 4\}$, after splitting x as x and n, x is live in instructions $\{1, 2\}$ and n in $\{3, 4\}$.

Example 4(b)
After Node Splitting




```
void func1(int k,int z)
{
1: x=k+5;
2: y=x;
3: n=z;
4: a=a+n;
}
```

Given a variable v, the possible sequence of *Read(R)* and *Write(W)* operations on v are 1. $\{W, R, R, R..R\}$ - value assigned to v is only *Read* through out the program. 2. $\{W, R, W, R..R\}$ - variable v is updated multiple times in the program. Based on this observation, we define the scope of node splitting as follows:

Definition IV.3. A node $u \in V.G_P$ is said to be a splitting node if the sub-graph that involves u can be split into two sub-graphs G_{P_1} and G_{P_2} such that functionality of both the programs P_1 and P_2 is equivalent to the functionality of P.

Theorem 3. *Let P be a program, G_P be the graph that corresponds to P. A node $u \in V.G_P$ is a splitting node of G_P if and only if $\exists$ a sub-graph ${G_P}^u$ that involves the node u as follows.*

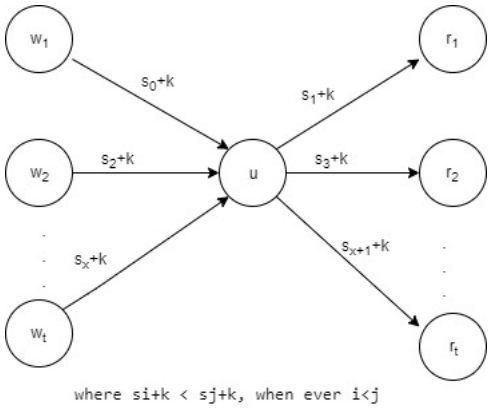


Fig. 1. u is Splitting Node of G_P.

Proof:

Hypothesis: u is a splitting node of G_P.

Claim: $\exists$ a sub-graph ${G_P}^u$ of G_P as shown in fig1.

Hypothesis: ${G_P}^u$ can be split into two sub-graphs ${G_{P_1}}^u$ and ${G_{P_2}}^u$ such that the functionality of P_1 and P_2 is equivalent to P.

$\implies$ $\exists$ a program P in which variable u is used more than once(t_1 times) for writing and u is used more than once (t_2 times) such that $t_1 \geq t_2$.

without loosing any generality, assume $t_1 = t_2 = t$

$\implies$ u is used t times for writing and u is used t times for reading purpose.

$\implies$ Again, without loss of any generality, for every *writing* to u, we have a *reading* from u.

$\implies$ Since u is a splitting node, we have a sequence of t blocks in P such that in each block, u is written first and then u is read.

$\implies$ A snippet of P that involves u will look as follows:

block 1 $\begin{cases} s_0 + k : u = w_1 + a_1 \\ ..... \\ s_1 + k : r_1 = u \end{cases}$

block 2 $\begin{cases} s_2 + k : u = w_2 + a_2 \\ ..... \\ s_3 + k : r_2 = u \end{cases}$

.

.

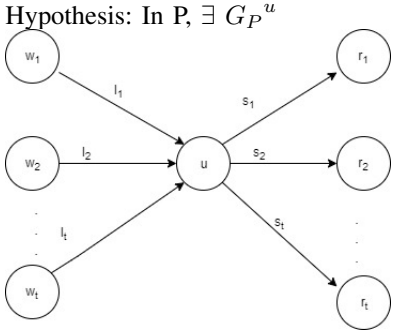
.

block t $\begin{cases} s_x + k : u = w_t + a_t \\ s_{x+1} : r_t = u \end{cases}$

Corresponding G_P will be as shown in fig.1, hence the claim.

Part II proof:

Hypothesis: In P, $\exists$ ${G_P}^u$

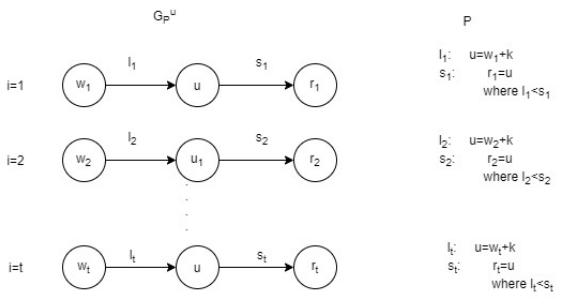


$l_i < s_i$

Claim: u is a splitting node.

Hypothesis: P has a sequence of t blocks and in each block, value is read from u after a value is written to u.

$\implies$ ${G_P}^u$ can be split as follows



In each P_i, a value is written to u first and then u is read.

$\implies$ From ${G_P}^u$, we infer that in P, value is written first and then read next.

$\implies$ By sheer observations, we infer that the total functionality of the snippets $P_i (i = 1 \ to \ t)$ is same as the functionality of P. The functionality of other statements (which does not involve u) in P remains as such in P_i also.

$\implies$ We have a node u in ${G_P}^u$ which can be split into a sequence of sub-graphs ${G_P}^u$, $i = 1, 2, .., t$ such that the total functionality of P_i, $i = 1, 2, .., t$ is equivalent to the functionality of P.

$\implies$ u is a splitting node of G_P. ■

Corollary 3.1. *Let P be a program. Let G_P be the graph that corresponds to P. P is parallelizable if and only if G_P has atleast one splitting node.*

Algorithm 4 Node Splitting

```
1: procedure NODE SPLITTING
   Input: Graph G_p(N, E, L)
2:     for every u ∈ N.G do
3:         if L((u', u)) = m and L((u'', u)) = n and n > m
   then
4:             add node w
5:             delete edge L((u'', u)) = n
6:             add edge L((u'', w)) = n
7:             for every edge (L((u, v)) > n) do
8:                 delete L((u, v)) and add L((w, v))
```

Let M be the adjacency matrix representation of G_P. In algorithm, lines 2 requires scanning each and every column of M to check if there exists any node u which satisfies the condition in line 3. Lines 4-6 i.e. adding a new node and edges takes *constant time*. In line 3, if a node u meet the condition then in lines 7-8 the entire row of the node u has to be examined. Let's say if there are n nodes among which m nodes satisfy the condition in line 3, then the complexity is $O(mn)$.

D. Loop Fusion

Loop fusion is a technique in which two loops are merged or fused to form a single loop. Generally, a loop iterates through the same set of instructions to perform a task. Two loops $L1$ and $L2$ can be fused if number of iterations, terminating conditions of both the loops match and the semantics of the code be intact after merging. Fusing of loops reduces the number of loops present in a program thereby mitigating the overhead involved in parallelization of many loops.

Definition IV.4. Loop Fusion: is a technique by which statements of multiple loops are merged into a single loop such that semantics of the code is intact.

Consider $L1$ be the first loop and $L2$ be the second loop in sequence, then $L1$ and $L2$ can be fused if the following conditions are satisfied:

- Loops $L1$ and $L2$ should have same looping conditions and should iterate for same number of times.
- Dependencies that exist between statements of loop $L1$ and $L2$ does not change the semantics of the code.

So, concept of fusion depends on the dependencies that exist between the loops. Hence, first we discuss different dependencies that exist between the loops. Two loops $L1$ and $L2$

are said to be data dependent if dependence exists between any statement of $L1$ and any statement of $L2$. Let statements $S_i \in L1$ and $S_j \in L2$. The following dependencies may exist between S_i and S_j:

Definition IV.5. No dependence: $L1$ and $L2$ are said to have no dependence if the statements S_i and S_j do not access any common memory location.

Definition IV.6. Flow dependence: If memory location M is accessed for 'Write' operation in statement S_i and the same location M is accessed for 'Read' in statement S_j. Then, flow dependence exist between statements S_i and S_j.

In example 6(case i), 'Write' operation is performed on an index location in array A in first loop and is 'Read' from the same index location in array A in second loop. So, there exist flow dependence between loops $L1$ and $L2$.

Definition IV.7. Anti dependence: If memory location M is accessed for 'Read' operation in statement S_i and the same location M is accessed for 'Write' in statement S_j. Then, anti dependence exist between statements S_i and S_j.

In example 7(case i), 'Read' operation is performed on an index location in array x in first loop and 'Write' operation on the same index location of array x in second loop, there exist Anti dependence between loops $L1$ and $L2$.

Definition IV.8. Loop carried forward dependence: If memory location M is accessed by an iteration of statement S_i and then the same location M is accessed in later iterations of statement S_j. Then, loop carried forward dependence exists between statements S_i and S_j.

In example 8, A[1] value computed in first iteration of first loop is read in the second iteration of second loop, shows existence of loop carried forward dependence between $L1$ and $L2$.

Definition IV.9. Loop carried backward dependence: If memory location M is accessed in an iteration of statement S_j and then the same location M is accessed in later iterations of statement S_i. Then, loop carried backward dependence exists between statements S_i and S_j. In example 9, A[2] value computed in second iteration of first loop is read in the first iteration of second loop, shows presence of loop carried backward dependence.

Identification of data dependencies using DDI and feasibility of fusion

So far we have discussed the dependencies that exist between the loops. Here, we will discuss the type of dependencies between $L1$ and $L2$ which does not affect the fusion of $L1$ and $L2$. We propose four theorems with which we can identify the type of dependence that exist between $L1$ and $L2$ using our DDI and the feasibility of fusing them.

Theorem 4. *Given program P with loops $L1$ with statements $\{i_{r_1}, i_{r_2}, ...i_{r_n}\}$ and loop $L2$ with statements $\{i_{s_1}, i_{s_2}, ...i_{s_m}\}$. Let G_P be the corresponding graph of P, with G_{L1} and G_{L2} as the sub-graphs of G_P that corresponds to loops $L1$ and $L2$ respectively. $L1$ and $L2$ is said to have **no dependence** if there exists no edge (u, v), $\forall u \in G_{L1}$ and $\forall v \in G_{L2}$. Such loops $L1$ and $L2$ can be fused.*

Proof: Assume that there exists an edge (u, v) where $u \in V(G_{L1})$, $v \in V(G_{L2})$.

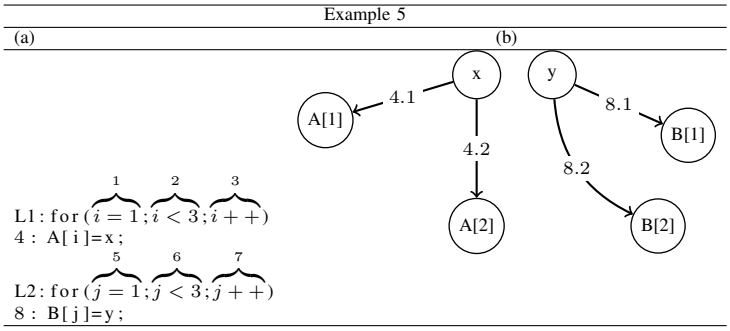
$\implies$ An edge (u, v) in G_{L1} means that a value is Read from variable u in some statement i_{r_j} (say) and Written to variable v. As $v \in G_{L2}$, v is accessed by some statement i_{s_k} (say) in $L2$.

$\implies$ By Algorithm 1, there exists a statement i_{r_j} in L_1 which Read's a value from variable u and that value is Written to a variable v in statement i_{s_k} in L_2.

$\implies$ Thus, we have proved that if there exists an edge (u, v) with label i_{r_j} then there exists a statement i_{r_j} which accesses the variables u and v .

$\implies$ By considering contrapositive statement of above proposition i.e, if there does not exist a statement i_{r_j} which accesses the variables u and v then there does not exist edge (u, v). $\implies$ There exists no common variable accessed by statements of $L1$ and $L2$. Therefore, by Definition 4.5, if statements of $L1$ and $L2$ do not access common memory location then there exists no dependence between statement of loops $L1$ and $L2$.

■



In Example 5, nodes $\{x, A[1], A[2]\} \in L1$ and nodes $\{y, B[1], B[2]\} \in L2$, there exists no common nodes among $L1$ and $L2$, no edges between nodes of $L1$ and $L2$. Therefore, no dependence exist between the two loops, in which case merging of loops is possible.

Theorem 5. *Given program P with loops $L1$ with statements $\{i_{r_1}, i_{r_2}, ...i_{r_n}\}$ and $L2$ with statements $\{i_{s_1}, i_{s_2}, ...i_{s_m}\}$. G_P be the corresponding graph of P, G_{L1} and G_{L2} are the sub-graphs of G_P that corresponds to loops $L1$ and $L2$ respectively. Let there exist edges $L((u, v)) = i_{r_j}.n$ and $L((v, w) = i_{s_k}.m$ such that $i_{r_j} \in L1$, $i_{s_k} \in L2$. i*

1) *$L1$ and $L2$ is said to have **flow dependence** if $i_{s_k} > i_{r_j}$.*
2) *$L1$ and $L2$ can be fused if $m \geq n$ and $i_{s_k} > i_{r_j}$.*
3) *$L1$ and $L2$ can not be fused if $n > m$.*

Proof: Let G_{L1} and G_{L2} be the sub-graphs of G_P that corresponds to loops $L1$ and $L2$ in P and there exist edges $L((u, v)) = i_{r_j}.n$ and $L((v, w) = i_{s_k}.m$ in G_P.

Hypothesis 1: There is a flow dependence if $i_{s_k} > i_{r_j}$.

$\implies$ An incoming edge with label $i_{r_j}.n$ to node v means variable v is Written in nth iteration of instruction i_{r_j}. An outgoing edge with label $i_{s_k}.m$ to node v means variable v is Read in mth iteration of instruction i_{s_k}.

$\implies$ The condition $i_{s_k} > i_{r_j}$ means that first a value is Written to v in nth iteration of instruction i_{r_j} and then Read from v in mth iteration of instruction i_{s_k}.

$\implies$ By Definition IV.6, there exists flow dependence between statements i_{s_k} and i_{r_j} if Read operation succeeds Write operation.

Hypothesis 2: $L1$ and $L2$ can be fused if $m \geq n$ and $i_{s_k} > i_{r_j}$.

$\implies$ As $m \geq n$ and $i_{s_k} > i_{r_j}$, even after fusing the statements i_{s_k} and i_{r_j} as Read operation succeeds Write operation on variable v, semantics of code is unchanged.

$\implies$ By definition IV.4, statements of loops $L1$ and $L2$ can be fused if the semantics of the code is intact.

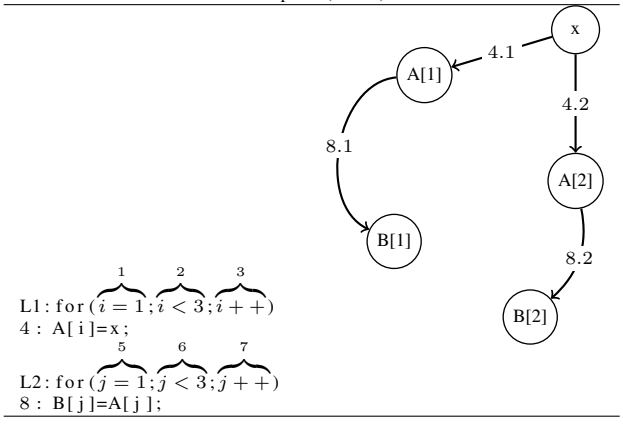
Hypothesis 3: $L1$ and $L2$ can not be fused if $n > m$.

$\implies$ Before fusing, in loop $L1$ write operation is performed on variable v in nth iteration of instruction i_{r_j}. In loop $L2$ variable v is Read in mth iteration of instruction i_{s_k}.

$\implies$ If $L1$ and $L2$ are fused, as $n > m$, variable v is read in mth iteration of instruction i_{s_k} even before v is written in nth iteration of instruction i_{r_j} i.e., older value of variable v is read not the updated.

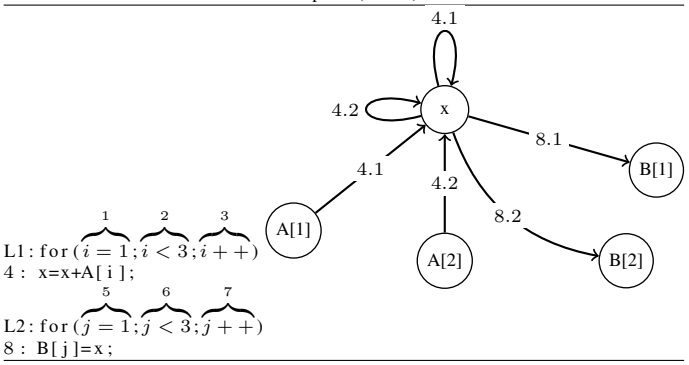
$\implies$ As semantics of code changes, loop fusion is not possible if $n > m$. ■

Example 6 (Case i)



If flow dependence exists between loops $L1$ and $L2$ i.e., if an instruction in $L1$ access a memory location M for 'Write' and the same location is accessed by an instruction in loop $L2$ for 'Read' then merging of loops is possible if M is accessed for 'Write' and then for 'Read' even after fusion. In example 6(case i), $L((x, A[1])) = 4.1$ and $L((A[1], B[1])) = 8.1$ says $A[1]$ is updated in iteration 1 of instruction 4 and read in iteration 1 of instruction 8. As a value is updated in first loop and read in second loop in the same iteration, merging of loops will not change the semantics of code. Therefore edges $L((u, v)) = n.i$ *and* $L((v, w) = m.j$ where $n \in L1$, $m \in L2$ and $j \geq i$ in the graph represents flow dependence where merging of loops is possible.

Example 6 (case ii)



If flow dependence exists between loops $L1$ and $L2$ merging of loops is *not* possible if on fusing of loops 'Write' operation succeeds 'Read' on memory location M, which changes the semantics of the code. In example 6(case ii), x is accessed for 'Write' in first loop and for 'Read' in second loop, which shows flow dependence. $L((A[2], x)) = 4.2$ and $L((x, B[1])) = 8.1$ says, value of variable x to be written in iteration 2 of instruction 4 is read in iteration 1 of instruction 8 i.e., value of x is read even before write operation. Therefore, merging of loops $L1$ and $L2$ changes the semantics of the code.

Theorem 6. *Given program P with loops $L1$ with statements $\{i_{r_1}, i_{r_2}, ...i_{r_n}\}$ and loop $L2$ with statements $\{i_{s_1}, i_{s_2}, ...i_{s_m}\}$. G_P be the corresponding graph of P, G_{L1} and G_{L2} are the sub-graphs of G_P that corresponds to loops $L1$ and $L2$ respectively. Let there exist edges $L((u, v)) = i_{r_j}.n$ and $L((w, u)) = i_{s_k}.m$ such that $i_{r_j} \in L1$, $i_{s_k} \in L2$. i*

1) *$L1$ and $L2$ is said to have **Anti dependence** if $i_{s_k} > i_{r_j}$.*
2) *$L1$ and $L2$ can be fused if $m \geq n$ and $i_{s_k} > i_{r_j}$.*
3) *$L1$ and $L2$ can not be fused if $n \geq m$, i.e., an outgoing edge from u of $L1$ have iteration number greater than an incoming edge to u.*

Proof: Let G_{L1} and G_{L2} be the sub-graphs of G_P that corresponds to loops $L1$ and $L2$ in P and there exist edges $L((u, v)) = i_{r_j}.n$ *and* $L((w, u) = i_{s_k}.m$ in G_P.

Hypothesis 1: There is an anti dependence if $i_{s_k} > i_{r_j}$.

$\implies$ An incoming edge with label $i_{r_j}.n$ to node v means variable v is Written in nth iteration of instruction i_{r_j}. An outgoing edge with label $i_{s_k}.m$ from node w to node u means variable u is Written in mth iteration of instruction i_{s_k}.

$\implies$ The condition $i_{s_k} > i_{r_j}$ means that first a value is Read from u in nth iteration of instruction i_{r_j} and then Written from w to u in mth iteration of instruction i_{s_k}.

$\implies$ By Definition IV.7, there exists anti dependence between statements i_{s_k} and i_{r_j} if Write operation succeeds Read operation.

Hypothesis 2: $L1$ and $L2$ can be fused if $m \geq n$ and $i_{s_k} > i_{r_j}$.

$\implies$ As $m \geq n$ and $i_{s_k} > i_{r_j}$, even after fusing the statements i_{s_k} and i_{r_j} as Write operation succeeds Read operation on variable u, semantics of code is unchanged.

$\implies$ By definition IV.4, statements of loops $L1$ and $L2$ can be fused if the semantics of the code is intact.

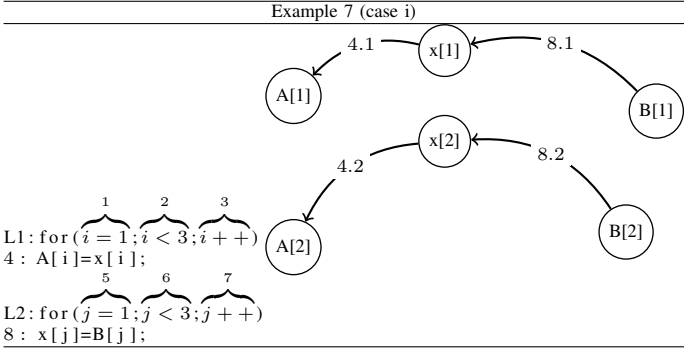
Hypothesis 3: $L1$ and $L2$ can not be fused if $n > m$.

$\implies$ Before fusing, in loop $L1$ Read operation is performed on variable u in nth iteration of instruction i_{r_j}. In loop $L2$ variable u is Written in mth iteration of instruction i_{s_k}.

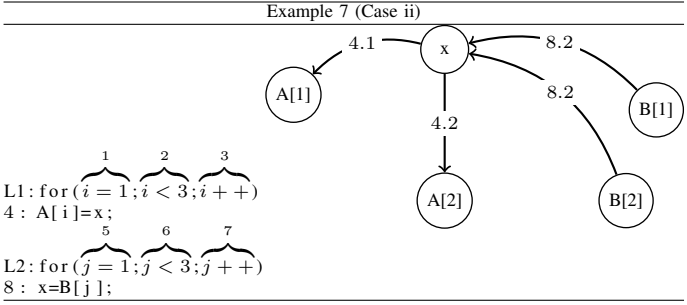
$\implies$ If $L1$ and $L2$ are fused, as $n > m$, variable u is Written in mth iteration of instruction i_{s_k} even before u is Read in nth iteration of instruction i_{r_j} i.e., a new value is written to u even before older value is Read.

$\implies$ As semantics of code changes, loop fusion is not possible if $n > m$. ■

Example 7 (case i)



Example 7 (Case ii)

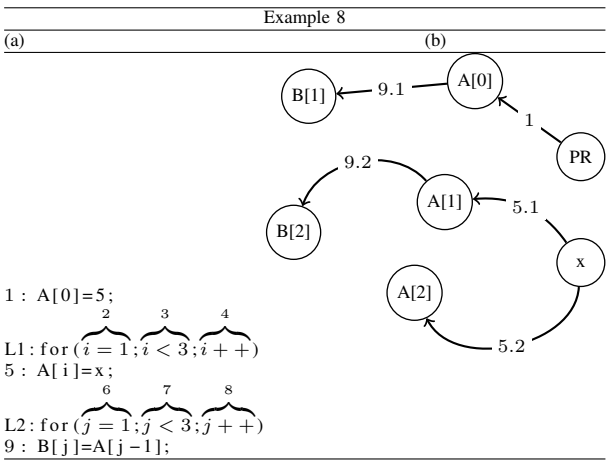


If anti dependence exists between loops $L1$ and $L2$ i.e. if an instruction in $L1$ access an memory location M for Read and the same location is accessed by an instruction in loop $L2$ for Write, merging of loops is possible if M is accessed for *Read* first and then for *Write* even after fusing. In example 7(case i), $L((x[2], A[4])) = 4.2$ and $L((B[2], x[2])) = 8.2$ says $x[2]$ is read in iteration 2 of instruction 4 and written in iteration 1 of instruction 8. As the value is 'Read' in first loop and 'written' in second loop in the same iteration, merging of loops will not change the semantics of code. Therefore edges $L((u, v)) = n.i$ *and* $L((w, u) = m.j$ where $n \in L1$, $m \in L2$ and $j \geq i$ in the graph represents anti dependence where merging is possible.

If anti dependence exists between loops $L1$ and $L2$ merging of loops is *not* possible if a memory location M which is accessed for 'Read' in $L1$ and then for 'Write' in $L2$ is not preserved after fusing. Example 7(case ii) shows the anti dependence where memory location x is accessed for 'Write' many times in first loop and for 'Read' in second loop, merging of loops is not possible.

If **Loop carried forward dependence** exists between loops $L1$ and $L2$ merging of loops is possible. In Example 8, $L((x, A[1])) = 5.1$ and $L((A[1], B[2])) = 9.2$ says memory location $A[1]$ is written in iteration 1 of instruction 5 and is read in iteration 2 of instruction 9. As a value computed in iteration i of first loop is accessed in iteration j of second loop where $j \geq i$, merging of loops will not change the semantics of the code.

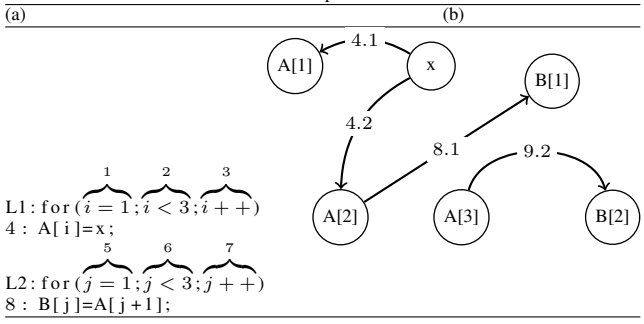
Example 8



If **Loop carried backward dependence** exists between loops $L1$ and $L2$ merging of loops is *not* possible. If a memory location is accessed by an iteration of a statement S_i in loop $L1$ and the same location is accessed by previous iterations of statement S_j in loop $L2$, when such statements are merged S_j in $L2$ will access the memory location first and then S_i which will change the order of execution. As semantics of code will change, fusing of loops is not possible if *loop carried backward dependence* is present between $L1$ and $L2$.

In Example 9, $L((x, A[2])) = 4.2$ and $L((A[2], B[1])) = 8.1$ says memory location $A[2]$ is written in iteration 2 of instruction 4 and is read in iteration 1 of instruction 8 i.e., memory location $A[2]$ is read even before it is updated. As a value computed in iteration i of first loop is accessed in iteration j of second loop where $j < i$, merging of loops can change the semantics of the code.

Example 9



Thus, we conclude that fusion of two loops $L1$ and $L2$ is possible though the above discussed dependencies such

as flow dependence(case i), loop carried dependence, anti dependence(case i) exists between the statements of the loops.

V. Conclusion and Future Work

In this paper, we have introduced a model to perform various optimizations like loop invariant code motion, live range analysis, node splitting and loop fusion through a graphical representation of the program called as Data Dependence Identifier (DDI). For each of the optimization we have investigated on the condition that has to be satisfied by DDI (graphical representation of the program P) so that optimizations can be performed which leads to an effective parallelization of P.

All the optimizations that were discussed are justified as well as validated conceptually with a sequence of rigorous theorems. These theoretical proofs also serve the purpose of the *correctness of proposed algorithms* with which one could easily perform the optimizations of a program.

Salient Features: Though there are many graphical representations for a program, our graphical representation referred as DDI is a unique graphical representation in the state that the variables of P are used as nodes and the edges between the nodes reflect the nature of access (read/write) of the variables from the memory.

Thus, salient features of our work are:

- a novel graphical representation of a program.
- performing almost all the optimizations with one model DDI.

Future Work: The optimization procedures are the main components of parallelization process. With our DDI model, in this paper we have just established the performance of various optimization procedures. Validating the optimization procedures with the benchmarked programs may not yield any significant insight on the performance of optimization procedures with DDI. The reason being that, performance of the various components of a machine may not yield any useful information on the performance of the machine built with those components. For this reason, experimental validation of a full DDI based parallelizer is proposed as future work and to be taken as separate work.

Further, one can initiate investigating DDI for extending the DDI as an optimizer, to as parallelizer. Extension of the DDI as a full fledged parallelizer and the empirical comparison of the DDI based parallelizer with the contemporary parallelizers are the two major works worthful to be considered as future works in the direction of the present paper.

References

[1] Kavya Alluru, Jeganathan L. Graph based Data Dependence Identifier for Parallelization of Programs. https://arxiv.org/abs/2102.09317.

[2] Michael Wolfe. Parallelizing Compilers. ACM Computing Surveys, Vol.28, No.1, March 1996.

[3] A.Aho, R.Sethi, and J.Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley.

[4] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, Shih-Wei Liao, E. Bugnion, M.S Lam. Maximizing multiprocessor performance with the SUIF compiler. Computer, Volume 29, Issue 12, 1996.

[5] https://suif.stanford.edu/

[6] Robert R Wilson, Robert S. French, Christopher S. Wilson, Saman R Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lain, and John L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. ACM SIGPLAN Notices, Volume 29, No. 12, December 1994.

[7] Uptal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A.Padua. Automatic Program Parallelization. Proceedings of the IEEE, Volume 81, Issue 2, Feb 1993.

[8] W. Blume, R. Eigenmann. The range test: a dependence test for symbolic, non-linear expressions. Proceedings of the 1994 ACM/IEEE Conference on Supercomputing.

[9] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, Samuel Midkiff. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. Computer, Volume 42, Issue 12, 2009.

[10] http://pluto-compiler.sourceforge.net/

[11] Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical and Automatic Polyhedral Program Optimization System. Proc. ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, June 2008.

[12] Aravind Acharya and Uay Bondhugula. Effective Loop Fusion in Polyhedral Compilation Using Fusion Conflict Graphs. ACM Transactions on Architecture and Code Optimization, Vol. 17, No. 4, Article 26, September 2020.

[13] Nathanaël Courant, Xavier Leroy. Verified Code Generation for the Polyhedral Model. Proceeding of the ACM on Programming Languages, ACM, 2021, 5 (POPL), pp.40:1-40:24.

[14] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages,1981.

[15] J. Ferrante (IBM), K. J. Ottenstein (Michigan Technological University) and Joe D. Warren (Rice University), 1987. The Program Dependence Graph and Its Use in Optimization.

[16] https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-automatic-parallelization, 2019.