Exploitation of Task Level Parallelism

MAYANK MANGAL¹, ANKIT SANGHAVI², SANDEEP PARODKAR³

¹Assistant Professor, Dept of IT, ARMIET, Thane, Mumbai, India,
²Assistant Professor, Dept of COMPUTER, ARMIET, Thane, Mumbai, India,
³Assistant Professor, Dept of EXTC, ARMIET, Thane, Mumbai, India

Abstract: Existing many systems were supporting task level parallelisms usually involving the process of task creation and synchronization. The synchronization of task requires the clear definition about existing dependencies in a program or data-flow restraints among functions(tasks), or data usable information of the tasks. This paper describes a method called Symbol-Table method which will used to exploits and detects the task level parallelism at inner level of sequential C-programs. This method is made up of two levels: a normal symbol table and an extended symbol table. A sequential program of C language is the input to the normal gcc compiler in which the procedures are defines as functions(tasks). Then we generate a normal symbol table with specific command by gcc compiler in Linux as an output. Then we use the information of that symbol table for generating the extended symbol table with additional information about variable's extended scopes and inner level function dependency. This extended symbol table is generated by the use of previously generated normal symbol table on the basis of variable's scope and L-value/R-value attributes. By that table we can identify the functions and variables those who are sharing the common variables and those who are accessing the different functions with extended scopes respectively. Then we can generate the program dependency graph by the using of that extended symbol table's information with a specific java program. A simple program for using this method has been implemented on a 64-bits Linux based multiprocessor. Finally we can generate the Function graph for every variable in the program with the help of table's info and dependency graph's states. From that graph we can get the info about extended scoping of variables to identify and exploit the task level parallelism in the program. Then we can apply the parallelism with MPI or other parallel platforms to get optimized and error free parallelism.

Index Terms: Task synchronization; Function dependencies; Symbol table; Program Dependency graph;

INTRODUCTION

Parallel programming is much more difficult and erroneous than sequential programming. So, more intension is required to confirm that the parallel source programs give the correct outcomes. Error-free parallel source program does not always compulsorily result in a good performance. So, to achieve the good performance and good efficiency, a parallel program must have load balance, low overhead and the good data locality. In general, parallel programs are much complex and difficult to maintain because they implement complex type of parallel behaviour algorithms, and hold platform-specific optimized source code.

The uninvited difficulties of parallel programming have energized research in the range of parallelizing and rebuilding compilers. These parallelizing compilers consequently discover parallelism in successive projects and rebuild them into parallel projects.

The lion's share of parallelizing compilers [1] [2] [3] have concentrated on parallelism inside loops, where this parallelism gives outcome fromexecuting free cycles of a loop in a parallel way. That is usually called as loop level parallelism. In spite of the fact that these type of parallelizing compilers take out the requirement for parallel programming and are for the most part compelling, later studies [4] [5] have indicated that they have some constraints, and that this parallelism is not sufficient to use the all resources of the parallel computers. So we finally came to task-level parallelism which provides a task(function) as a procedure invocation, a program block or an arithmetic operation. A few provisions are all the more characteristicly communicated as an accumulation of related tasks [6] [5]. Besides, for extensive provisions, it is important to exploit the loop level parallelism and task level parallelisms [5]. On the other hand, not like as parallel compilers, existing frameworks that using task level parallelism interest programming exertion, which extends from needing to physically make andsynchronize of tasks to needing to program in diverse dialects and ideal models. So because of this there is a requirement for frameworks that can exploit task level parallelism with the sequential type of programs.

I. METHOD DESCRIPTION

Method Description

This method is mainly composed of two levels: a normal symbol table and an extended symbol table. The input given to the normal gcc compiler is a sequential C program in which the procedures are defines as functions(tasks) with declarations. For generate a normal symbol table we provide a specific command to gcc compiler in Linux. Then we use the information of that symbol table for generating the extended symbol table with additional information about variable's extended scopes, L/R-value attributes and inner level function dependency. This extended symbol table is generated by the use of previously generated normal symbol table on the basis of variable's scope, Declared line and referenced line of variable in the program. From that table we can identify the functions and variables those who are sharing the common variables and those who are accessing the different functions with extended scopes respectively. Then we can generate the dependency graph based on that table's information to get the clear understandings about the dependencies which are exists in the program. Finally, we can generate the Function graph for every variable in the program with the help of table's info and
dependency graph's states. From that graph we can get the info about extended scoping of variables to identify and exploit the task level parallelism in the program. Then we can apply the parallelism with MPI or other parallel platforms to get optimized and error free parallelism.

Types of Data dependence
This part describes about data dependence and its types. That data about dependence is covered in literature [7] [8] [9] and presented here.
Data dependencies are occurs when two or more iterations, statements or operations of a loop cycle can be executed in parallel. Basically four types of data dependencies are there:
1. True dependency: (also known as Flow Dependency) it occurs between the two statements of a program, if the first statement write the data and the second other statement read it later.
For example, in this program:

\[
\begin{align*}
St1 &: c = a + b \\
St2 &: d = c \times 2
\end{align*}
\]

So from these statements it is clearly shown that here is a true dependency exists between these statements St1 and St2, denoted as St1 St2, because St1 writes var c and St2 reads var c.

2. Anti dependency
3. Output dependency
4. Input dependency

So these types of dependencies can be overcome by variable renaming technique [9]. The only actual dependency is true dependency.

Dependencies can occur between parts or instances of statements in a loop cycle, when the same element (variable) of an array can be accessed by two instances of statements. If these having dependencies related to the same loop cycle then the dependency is known as a loop independent dependency, and without synchronization they can be executed parallel in a concurrent way.

For example, in the program:

\[
\begin{align*}
&\text{for } i = 0 \text{ to } N-1 \\
&\text{St1: } p[i] = q[i] + r[i] \\
&\text{St2: } s[i] = p[i] + 1 \\
&\text{endfor}
\end{align*}
\]

Here it is clearly shown that the dependency that is exist here among the instances of St1 and St2 are loop independent dependency; so, without synchronization they can be executed parallel in a concurrent way. In such type of cases, that loop is called as a parallel loop. However, if the instances relate to different loops cycle, then that type of dependency is called as a loop carried dependency, and without synchronization they cannot be executed parallel in a concurrent way.

For example,

\[
\begin{align*}
&\text{for } i = 0 \text{ to } N-1 \\
&\text{St1: } p[i] = q[i] + r[i] \\
&\text{St2: } s[i] = p[i-1] + 1 \\
&\text{endfor}
\end{align*}
\]

So it is clearly shown that a loop carried true dependency is here between the St1 in iteration i and the St2 in iteration i + 1 where i = 0 to N - 1.

Task level Parallelism

It is another type of parallelism which is a group of co-operating tasks. A unifor computation as task or procedure which can be as coarse-grain as a procedure invocation or as fine-grain as an arithmetic operation that executes more and more number of instructions. When the independent tasks are executing concurrently then this type of parallelism occurs. Tasks which are executing concurrently are not in limited range to operating same type of set of operations in comparison to data-parallelism.

The thought of data dependence could be reached out to tasks. In frameworks where tasks are indicated by the data stream stipulations around the tasks, when task P generates some data which is needed by another task Q then it is said that task Q is dependent on task P. Subsequently, task Q can't begin its execution until task P has finished its execution about the data. In frameworks where the tasks are well synchronized to concurring a particular order, for example, the successive order of data access of the tasks, at that point when task P goes in execution before task Q in execution then it is also said that task Q is dependent on task P. Furthermore, the same data is written by either P or Q. For that situation, task Q can't begin its execution until the task P has completed its execution of the data. In both of the situations, we are referring the task as the pre-requisite task and task Q referred as the dependent type task on P. A graph i.e. which is used to represent the dependencies among the tasks is called task graph. The task graph is a directed type acyclic graph in which the nodes are representing the tasks and the edges are showing the dependencies between the tasks. The source and sink of a dependency edge is a pre-requisite task and a dependent task respectively. In general, when a pre-requisite task completes its execution then only a dependent task can execute.

For example, A task graph is represented in figure of a program. In this the task is taken as a procedure invocation. The program execution would be follows like as. First of all Task 'init' will execute because of no pre-requisite tasks it has. After the completion of the task 'init' three 'comp' type tasks will be execute in parallel. Another task 'sum' will begin its execution when the first two 'comp' tasks will finish their executions, and the another task 'collect' will execute when the last two 'comp' tasks will finish their execution. So it is clearly shown in example that the results of parallelism occurs from executing the 3 'comp' tasks in parallel, and the tasks 'sum' and 'collect' are executing concurrently.

Nonetheless, the adaptable behaviour of this task-level parallelism prompts a few drawbacks. As opposed to building parallelizing tools...
concentrating on a particular programming developed loop, different languages or frameworks have been made to help diverse kinds of task-level parallelism. According to their meaning of the task the frameworks changes; according to the time on which tasks are made; according to the systems which is used to help in task synchronization and communication between tasks; and additionally according to the programming languages and standards used to exploit the parallelism. Hence, to exploit the task-level parallelism, a developer should first pick a framework, and either broadly adjust the sequential type of C-programs or modify them in the standardor programming languages needed by the framework.

II. Proposed method and implementation

The proposed method of this research work motivated by the task level parallelism [16] to exploit the parallelism at inner level of functional dependencies in a sequential C program. The idea of the proposed method developed here only to identify the inner level dependencies of tasks(functions) statically at compile time. For exploiting the concurrency with Task level parallelism we have implemented a simple sequential C program with four functions on the Linux OS with gcc compiler. Here we are targeting the general workings of the compiler about symboltable creation. From that symbol table we are generating the Extended symbol table with additional information about variables and functions of program. From that table we can identify the inner dependencies of functions in a program and by the functional graph we can show the relation between functions about the dependencies. We are using "NetBeans" tool with java platform for generating the dependency graph of program by using that Extended symbol table data in java program.

Exploitation of Task level parallelism with Symbol-Table method:

Our objective is "if 2 functions have common variable then they can not execute parallel(because they are accessing the same memory location of common variable). How these functions are accessing the same memory location?". For proving this we are using symbol table analysis. According to our proposal there are three levels to solve that problem:
2. Extended Symbol-table.
3. Dependency and Function graph representation.

First we are taking a simple sequential C program as input to the gcc compiler with the Linux OS. Then with the specific commands (i.e. gcc filename.c -o filename

gcc -c filename.c

readelf -a filename.o)

We can generate internal process of execution and compilation of program(Executable Locate File(Elf)). It has all information about program like ELF Headers, section headers, section groups, program headers, key to flags, relocation sections, unwind section and most important "Symbol-table". From that we can generate the generalized Symbol-table with information like:
1. Name of Variables and Functions
2. Characteristics Class
3. Token id
4. Scope of Variable and Function
5. Declared line

6. Referenced line
7. Other information like parameters used by functions.

Then for getting the Scope information we generate the Symbol-table for every function which is present in the program. After generating the generalized Symbol-table with all this information we can generate the second level of this method which is called "Extended Symbol-table" with the additional information about extended scopes of variables and the L-value/R-value attributes of the variables as output with the help of previously generated generalized Symbol-table. In that Symbol-table clear information is provided about extended scopes of the variables and L-value/R-value attributes of the variables. For "Extended Symbol-table" the attributes are:
1. Name of Variables and Functions.
2. Scope of Variables and Functions.
3. Declared Line of Variables and Functions in program.
4. Referenced Line of Variables and Functions.
5. Extended Scope of Variables and Functions.

On the basis of Declared line and Referenced line we can draw the dependency graph with the help of java based program of NetBeans tool. And on the basis of Extended scope and L-value attribute we can represent the function graphs manually regarding the information about variables those are changing their scopes with L-value attribute. From these graphs we can easily identify the dependencies among functions and which functions can execute parallel to each other.

Design and implementation of the proposed method:

Our proposed method is based on Symbol-table design and implementation. Here Symbol-table designing process is divided into mainly two parts.

- Generalized Symbol-table.
- Extended Symbol-table.

So for designing and implementing using this Symbol-table analysis method we have taken a simple and small sequential C-program. The source code of that program is given here:

Program source code

```c
#include<stdio.h>
#include <conio.h>

int g = 10;
int I = 15;

Int *h;

int add(int a, int b)
{
    a = a + b;
    g = a + b;
}

int sub(int a, int b)
{
    a = g - b;
    g = *h - b;
}

intmul(int *a, int *b)
{
    *a = *a * *b;
    *b = *h * *a;
}

int div(int *a, int *b)
```

```c
```
```c
{
    *a = *a / *b;
    *h = *h / *b;
}
int main()
{
    printf("main 1 I m = %d %u, g,&g\n\nInt x = 10;\nint y = 5;
int sum = 0, minus = 0;
int multi = 0, divide = 0;

It is a 40 line simple source program which has mainly four functions i.e. add(), sub(), mul(), div(). These functions are sharing some common variables. The variables used in this program are g, h, i, a, b, x and y in which g and h and i are global variables and rest are local variables of different functions. Because of the data inconsistency these functions can't execute with parallel platform. So, For finding the inner dependencies among the functions we are using this Symbol-table method.

So first we generate the symbol-table by gcc compiler with specific command on Linux platform. Than from that table we can generate the "Generalized Symbol-table" using unordered-listed data structure on the basis of some important attributes like Name, Char class, Token id, Scope, Declared line, Referenced line and other info about variables and functions in the program.

We can generate a separate Symbol-table for each scope(function) which is used in the source program. The "Generalized Symbol-table" for that above program is shown in table 1, 2, 3, 4, 5:

Table 1: Generalized Symbol-table with Main Function

<table>
<thead>
<tr>
<th>Name</th>
<th>Char class</th>
<th>Token id</th>
<th>Scope</th>
<th>Decl. line</th>
<th>Ref line</th>
<th>others</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>var int</td>
<td>id &lt; 1 &gt;</td>
<td>(global)</td>
<td>3</td>
<td>9,13,14,26,38</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>var int</td>
<td>id &lt; 2 &gt;</td>
<td>(global)</td>
<td>4</td>
<td>5,14,19,24,38</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>ptr var int</td>
<td>id &lt; 3 &gt;</td>
<td>(global)</td>
<td>5</td>
<td>14,19,24</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>func int</td>
<td>-</td>
<td>1</td>
<td>6-10</td>
<td>31</td>
<td>two parameters a,b</td>
</tr>
<tr>
<td>sub</td>
<td>func int</td>
<td>-</td>
<td>2</td>
<td>11-15</td>
<td>33</td>
<td>two parameters a,b</td>
</tr>
<tr>
<td>mul</td>
<td>func int</td>
<td>-</td>
<td>3</td>
<td>16-20</td>
<td>36</td>
<td>two parameters a,b</td>
</tr>
<tr>
<td>div</td>
<td>func int</td>
<td>-</td>
<td>4</td>
<td>21-25</td>
<td>37</td>
<td>two parameters a,b</td>
</tr>
<tr>
<td>main</td>
<td>func int</td>
<td>-</td>
<td>5</td>
<td>26-40</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>var int</td>
<td>id &lt; 12 &gt;</td>
<td>5</td>
<td>29</td>
<td>34,35,36,37,38</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>var int</td>
<td>id &lt; 13 &gt;</td>
<td>5</td>
<td>30</td>
<td>34,35,36,37</td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td>var int</td>
<td>id &lt; 14 &gt;</td>
<td>5</td>
<td>31</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>minus</td>
<td>var int</td>
<td>id &lt; 15 &gt;</td>
<td>5</td>
<td>31</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>multi</td>
<td>var int</td>
<td>id &lt; 16 &gt;</td>
<td>5</td>
<td>32</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>div</td>
<td>var int</td>
<td>id &lt; 17 &gt;</td>
<td>5</td>
<td>32</td>
<td>37</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Symbol-table for add() Function

<table>
<thead>
<tr>
<th>Name</th>
<th>Char class</th>
<th>Token id</th>
<th>Scope</th>
<th>Decl. line</th>
<th>Ref line</th>
<th>others</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>var int</td>
<td>id &lt; 4 &gt;</td>
<td>1(local)</td>
<td>6</td>
<td>8,9</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>var int</td>
<td>id &lt; 5 &gt;</td>
<td>1(local)</td>
<td>6</td>
<td>8,9</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Symbol-table for sub() Function

<table>
<thead>
<tr>
<th>Name</th>
<th>Char class</th>
<th>Token id</th>
<th>Scope</th>
<th>Decl. line</th>
<th>Ref line</th>
<th>others</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>var int</td>
<td>id &lt; 6 &gt;</td>
<td>2(local)</td>
<td>11</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>var int</td>
<td>id &lt; 7 &gt;</td>
<td>2(local)</td>
<td>11</td>
<td>13,14</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Symbol-table for mul() Function

<table>
<thead>
<tr>
<th>Name</th>
<th>Char class</th>
<th>Token id</th>
<th>Scope</th>
<th>Decl. line</th>
<th>Ref line</th>
<th>others</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>var int</td>
<td>id &lt; 8 &gt;</td>
<td>3(local)</td>
<td>16</td>
<td>18,19</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>var int</td>
<td>id &lt; 9 &gt;</td>
<td>3(local)</td>
<td>16</td>
<td>18,19</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Symbol-table for div() Function

This is a Generalized Symbol-table for above program which contains the information about variables and functions like name, type, token-id, scope, declared line in program, referenced line in program etc. Now on the basis of scope, declared line and referenced line we can generate the "Extended Symbol-table" with additional information about variables like extended scopes and L/R-values attributes.

The Extended Symbol-table over the Generalized Symbol-table is shown in table 6. From this table we can get the additional info about variables like Extended scopes and L/R-value on the basis of reference lines. We can get the common variables from this table those are shared by functions. The variables those have Extended scopes with L-value attributes are comes in focus only.

So with the help of this additional information we can go to generate for innerdependency graph and Function graphs as result.

Table 6: Extended Symbol-table for whole program

<table>
<thead>
<tr>
<th>Name</th>
<th>Char class</th>
<th>Token id</th>
<th>Scope</th>
<th>Decl. line</th>
<th>Ref line</th>
<th>others</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>var int</td>
<td>id &lt; 4 &gt;</td>
<td>(local)</td>
<td>6</td>
<td>8,9</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>var int</td>
<td>id &lt; 5 &gt;</td>
<td>(local)</td>
<td>6</td>
<td>8,9</td>
<td></td>
</tr>
</tbody>
</table>
```
Results:
As we have completed both parts of this Symbol-table analysis method. Now we can generate the inner-Dependency graph with the help of both tables with Declared line and Reference line. The figure 2 shows the dependency graph on the basis of declared and referenced line. This dependency graph is generated by the NetBeans tool which is a java based platform tool. With the help of a java program we are creating a .txt file. Then we are putting the information about declared line, referenced line and number of lines from the generated tables into the .txt file. Than we can execute that java program to get the inner dependency graph. In this graph the nodes or circles represent the line number at which the variable declared or referenced in the program. And the edges represent the data dependencies from declared line node to referenced line node.

In this Dependency graph, black color edges representing Data dependencies between nodes (lines) and blue color edges representing temporary dependencies and red color edges representing the referential dependencies. With the help of this dependency graph and Extended Symbol-table's additional information we can generate the "Function graph" finally.

Function graph:
On the basis of Dependency graph and Extended Symbol-table's information like L-value attribute (only) and extended scope of variables through referenced line we can get the data to draw the Function graph.

The data is like that:
- \( g \rightarrow \{(0,3), (2,14), (5,28,38)\} \)
- \( i \rightarrow \{(0.4), (4,24), (5,38)\} \)
- \( h \rightarrow \{(0,19), (5,29)\} \)
- \( x \rightarrow \{(5,29), (3,36,18), (4,37,23)\} \)

The expressions is written as like: the description of first expression data is:
Here, the left hand side data of an arrow represents the info about variable, forwhich we have to generate the Function graph. The info is, \( g \) is a variable which is defined in scope 0 and declared at line 3 in the program. And the right hand side data of an arrow represents the info about variable \( g \) that what are the extended scopes of \( g \) within it is used, i.e., in scope 1 on line 9 (which is referenced line of \( g \)) variable \( g \) is used, like that it is all the data about var \( g \). Like that for variables \( h, i, x \) and \( y \) that data is given by expressions. On the basis of that data we can generate the graph. From these graphs we can conclude some assumptions about exploiting the task level parallelism that:
- 1 to 1 scoping is allowed between scopes.
- self-loop is allowed (local scoping).
- 1 to many scoping is not allowed.

Figure 3 shows the Function graph for var \( g \). The Function graph clearly shows
Figure 3: Function graph of \( g \) var.

Figure 4: Function graph of \( i \) var.

Figure 5: Function graph of \( h \) var.

Figure 6: Function graph of \( x \) var.

Figure 7: Function graph of \( y \) var.

From figure 4 which is showing Function graph for \( i \) var, we can evaluate that \( i \) var is used by only scope 4 (\( \text{div()} \) function) with actual scoping(Extended scope + \( L \)-value attribute only).

By scope 5 it is used as temporary scoping. So there are no problem with \( i \) var. Likewise figure 5 showing function graph for \( h \) var, in which it is clearly shown that \( h \) var has only one actual scoping in scope 4. According to assumption 1 to 1 scoping is allowed between scopes so there are no problem with \( h \) var.

Figure 6 showing the Function graph for \( x \) var, in which because of referencing dependency which is identified by dependency graph \( x \) var is referenced through scope 3 to scope 5. By graph it is clear that there 1 to many actual scoping is present between scopes for \( x \) var. So these specific scoping functions are sharing common \( x \) var, that's why they can't execute parallel to each other. Likewise figure 7 showing function graph for \( y \) var, in which it is clearly shown that \( y \) var has 1 to 1 actual scoping in scope 3. So there are no problem with \( y \) var.

After analysis of all these graphs finally we can say that:

- \( \text{add()} \) function and \( \text{sub()} \) function can't be execute parallel because of \( g \) var's 1 to many actual scoping.
- \( \text{mul()} \) function and \( \text{div()} \) function also can't be execute parallel because of \( h \) var's 1 to many referencing actual scoping.
- \( \text{add()} \) and \( \text{mul()} \) or \( \text{add()} \) and \( \text{div()} \) may be execute parallel because there are no dependency or scoping conflict between these functions.
- Also \( \text{sub()} \) and \( \text{mul()} \) or \( \text{sub()} \) and \( \text{div()} \) may be execute parallel because there are no dependency or scoping conflict between these functions.

III. CONCLUSION:

The Loop level parallelism had some limitations which are solved by the task level parallelism. After all, with the currently existing frameworks also those support this type of TLP, a system programmer must make the executable updates to the sequential C source program to achieve the required level of task parallelism. In this thesis this work has been done using a method is called Symbol-table method at the time of compilation. This method has basically two different parts i.e. Generalized Symbol-table generation and Extended Symbol-table generation. By these parts of the method we can get the additional information like referencing line, declared line, scope, extended scope and \( L/R \)-value attributes about the variables and functions which is used in the source program. With the help of this
information first we can draw the program dependency graph and after that with whole information about variables we can generate the function graph for each variable. From that graph we can clearly evaluate the inner level dependencies among the functions and extended scoping information about variables. On the basis of that information we can detect and exploit the task level parallelism. Then we can apply the parallelism with MPI or other parallel platforms to get optimized and error free parallelism.

**LIMITATIONS AND FUTURE SCOPE:**

This method has some limitations because of the quality of available compiler analysis, data dependencies that occurs in the sequential C-programs and the choice of only functions as tasks as units of parallelism. The symbol table method is used to get information about variables and functions, is generated manually. So the correct exploitation of parallelism depends on the correctness of the symbol table, which totally depends on the programmer. So we suggest some future work for Symbol-table method to exploit the Task Level parallelism in general. To generate the generalized and extended symbol tables with correct and additional information for program we have to improve the quality of compiler analysis. The next one is that if we want to implement an analysis module that provides side-effects of the statements in a program. Then, in place of providing a wide set of features like as side-effect analysis, code generation and manipulation and dependency analysis, this will concentrate on the information that would help systems to support task level parallelism. This module should be able to accurately describe the accesses of Symbol-table to dynamic data structures such as linked-lists and trees. To identify the recursive procedure's data accesses is also a challenge.

**REFERENCES**


**Author's Profile:**

Mr. Mayank Mangal, received the Master Of Technology degree in Computer Science & Engineering with Software Engineering Specialization from the NIT Rourkela, he received the Bachelor Of Technology degree from RCERT Jaipur. He is currently working as Assistant Professor
in Information Technology Department at ARMIET, A. S. Rao Nagar, Shahapur, Thane, Mumbai.

**Mr. Ankit Sanghavi**, received the Master Of Technology degree in Computer Engineering from the Mumbai University, he received the Bachelor Of Engineering degree from SantGadge Baba Amravati University. He is currently working as Assistant Professor in Computer Department at ARMIET, A. S. Rao Nagar, Shahapur, Thane, Mumbai.

**Mr. Sandeep Parodkar**, received the Master Of Technology degree in EXTC Engineering from the Mumbai University, he received the Bachelor Of Engineering degree from North Maharashtra University. He is currently working as Assistant Professor in EXTC Department at ARMIET, A. S. Rao Nagar, Shahapur, Thane, Mumbai.