Cost Metric for Evaluation of OpenMP Codes

ABSTRACT

In the recent times, there has been a surge in techniques to automatically parallelize sequential codes at the compile time to achieve significant speedups. Parallelizing compilers and automatic parallelization tools transforms sequential codes to parallel ones by making use of the parallelism present in the source code. With the refinement of existing tools and development of new tools incorporating these parallelization techniques, it is now possible to have more than one version of parallelized code for a single sequential problem. These multiple parallel versions have various degrees of speedup on a given architectural setup. However, currently there is no metric which can comment on these multiple versions of parallel codes and determine which one is the most efficient representation of the inherent parallelism present in the source code. In this paper, we introduce a performance measuring metric for parallel OpenMP codes and propose a cost model for statically measuring it. This metric is based on four basic parameters and thus has four components. We also present C-OMP tool, which is an implementation of this cost model and we comment on the quality of few common parallel benchmarks.

1. INTRODUCTION

With the advent of multi-core processors, parallelizing sequential programs has been a crucial agenda for software engineers and researchers alike, for decades. This stems from the basic fact that multicore processors allow concurrent execution of threads. Therefore, each block of a parallel code can be executed by separate threads concurrently and hence, we would have a reduction in execution time as compared to the serial version of the code segment.

Most parallelization opportunities in computer intensive applications arise while executing nested loop statements. This is because loops iterate over program statements and the total number of iterations can extend to over thousand times for common compute-intensive workloads like NAS OpenMP Benchmark and Parsec Benchmark. The most obvious way to reduce the machine cycles is to execute multiple loop iterations in parallel. **Therefore, the principle focus of parallelizing sequential programs is nested loops**. However, in most practical cases parallelism in nested loops is not easy to achieve. This is because there are many ‘inhibitors’ to parallelism which prevent the concurrent execution of the statements inside the loops. Data dependencies are the most common inhibitors to parallelism in loops.

Data dependencies arise when the statements inside loops make many accesses to same memory location and some of these accesses change the contents of that memory location. This prevents parallel execution of iterations and we need to check for data dependencies and resolve them whenever possible. Apart from data dependencies, branch and jump conditions also prevent loop parallelization, but it is very uncommon to find these constructs inside a loop body. Functional exits also prevent parallel execution and not easy to handle.

As a part of checking and resolving dependencies, several loop transformations are applied to the sequential loops in order to effectively utilize the parallelism present in the program. The resultant parallelized code depends on the exact transformations and order in which they are applied. Parallelizing compilers and parallelization tools often use different frameworks like Unimodular framework, Presburger framework for performing loop transformations. For instance, popular parallelization tool AutoPar is based on Rose Compiler Framework while Graphite framework and automatic parallelization tool PLuTo uses the Polyhedral model. Recently, gcc has also been integrated with Graphite framework. Therefore, the transformed parallel code of a sequential program can have multiple versions depending on compiler/tool which generated them. These transformed codes, depending on the level of exploited parallelism by their respective generators, can have different degrees of speedup on a given architectural setup. In this paper, we have introduced a cost metric, which can differentiate between two or more parallel versions of a sequential code by providing insight into the level of exploited parallelism in them. In our discussions we have given special emphasis to parallel codes generated by the Polyhedral framework.

Our aim is to analyze the parallel codes statically and assign the cost metric to each of them. The one with the lower value of the cost metric can be considered to be superior to the other versions. To calculate the cost metric of parallel codes, we need some cost model which can quantify the amount of parallelism based on specific parameters. This cost model will depend upon the parallel programming model used to generate the parallel codes. OpenMP, MPI, Pthreads are the commonly used parallel programing models. We have proposed a **novel cost model based on parallel OpenMP codes.**

Finally, we have presented the **C-OMP tool**, which is an implementation of this cost model using object-oriented programming in Python. We have used this tool to calculate the cost metric of few transformed parallel codes of popular benchmark PARSEC and NAS OpenMP benchmark.

2. BACKGROUND

In this section, we will present a brief overview of OpenMP parallel programming model and Polyhedral Framework. We start by discussing some basic examples of data dependencies and intuition formulating the cost model for OpenMP codes.

2.1 Classifying Data Dependencies

Data Dependence is considered to be the most fundamental roadblock to parallelism. **Two statements S_1 and S_2 are said to contain a data dependence if there is a write operation by atleast one of the statements in the same location which is accessed by both of the statements.** [1] We can classify data dependencies into the three categories:

1. *True Dependence*: S_1 writes into a memory location and the same location is accessed by S_2. If the order of these statements are changed, then A[0] in S_2 might read the wrong value.

   $S_1 : X = 10$
   $S_2 : A[0] = X$

2. *Anti Dependence*: S_1 reads from a memory location and the same location is written by S_2. If the order of these statements are changed, then A[0] in S_1 might read the wrong value.

   $S_1 : A[0] = X$
   $S_2 : X = 10$

3. *Output Dependence*: S_1 writes into a memory location and the same location is written by S_2. If the order of these statements are changed, then some future statement reading the value of the memory location X might get the wrong value.

   $S_1 : X = 10$
   $S_2 : X = 20$

Dependencies in loops are characterized by the iterations in which one of the above dependencies occurs. For example, in the following code fragment there is a true dependence between the current loop iteration and the previous iteration.

```
for i in range(0,N):
    A[i] = A[i+1] + B[i]
```

The above loop cannot be directly parallelized. Thus, checking data dependencies in statements inside the body of loops is one of the prerequisite tasks for parallelizing loops. Compilers and parallelization tools use dependence tests like GCD test, Banerjee‘s Test, etc to check whether it is possible to parallelize a given loop. But before applying these dependence tests, most of the loops present in the programs need to be transformed in order to make them suitable for dependence analysis. Loop Normalization and Induction-Variable Substitution are examples of such pre-dependence transformations. Fine grain parallelism can be further enhanced by applying transformations like Loop Interchange and Loop Skewing.

Different transformations are performed by different parallelizing compilers and tools, based on the framework they are using. Each framework is essentially a high-level abstraction for performing loop transformations. We now provide a brief macro-level view of the Polyhedral Framework.

2.2 Polyhedral Model : Brief Outline

The Polyhedral model is a high-level abstraction for performing loop transformations. It represents an iteration of each statement inside a nested loop as an integral point in a space called ‘polyhedron’ of that statement. A bounded polyhedron is called a ‘polytope’ and these bounds are the upper and lower limits of the loop. The loop transformations are done by doing affine transformations on the polytopes and the main idea is to partition the iteration domain in such a way that the data dependencies are resolved.

The Polyhedral model only works for nested loops where the bounds are affine functions outer loop variables. [2]

2.3 OpenMP Parallel Programming Model

During the past decade, OpenMP has emerged as one the most popular APIs for shared-memory parallel programming. Its simplicity is derived by the fact that it provides an extension to the source programming language by allowing the insertion of directives and pragmas in appropriate junctures at the sequential source code, without much modification. In contrast to other parallel programming models like MPI and Pthreads, OpenMP does not demand that the programmer should possess internal knowledge like associating a specific code region with a specific thread or avoiding deadlocks between two or more processes and so on. In addition to that, all data in an OpenMP program is global and shared among the threads by default, which makes parallelization of loops easier. [3]

Parallelizing Compilers like Intel Compiler, TRACO, Oscar Compiler, Portland’s Compiler transforms sequential C/C++ and Fortran codes to corresponding parallel codes using the OpenMP API. Traditional Compilers like GCC also supports distribution of sequential codes into multi-threaded codes for specific loop constructs using OpenMP.

2.3.1 General Structure of an OpenMP Code

The structure of a general OpenMP code is similar to that of a normal C/C++ code, with the inclusion of directives. Directives in OpenMP are special comments which alter the behaviour of the code segment immediately following it. Fig.3.1 shows the syntax of an OpenMP directive.

```
#pragma omp directive-name [clause[[,] clause]...]
```

Fig 3.1 Structure of a general OpenMP Directive: The directive name specifies the behaviour of the code segment following it.

Broadly, any OpenMP code can be broken down into one or more of these following components:

1. *Parallel Loops*: These are the loops can be completely executed in parallel, i.e. the loop iterations do not have to follow a strict order and can be rearranged. This is only possible if there are no dependencies present in the final loop statements.

2. *Vector Loop*: These loops allow the interleaving of operations from one iteration to another, but loop iterations cannot be executed in any random order. Iterations of a vector loop are executed in a single thread and these

iterations form a chunk, whose size depends upon the vector resources of the target machine.

3. *Sequential Loops*: These are the loops which cannot be parallelized because of possible dependencies and has to be executed sequentially.

4. *Barriers*: These occur between two parallel segments of OpenMP code to act as synchronization between these regions.

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

void mxv(int m, int n, double * restrict a,
double * restrict b, double * restrict c)
{
int i, j;

#pragma omp parallel for default(none) \
shared(m,n,a,b,c) private(i,j)
for (i=0; i<m; i++)
{
a[i] = 0.0;
for (j=0; j<n; j++)
a[i] += b[i*n+j]*c[j];
} /*-- End of omp parallel for --*/
}
```

Listing 1: OpenMP implementation of the matrix times vector product in C

3. COST MODEL

Based on the general structure of an OpenMP Parallel Codes, there are several factors which can determine the performance of one parallel code from another in a given architectural setup. Multiple OpenMP codes can be differentiated based on the counts of the four basic components (Parallel, Vector and Sequential Loops and Barriers). The Cost Model is based on the intuition that each of these four components separately impacts the degree of speedup. Our goal is to assign a cost metric with four numerical parameters to every OpenMP code. Each of these parameters will be based on the four fundamental components. This metric will provide insight into the level of parallelism expressed in different OpenMP codes.

The OpenMP Cost model states that: ***"Performance of any OpenMP Parallel program/code largely depends on the following basic components:***

- ***Cost of Parallel Loops***
- ***Cost of Sequential Loops***
- ***Cost of Vectorized Loops***
- ***Number of Synchronizations(Barriers) "***

Deeper inspection into structure of OpenMP codes reveal that other advanced components like addition of control statements, function calls and pointers as arrays also significantly impacts the performance of OpenMP codes, but we are not considering these components into the cost model in this discussion. We shall now look into the ways to calculate each of the four parameters associated with the cost metric.

3.1 Basic Parameter 1: Cost of Sequential Loops

Sequential Loops in OpenMP programs can be easily identified, as these loops do not have any OpenMP directive preceding them. Execution of Sequential loops each time incurs an **unit cost** for each of the statements present in the body of the loop.

```
for $(i = 0;\ i < 10;\ i++)$
    printf("%d",i);  //$S_1$
```

Figure 1.1 Example of a simple sequential loop - The cost incurred by statement S_1 is 10 units, since the loop runs 10 times

Cost of Simple Sequential loops like the one depicted in **Figure 1.2** can easily be found out by subtracting the upper-bound of the loop from its lower-bound.

```
for $(I_1 = L_1;\ I_1 < U_1;\ I_1++)$
  for $(I_2 = L_2;\ I_2 < U_2;\ I_2++)$
    .
    .
    .
    .
    for $(I_k = L_k;\ I_k < U_k;\ I_k++)$
        $S_1$
```

Figure 1.2 Generic form of a simple sequential loop - The total cost of this loop is obtained by evaluating the expression $\prod_{i=1}^{k}(U_k - L_k)$

However, in practical situations when we encounter OpenMP codes generated by Parallelizing Compilers and Parallelization Tools, we get transformed sequential loops with functional bounds.**[Figure 1.3]**

```
for $(i = 0;\ i < 10;\ i++)$
  for $(j = max(2*i+4, 10);\ j <= min(3*i, 2);\ j++)$
      $S_1$
```

Figure 1.3 Example of a sequential loop having functional bounds generated by PLuTo - The functions used here are **max** & **min**, which are pre-defined as **Macros** in the generated codes.

To evaluate the cost of sequential loops having functional bounds, we have broadly divided these loops into two categories:

- **Category 1** : Sequential Loops having single level of nesting with functional bounds of form $f(a*t+b,k)$, where **t** is the parent loop variable and **a, b, k** are constants. These are handled by **Point of Inflection Method.**

- **Category 2** : Sequential Loops having k-levels of nesting with functional bounds of form $f(a_1*t_1+a_2*t_2+..+b, a_1^{'}*t_1+a_2^{'}*t_2+..+b^{'})$, where $t_1, t_2, ...$ are the parent loop variables and $a_1, a_2, ..., b_1, b_2, ..$, are constants. These are handled by **Critical Point Analysis Method.**

It is clearly noticeable that loops in **Category 1** are a subset of the larger set of Sequential Loops in **Category 2.** Due to the prevalence of **Category 1** loops in OpenMP Codes generated by Parallelizing Compilers and Parallelization Tools, the Point of Inflection Method is used to evaluate them, rather than applying the Critical Point Analysis Method, for the sake of reducing the computation time.

3.1.1 Point of Inflection Method

This method is used to determine the number of times a single-nested loop, having functional bounds, is executing, statically at the compile-time. Typically, loops handled by this method falls under Category 1.

Let us consider a simple **Category 1** Sequential Loop, having functional bounds and single-level of nesting.

```
for ($I_1 = L$; $I_1 <= U$; $I_1 ++$)
    for ($I_2 = f(a * I_1 + b, k_1)$; $I_2 <= f'(c * I_1 + d, k_2)$;
    $I_2 ++$)
        $S_1$;
```

Figure 1.3 Example of a General *Category 1* loop - functions f and f' are defined as Macros in the generated parallel OpenMP Codes and are generally restricted to **max, min, floor** and **ceil.**

In the **Point of Inflection Method**, we only consider functions **max** and **min** in the functional bounds of the Sequential Loops.

Considering the loop in Figure 1.3, the outer-most loop is executing $(U - L + 1)$ times. For finding out the number of times the inner loop is executing, we define the **Point of Inflection** as:

"The point 'i' in the domain interval [L,U] of function** $f(a*t+b,k)$ **from where the value of the function changes."

For Example, let us consider the function $min(t+2,6)$, where $0 \le t \le 10$. The value of this function over its domain can be written as:

$$\text{min(t + 2, 6)} = \begin{cases} \text{t + 2}, & 0 \le t \le 4 \\ 6, & 4 \le t \le 10 \end{cases}$$

The value of this function is $t+2$ when the value of t lies in range [0,4] and when t lies in range [4,10], the value becomes 6. As the value of this function changes at $t = 4$, 4 is the Inflection Point of $min(t+2,6)$, when $t\varepsilon[0,10]$. **[Figure 1.5]**

3.1.2 Finding the Infection Point Analytically

For any **max** or **min** functions of type $f(a*t+b, c*t+d)$, the Point of Inflection can be calculated as:

$$\begin{aligned} \Rightarrow a*i+b &= c*i+d \\ \Rightarrow (a-c)*i &= (d-b) \end{aligned}$$

$$\boxed{\Rightarrow i = \frac{d-b}{a-c}}$$

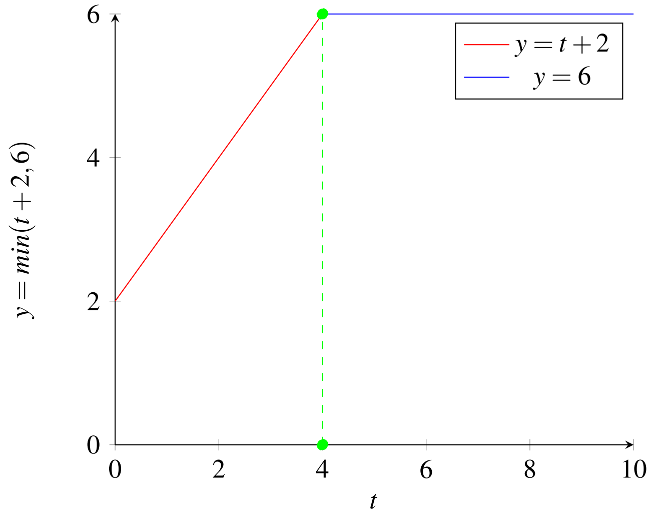

Figure 1.5 Plot of $min(t+2,6)$ **, where** $t\varepsilon[0,10]$ - inflection point is at $t = 4$

Considering the loop in **Figure 1.3**, let 'i_1' and 'i_2' be the Points of Inflextion of the lower-bound and upper-bound of Loop Variable I_2 respectively.

$$\text{Lower-Bound} = f(a*I_1+b,k_1) = \begin{cases} a*I_1+b, & L \le I_1 \le i_1 \\ k_1, & i_1+1 \le I_1 \le U \end{cases}$$

$$\text{Upper-Bound} = f'(c*I_1+d,k_2) = \begin{cases} c*I_1+d, & L \le I_1 \le i_2 \\ k_2, & i_2+1 \le I_1 \le U \end{cases}$$

If the values of i_1 and i_2 are not equal, then they need to *partitioned* into uniform intervals so that the expression (Upper-Bound - Lower-Bound + 1) can be evaluated to denote the number of times the inner-most loop in **Figure 1.3** is executing.

It is possible that the actual values of the Upper and Lower-Bounds might get flipped with respect to the intervals given, i.e the Lower-Bound might take the value k_1 in $[L, i_1]$ and vice-versa.

3.1.3 Partitioning of Intervals

The intervals of the bounds of a General Category-1 Loop can be written as:

$$\begin{array}{lccc} \text{Lower-Bound Intervals} = & [L,i_1] & to & [i_1+1,U] \\ \text{Upper-Bound Intervals} = & [L,i_2] & to & [i_2+1,U] \end{array}$$

These intervals needs to be partitioned in such a way that they are uniform for both Upper and Lower Bounds of the loop. This can be achieved by splitting the boundary points of the larger interval to match the boundary points of the smaller interval, as shown here :

- **Case 1**: $i_1 < i_2$
  Partitioning is achieved by splitting the intervals in the following manner:
  $$\begin{array}{lccccc} \text{Lower-Bound Intervals} = & [L,i_1] & to & [i_1+1,i_2] & to & [i_2+1,U] \\ \text{Upper-Bound Intervals} = & [L,i_1] & to & [i_1+1,i_2] & to & [i_2+1,U] \end{array}$$
- **Case 2**: $i_1 > i_2$

Partitioning is achieved by splitting the intervals in the following manner:
Lower-Bound Intervals = $[L, i_2]$ to $[i_2+1, i_1]$ to $[i_1+1, U]$
Upper-Bound Intervals = $[L, i_2]$ to $[i_2+1, i_1]$ to $[i_1+1, U]$

- **Case 3**: $i_1 = i_2$
  Partitioning is not required as the intervals of Upper and Lower Bounds are already uniform.

If we rewrite the loops with in-accordance with the 'new' partitioned intervals(assuming $i_1 < i_2$), we get:

$$\text{Lower-Bound} = f(a*I_1+b, k_1) = \begin{cases} V_1, & L \le I_1 \le i_1 \\ V_2, & i_1+1 \le I_1 \le i_2 \\ V_3, & i_2+1 \le I_1 \le U \end{cases}$$

$$\text{Upper-Bound} = f'(c*I_1+d, k_2) = \begin{cases} V_4, & L \le I_1 \le i_1 \\ V_5, & i_1+1 \le I_1 \le i_2 \\ V_6, & i_2+1 \le I_1 \le U \end{cases}$$

3.1.4 Calculation of Loop Iterations

The number of iteration of a general Category-1 Loop in **Figure 1.3** can be divided into three parts, based on the three interval ranges found.

We determine the number of values occurring in each interval, then we can sum over all these values to determine the number of times a loop is executing.

Number of Values in Interval	$[L, i_1]$	$=$	$(V_4 - V_1 + 1)$
Number of Values in Interval	$[i_1+1, i_2]$	$=$	$(V_5 - V_2 + 1)$
Number of Values in Interval	$[i_2+1, U]$	$=$	$(V_6 - V_3 + 1)$

Summing over all these values, we get:

$$\sum_{I_1=L}^{i_1}(V_4 - V_1 + 1) + \sum_{I_1=i_1+1}^{i_2}(V_5 - V_2 + 1) + \sum_{I_1=i_2+1}^{U}(V_6 - V_3 + 1).......(1)$$

The above expression gives an estimate of number of times the inner-most loop in **Figure 1.3** is executing. To get the overall cost, we multiply the outer-loop cost, i.e (U - L + 1) with the inner-loop cost.**[Figure 1.6]**

$$(U - L + 1) * [\sum_{I_1=L}^{i_1}(V_4 - V_1 + 1) + \sum_{I_1=i_1+1}^{i_2}(V_5 - V_2 + 1) + \sum_{I_1=i_2+1}^{U}(V_6 - V_3 + 1)]$$

Figure 1.6 Cost of a General Category-1 Sequential Loop- this result can be evaluated at the compile time.

3.1.5 Drawbacks of Point of Inflection Method

One of the major drawback of this method is that the final result cannot be estimated analytically; the final expression will depend upon the parameters of the loops and cannot be generalized. Apart from this, it is Only applicable to Single Level Nesting, with a specific form of parameters in the functional bounds. These limitations direct us to **Critical Point Analysis Method**, which will generate analytical expressions for complex nested loops with functional bounds.

In spite of these limitations, **Point of Inflection Method** can still be applied to estimate the cost of Category 1 loops with a computation time minimal to that of other methods.

3.1.6 Critical Point Analysis Method

This method is used to determine the number of times a k-level nested loop, having affine functional bounds, is executing, statically at compile time.

The bounds of the inner-loops(child loops) are affine functions of the outer-loops(parent loops) with respect to the given loop in consideration.

Let us consider a k-level nested loop, falling under **Category-2** Sequential Loops.

```
for ($I_1 = L_1; I_1 \le U_1; I_1++$)
  for ($I_2 = f(I_1); I_2 \le f'(I_1); I_2++$)
    .
    .
    .
    .
    for ($I_k = f(I_1, I_2, ..., I_{k-1}); I_k \le f'(I_1, I_2, ..., I_{k-1});$
    $I_k++$)
        $S_1$
```

Figure 1.2 Generic form of a simple Category 2 sequential loop - This loop contains algebraic bounds which are obtained by transformation from the original functional bounds.

Specifically, we have:

- Bounds of Level-1: L_1 to U_1
- Bounds of Level-2: $f(I_1) = (a_{21} * I_1)$ to $f'(I_1) = (a'_{21} * I_1)$
- Bounds of Level-3: $f(I_1, I_2) = (a_{31} * I_1 + a_{32} * I_2)$ to $f'(I_1) = (a'_{31} * I_1$ to $a'_{32} * I_2)$
- Bounds of Level-k: $f(I_1, I_2, ..., I_k) = (a_{k1} * I_1 + a_{k2} * I_2 + .. + a_{kk-1} I_{k-1})$ to $f'(I_1) = (a'_{k1} * I_1 + a'_{k2} * I_2 + .. + a'_{kk-1} I_{k-1})$

Representing the bounds as a system of inequalities:

$$\begin{array}{ccc} L_1 & \le I_1 \le & U_1 \\ L_2 + a_{21}I_1 & \le I_2 \le & U_2 + a'_{21}I_1 \\ L_3 + a_{31}I_1 + a_{32}I_2 & \le I_3 \le & U_3 + a'_{31}I_1 + a'_{32}I_2 \\ & \cdot & \\ & \cdot & \\ L_k + a_{k1}I_1 + a_{k2}I_2 + .. + a_{kk-1} & \le I_k \le & U_k + a'_{k1}I_1 + a'_{k2}I_2 + .. + a'_{kk-1} \end{array}$$

3.2 Basic Parameter 2: Cost of Parallel Loops

Parallel loops in OpenMP are preceded by the directive **#pragma omp parallel**. This makes the thread which executes the region to create a number of other threads which executes different iterations in the loop. The number of created threads roughly depends upon the available cores for execution. All iterations which execute simultaneously at a time incur a unit cost.

Let us consider a simple parallel loop with upper bound 'U' and lower bound 'L' and let 'n' by the number of cores present in the machine. **[Figure 3.1]**.

```
# pragma omp parallel
for (i = 0; i < 10; i++)
     printf("%d",i);  //S1
```

Figure 3.1 Example of a simple parallel loop - The cost incurred by statement S_1 is 2.5 units, since there are 10 iterations of loop divided over 4 cores

The cost contributed by this loop can be calculated as:

$Cost_{parallel}$ = (Total number of loop iterations)/(number of cores) = $(U - L + 1)$ /(number of cores

3.3 Basic Parameter 3: Cost of Vector Loops

In this version of cost model, for simplicity, we have assumed that vector loops behaves as the same way as parallel loops.

3.4 Basic Parameter 4: Synchronization Costs

The synchronization costs occur whenever there is more than one parallel region in the code and a barrier has to be inserted to make sure that these regions do not overlap. It can be simply calculated as:

$Cost_{Barrier}$ = (Total number of parallel loops) + (Total number of vector loops) - 1

4. INTRODUCING C-OMP TOOL

In this section we introduce C-OMP, which is an implementation of the cost model for evaluating OpenMP Codes. C-OMP takes in one or more parallel OpenMP codes as its input and provides the user with the program cost metric and loop profiles, which enables comparison between the OpenMP codes in terms of quality and extent of parallelization. In contrast to other models, C-OMP estimates the cost of OpenMP Programs statically and at the compile time.

The working of C-OMP is divided into four phases -

- **Program Initialization Phase**
- **Program Construction Phase**
- **Analytical Calculations Phase**
- **Result Comparison and Analysis Phase**

4.1 Program Initialization

This is the first phase of the C-OMP tool. In this phase, the OpenMP Program files are read and made suitable for further processing. A *'Cleaned-Up'* version of the input code generated by stripping the blank and comment lines from the source program. In addition to that, certain adjustments to *'for loops'* and *'conditional statements'* in the source code are made to make them suitable for parsing. This *'Cleaned-Up'* code is then passed on the next phase of the tool.

4.2 Program Construction

The second phase of the C-OMP Tool primarily focuses on creating loop objects, building 'regions' and mapping them to the **'Program'** Class.

A **'Region'** in the source program is defined as any fragment of the code enclosed by and braces. After all possible regions are identified from the *'Cleaned-Up'* code, they are categorized into regions occurring from loops, conditional statements and functions. **'Loop'** class is a high level abstraction of loops present in OpenMP Codes. This base class by inherited by three sub-classes: **Parallel_Loop**, **Vector_Loop** and **Sequential_Loop** corresponding to the nature of loops encountered in OpenMP Codes. Code of function *make_regions()* is presented below.

```python
def make_regions():
    """function analyzes the cleaned-up code for '{' and '}' braces and determines the regions. Returns a list of list contains starting and ending indices of regions enclosed by '{}' in the cleaned-up code.Also, finds out all the assignment statements in the cleaned-up code."""

        regions = []
        cache = []
        for index in range(len(List)):  # List contains the cleaned-up code

            if "{" in List[index]:
                cache.append(index)
            elif "}" in List[index]:
                regions.append([cache.pop(), index])

            #searching for assignment statements
            if z.assign_st.search(List[index]) is not None:
                if 'for' not in List[index]:
                    assignment_st[index] = List[index]

            #searching for 'if' statements
            if z.if_st.search(List[index]) is not None:
                if 'for' not in List[index]:
                    if_st[index] = List[index]
        return regions
```

Listing 2: Python Code to construct regions

4.3 Analytical Calculations Phase

In this phase, the specialized cost functions of each loop types will be called and the output will be a four element tuple $(Cost_{Sequential}, Cost_{Parallel}, Cost_{Vector}, Cost_{Barrier})$

4.4 Result Comparison and Analysis Phase

In this phase, we will compare the cost metric output tuple of various OpenMP codes and attempt to distinguish the levels of parallelism expressed in those codes.

5. RESULTS

5.1 Loop Distributions

We first start our results phase by first analyzing the distribution of both For Loops and While loops in common workloads. The goal of this study is to show that although parallelization opportunity occurs in any both types of the loops, we can neglect the effect of While loops in our discussion without sacrificing too much correctness. Another reason for not considering While loops in our discussions is that it is very difficult to know the number of iterations for a while loop statically, at the compile time. So, estimating their costs is a daunting task.

As we can see from Table 1 and Table 2 [4], majority of compute-intensive workloads use For loops over While loops. Listing-3 shows an example of OpenMP Code generated by PLuTo. The parallel loop will run from 0 to 99 and assuming a quadcore machine, we get $Cost_{Parallel} = (100/4) = 25$. Similarly,$Cost_{Vector} = (10/4) = 2.5$. Therefore, cost metric output tuple of this code is : (0,25,2.5,1)

Table 1: Distribution of FOR Loops and WHILE Loops in PARSEC benchmark

File Name	FOR Loops	While Loops
test_parallel_do	4 100%	0 0%
test_parallel_for	7 88%	1 12%
test_parallel_for_each	1 100%	0 0%
test_parallel_invoke	9 100%	0 0%
test_parallel_pipeline	6 100%	0 0%
test_parallel_scan	8 100%	0 0%
test_parallel_sort	8 73%	3 27%
test_parallel_reduce	4 100%	0 0%
parallel_preorder	4 100%	0 0%

Table 2: Distribution of FOR Loops and WHILE Loops in some Open Source Projects

File Name	FOR Loops	While Loops
Linux Kernel	58431 64%	31593 36%
OpenCV	8441 86%	1334 14%
VLfeat	406 74%	142 26%
Quantlib	3547 85%	640 15%
GCC	29965 77%	8729 23%
GSL	4226 94%	290 6%
Scilab	4533 92%	414 8%
ASL	481 96%	17 4%
Blender	14748 84%	2767 16%
FFTW3	514 91%	51 9%
MLPack	695 92%	60 8%
SuiteSparse	2377 98%	48 2%

```
int t1, t2;
int lb, ub, lbp, ubp, lb2, ub2;
 register int lbv, ubv;
/* Start of CLooG code */
lbp=0;
ubp=99;
#pragma omp parallel for private(lbv,ubv,t2)
for (t1=lbp;t1<=ubp;t1++) {
  lbv=0;
  ubv=10;
#pragma ivdep
#pragma vector always
  for (t2=lbv;t2<ubv;t2++) {
    A[t1+2][t2] = 2 + t1 + t2;;
    B[t1][t2] = x[t1+3][t2+2];;
  }
}
/* End of CLooG code */
```

Listing 3: Example OpenMP code generated by PLuTo

6. REFERENCES

[1] K. Kennedy and A. J. R., *Allen, R.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann*. 01 2002.

[2] U. Bondhugula, J. Ramanujam, and P. Sadayappan, “Pluto: A practical and fully automatic polyhedral program optimization system,” 12 2018.

[3] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[4] B. Supratim, “Static analysis of array references inside loops for parallelization,” 01 2015.