Abstract—There is a growing interest in the migration of legacy sequential applications to multicore hardware while ensuring functional correctness powered by automatic parallelization tools. OpenMP eases the loop parallelization process, but the functional correctness of parallelized code is not ensured. We present a methodology to automatically analyze and prepare OpenMP constructs for automatic parallelization, guaranteeing functional correctness while benefiting from multicore hardware capabilities. We also present a framework for procedural analysis, and emphasize the implementation aspects of this methodology. Additionally, we cover some of the imperative enhancements to existing dependency analysis tests, like handling of unknown loop bounds. This method was used to parallelize an Advance Driver Assistance System (ADAS) module for Lane Departure Warning System (LDWS), which resulted in a 300% performance increase while maintaining functional equivalence on an ARM™ based SOC.

Keywords: Automatic Loop Parallelization, OpenMP, Range Test, Conditional Parallelization, Loop Normalization, Loop Dependency Analysis

1. Introduction

Modern computer architecture ethos favors a multicore design paradigm for both servers and embedded solutions. Multicore platforms add a significant overhead to program development cycles due to added complexities, and strict verification of compatibility issues. The standard procedure to increase the throughput of a sequential program, is to optimize or parallelize the application. Parallelizability of an application is based on the inter-dependency of the tasks within that application. Detailed code analysis is required to decide the dependency and parallelizability of the tasks in code. Sometimes, the time overhead induced by multithreading using the OpenMP API may nullify the parallelization benefits. Thus, availability of multiple cores does not always guarantee a performance improvement. The majority of current as well as legacy applications, especially in safety critical domains like automotive, aerospace, etc. are sequential. When these applications are parallelized, their functional correctness for all possible set of inputs is a major concern. The application performance is essential but is of lower priority compared to functional correctness. Migration of these applications to parallel or multicore hardware requires thorough verification as parallelization may induce complex errors. For e.g. synchronization errors, deadlocks, resource contention, etc. using dependency analysis which plays a major role in parallelization process [13]. Manual analysis of code for parallelizability is a complex and time consuming [15] activity depending upon number of lines, inter-dependent functions, and file count. In these efforts, human error may still creep in, especially in large code sets as it is humanly impossible to keep track of multiple synchronizations across files. This paper focuses on the mechanism developed to automatically convert sequential code into parallelized code using OpenMP constructs. This approach comprises of:

1) automatically analyzing a loop block for dependencies, and
2) checking functional correctness.

The programmer can then fine-tune the parallelized code for further optimal behavior. In the upcoming section we will discuss in detail the methodologies used in the loop parallelization module of YUCCA tool, which was earlier known as S2P [3].

2. Literature survey

Automatic parallelization includes analysis of local and global variables, array indices, inter-procedural dependencies, alias analysis, control flow analysis, etc. Runtime preprocessing to efficiently reduce the cost of communication along with non-linear array sub-script analysis for irregular parallelizable code is handled by automatic parallelization techniques for irregular scientific applications [11]. Compiler guide lines to programmer and semi-automatic loop parallelization based on iteration analysis, variable induction analysis is handled by the technique of compiler guided refactoring [12]. There has been work done in the context of adaptive work load distribution across OpenMP loops using machine learning [8]. Machine learning based thread versioning results in optimum work-share distribution over the threads. Run-time comparison comes up with optimum OpenMP combinations in regards of chunk and number of threads. Fine coarse grain parallelization techniques such as tiling of a given data set, and performing distinct tile operation on different processing cores are also being taken care of by automatic parallelization tools like PLUTO [7]. A polyhedral model is used in PLUTO to analyze the irregular
dependencies in nested loops to find opportunities for parallelization. There are tools for automatic parallelization like Intel Compilers [1], Par4All [2], Cetus [4] etc. described in the automatic parallelization YUCCA tool [3], which do not cover implementation in detail.

3. Prerequisites and limitations for loop parallelization using OpenMP

- Loops to be detected by static analysis for parallelization needs to be normalized before processing.
- Loop indices should always be integers.
- OpenMP does not check for data dependencies, data conflicts, race conditions, and deadlocks.
- OpenMP does not allow static extent scope expanse to multiple code files.
- The functional correctness of a parallelized block is not verified by OpenMP.
- Jump statements should not directly or indirectly break the loop considered for parallelization.
- The types of variables under consideration are primitives, arrays, and pointers or array references.
- Loop stride and loop conditions are loop invariant.

4. Dependency analysis

4.1 Simple variable analysis

Variable analysis is an essential part of static analysis to find out the parallelizability of a loop. This section covers all possible types of variables and scenarios where variables are used. The typical flow for the proposed methodology is as shown in figure 1.

The following sections cover some methodologies for variable analysis.

4.1.1 Prediction of variable values

The dependency analysis gets changed based on the scope of the variable (i.e. whether the variable is local or global with respect to the loop). The values of the variables are identified based on their location.

In example 1, on line 4, variable $a$ is assigned a constant, 10, and in line 5 $b$ is assigned 23. Within the for loop on line 8, variable $c$ is an expression in the form of the abstract syntax tree (AST) as shown in figure 2.

The same AST is modified as shown in figure 3.

In the same way all the variable are reduced and possible values of that variables are stored in a list.

4.1.2 Handling of branches

If branching is detected, all the possible paths are noted. A list comprising of all the possible values for each variable in each branch is then generated. Branches that cannot be reached based on variable values are ignored for dependency analysis. For example,

```c
if (b==12){
```

Fig. 1: Flowchart of dependency analysis

![Flowchart of dependency analysis](image)

Example 1 Sample source code for variable value prediction

```c
main () {
    int a, b, c, i, j;
    ...
    a = 10;
    b = 23;
    for ( i = 0 ; i < 10; i++){
        ...
        c = a + b;
        b = c + 1;
        a = a + b + c + 43;
    }
}
```

Fig. 2: AST for expression $c = a + b$

![AST for expression $c = a + b$](image)

Note that jump statements can be used to break nested child blocks, but not the parent block which is considered for parallelization.
If it is known that 12 is not a possible value of \( b \), then the then path of if statement is ignored.

4.1.3 Read write test

The Read Write Test identifies if there is any loop carried dependency. It verifies whether a variable written in a loop is used in the next iteration. We do this by determining if there is a read access on the variable before a write access. In case such dependencies exist, the loop is marked non-parallelizable.

In example 2, \( x_2 \) is read on line 7 before it is updated. As the value of \( x_2 \) written in current iteration is used in the next iteration, this loop is considered non-parallelizable.

Example 2 Sample code for read write test

```c
1 main (){  
2   int i,k,x1,x2;  
3   int a[20];  
4   ...  
5   for (i = 0; i < k; i++) {  
6     ...  
7       x1 = x2 + 4;  
8       x2 = x1 + k + i;  
9     ...  
10   }  
11 }  
12 }
```

In example 3, \( l_2 \) is read before it is written. Hence, we mark this loop non-parallelizable as well. However, this scenario can be handled using the Reduction construct found in OpenMP.

Another case where the read write test requires some assistance from other tests is if there is a conditional control flow in the loop, and the variables are read or modified as a part of the conditional control flow.

Arrays and pointers in a loop are analysed using other tests which are explained in the sections below.

4.1.4 XNOR test

As discussed, the read write test considers the position of a variable, but sometimes it is also needs to consider the control flow of the loop where the variable is being used. The XNOR test is used for variables which are accessed first in a conditional block. If the variable is accessed before the conditional block, the read write test is considered for each branch.

The following scenarios illustrate the cases where the read write test needs the assistance of the XNOR Test.

A variable first accessed in a conditional block is,

(a) only accessed inside the conditional block,
(b) only written in all the branches,
(c) only read in all the branches, or,
(d) read in at least one branch as well as written in another branch

In scenario a, if variable \( x \) is accessed only in the conditional block, then the read write test is applied on that variable with respect to each branch separately. If there is at least one branch that the read write test marks as non-parallelizable, then the entire loop is marked non-parallelizable.

In scenario b, if the variable \( x \) is first accessed in the conditional block and all branches have write access to that variable, then that loop is marked non-parallelizable with respect to that variable. However, if there is a write after the branch statements, the loop is marked parallelizable.

In scenario c, if the variable \( x \) is first accessed in the conditional block and all branches have read access from that variable, then the loop is marked non-parallelizable with respect to that variable. However, if there is a write after the branch statements, the loop is marked parallelizable.

Static analysis of scenario d cannot determine which branch of the control flow graph will occur during any iteration, hence we mark this case as non-parallelizable as well.

In the example 4, variable \( b \) is only accessed in the conditional block. In one branch it is written first and read later, and in the other branch, it is read first and written later. This will cause the read write test to fail in one branch, so this loop is marked non-parallelizable.

4.2 Variable array analysis

Variable array analysis is different from typical variable analysis as we need to look through a range of memory
Example 4  Example 4. Sample code for XNOR test case 1

1 for (i = 0; i < 20; i++) {
2     if (k == i) {
3         a = b + c;
4         b = a + k;
5     } else {
6         b = k;
7         r = b;
8     }
9 }

Example 5 GCD Test example

1 for (i = 0; i < 100; i++) {
2     a[2*i + k1] = ...;
3     ... = a[2*i + k2];
4 }

4.2.1 Loop normalization

Loop normalization is carried out to ensure that correct conditions are achieved by the means of analysis. A loop is assumed to be normal when its stride is 1. For example, a regular loop in which the loop index \( i \) iterates from lower bound \( l \) to stride \( s \) in steps of \( \Delta \), the normalized iteration is written as: \((i-l+s)/\Delta\). The loop body condition will also change accordingly.

Further, to ease the analysis via the range test\[5\] for array references, several pre-processing steps need to be followed:

1) Multi-dimensional arrays need to be converted to single dimensional array. For e.g. \( a[640][480] \), can be expressed as \( a[480*i + j] \), and all the references to \( a[i][j] \) can be expressed as \( a[480*i + j] \).

2) Array references in a loop should be a function of the iterators of the current loop and the inner loops, and the loop invariants. This can be achieved by back tracing the abstract syntax tree (AST) of array reference.

If the array references are the return values of a defined function, then the AST of the function definition is back-traced to its dependency with respect to the iterators. This converts the array reference into a function of iterators and invariants.

4.2.2 GCD test

The GCD test \[14\] is used to identify loop carried dependencies. It works on the basis of solutions to Diophantine equations. In a loop iteration where we have two or more array references, and where at least one reference is a writing operation, the GCD test verifies whether the integral solutions of the equations are possible.

In example 5, there exists a dependency \( i_f \) and only if we can find \( i_1 \) and \( i_2 \) such that \( 2*i_1 + K1 = 2*i_2 + K2 \). The GCD test ignores the bounds of the loop in a simple case like this example. Given a bound, we can parameterize the equations to obtain the range in which there is a possibility of finding integral solutions.

Generating conditions: Firstly we need to determine whether the function \( f(...) \) is either strictly increasing or decreasing in the range of \( i \). For that, let

\[
P(i,j,\text{loop_invariants}) = f(i+1,j,\text{loop_invariants}) - f(i,j,\text{loop_invariants})
\]  

(1)

In order to identify the conditions, we use the symbolic range propagation methods as described in \[6\]. Since the bounds are unknown, we shall get an interval \([a,b]\) where \( a \) is derived from values of \( i,j \) where the curve \( P(i,j,\text{loop_invariants}) \) is minimum, and \( b \) where \( P(i,j,\text{loop_invariants}) \) is maximum. As \( i \) and \( j \) are increasing in nature, the condition thus derived will be \( a > 0 \).

Once it has been determined that the function has a strict positive/negative slope, we need to assess whether for any loop iteration \( i \), the range of memory locations accessed
are different from that accessed by the loop iteration $i + 1$. Using methods similar to the range test, we find the lowest memory location accessed by the index function $f(...)$ during iteration $i + 1$ by $f_{\text{min}}(i + 1, j, \text{loop_invariants})$, and the highest memory location accessed by the index function $f(...)$ during iteration $i$. This establishes whether the index function is increasing or decreasing.

If the difference between these two memory locations is greater than zero, we can identify that the concerned loop is parallelizable with respect to this array variable.

**Example 7 Code Snippet 1:**

```c
1 #pragma parallel for shared(a)\n2   if( (N > 0) && (N > M2) )\n3       for(i = 0 ; i < M1; i++)\n4           for(j= 0 ; j < M2 ; j++){\n5               a[ N * i + j] = ...;\n6         }\n7   }\n```

**Explanation using example 7:** In this example, we have two conditions in an OpenMP construct, which are resolved only during runtime. The following can be concluded by examining example 7:

$$f(i, j) = N * I + j$$

Lowerbound($i$) = 0, Upperbound($i$) = $M1$

Lowerbound($j$) = 0, Upperbound($j$) = $M2$

$$f(i + 1, j) - f(i, j) = N * (i + 1) + j - N * (i) + j = N$$

$f(i, j)$ is strictly increasing given $N > 0$; establishes condition 1. $f_{\text{min}}(i + 1, j)$ is lowest memory access in loop iteration $i + 1$. Thus we have,

$$f_{\text{min}}(i + 1, j) = N * (i + 1) + \text{lower bound of } j$$

$$= N * (i + 1) + 0$$

$$= N * i + N$$

Similarly we have

$$f_{\text{max}}(i, j) = N * (i) + \text{upper bound of } j$$

$$= N * i + M2$$

Difference between equations 2 and 3 needs to be greater than zero to determine memory independence. This establishes condition 2 as shown in equation 4.

$$N * i + N - N * i M2 > 0$$

$$N - M2 > 0$$

$$N > M2$$

Similarly conditions for $N < 0$ where the function is decreasing can be determined.

**Example 8 Code Snippet 2:**

```c
1 #pragma parallel for shared(A)\n2   if ( 2*L1 +1-a > 0 && \n3       *L1 +1 + L2-a-M2 > 0 )\n4       for( i=L1 ; i < M1 ; i++){\n5           for(j=L2 ; j< M2 ; j++){\n6               a[i*i - a*i5 + j] = ...;\n7         }\n8   }\n```

**Explanation using example 8:** This example uses a more complicated array accessing function which uses loop invariants. The following can be concluded by examining example 8:

$$f(i, j) = i^2 - a * i + j$$

where $i$ and $j$ are loop iterators and $a$ is a loop invariant which is a constant.

$$f(i + 1, j) = i^2 + 2 * i + 1 - a * i - a + j$$

Taking a difference of equation 6 and 5 we have

$$f(i + 1, j) - f(i, j) = 2 * i + 1 - a$$

For $f(...)$ to be an increasing function, the condition thus becomes

$$2 * i + 1 - a > 0$$

Thus

$$i > (a - 1)/2$$

$$f_{\text{min}}(i + 1, j) = i^2 + 2 * i + 1 - a * (i + 1) + L2$$

$$= i^2 + 2 * i + 1 - a * i - a + L2$$

$$f_{\text{max}}(i, j) = i^2 - a * i + M2$$

Difference between equations 10 and 11.

$$2 * i + 1 + L2 - a - M2 > 0$$

thus for all values of $i$,

$$M2 < 2 * i + 1 - a + L2$$

$$M2 < 2 * L1 - a + 1 + L2$$
4.2.4 Arrays as loop private variables

The dependency analysis is performed by considering an array variable as private or shared. If the modified values of an array are used after the completion of the loop block, and no dependency exists between any of the array subscripts, then the array is used as a shared variable using the shared OpenMP construct. If array values are not required after loop block, then the array can be used as a private variable. So the parallelization of a loop is still possible if there is a dependency existing within an array, but it is subject to the array not being required after the loop block execution. The parallelization is achieved through privatization of the array [13].

4.3 Variable pointer analysis

Pointers of variables need to be analyzed separately as the methodology to understand the memory contention is different. Our methodology supports a single level of indirection while considering pointer analysis. Memory maps of each OpenMP thread are considered to determine the correct OpenMP constructs. There are three major possibilities while considering pointer access in a loop body which may result in a loop carried dependency:

1) The value stored at the address to which a pointer points is altered. Alias analysis [10][9] is done to capture all the variables which are affected by the pointer and are used in the loop. Further variable analysis [16] is done over these variables to identify loop carried dependencies, if any exist. As a pointer cannot be kept private with the memory still remaining the same, the write and read access of that address is kept atomic.

2) The pointer is not modified, but an arithmetic expression of the pointer is used to modify values different addresses. Pointers of this form can be converted into array form as shown below:

\[ p = \&A; \]
\[ *(p + i) = A[i]; \]

After this conversion, array analysis, as mentioned in section 4.2, is used to check for parallelizability.

3) The pointer is made to point at a different address. The first check is to analyze how many times the pointer has been changed. Separate alias analysis has to be done for each write operation on that pointer. Variable analysis is run over all the aliased variables for identification of loop carried dependencies. Precautions taken for case (1) are also to be considered here.

4.4 OpenMP constructs usage

In this section we look at the OpenMP clauses used for automatic parallelization.

4.4.1 Identification of OpenMP clause of a variable

The previous sections covered the analysis of different types of variables. During these analyses, we identified the appropriate OpenMP clause for each variable. These clauses are subject to the parallelizability of the loop.

Based on its functionality and location, a single variable may be assigned multiple OpenMP clauses. In case a conflict arises due to multiple clauses, the highest priority clause would be used. Clause priority is based on table 1.

<table>
<thead>
<tr>
<th>Clause 1</th>
<th>Clause 2</th>
<th>Result Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>First-private</td>
<td>First-private</td>
</tr>
<tr>
<td>Private</td>
<td>Last-private</td>
<td>Last-private</td>
</tr>
<tr>
<td>Private</td>
<td>Shared</td>
<td>First-private</td>
</tr>
<tr>
<td>First-private</td>
<td>Last-private</td>
<td>Last-private &amp; First-private</td>
</tr>
<tr>
<td>First-private</td>
<td>Shared</td>
<td>First-private</td>
</tr>
<tr>
<td>Last-private</td>
<td>Shared</td>
<td>Last-private &amp; First-private</td>
</tr>
</tbody>
</table>

4.4.2 Structure and union individual element analysis – OpenMP construct

In case of structures and unions, each element is analysed separately, and for every element, an appropriate clause is used. Once the clauses are finalized for each element, a single clause is decided for the structure based on the priorities found in table 1.

4.4.3 Static and dynamic clause

Based on the analysis of the code, a corresponding scheduling clause of OpenMP is used. Here, the focus of analysis is to identify the conditional code existing within a parallelized loop. Based on the analysis of conditions, we determine whether the entire code gets executed. If the code present within a loop is definite\(^3\), then we use a static scheduling clause along with OpenMP’s standard clause in order to parallelize the loop. If there are dependencies on conditional statements like if, while, switch etc. or function calls within a loop block, then we use OpenMP’s dynamic scheduling policy.

\(^3\)That is, not having any dependency during execution on any type of conditional statements like if, while, switch etc. or function calls within the loop block.
4.4.4 If clause

During code analysis, if we come across a condition which needs to be satisfied before the parallelization of code (as mentioned in range test, section 4.2.3), the condition is used inside OpenMP’s if clause to increase the probability of parallelization without losing the correctness of the application.

4.5 Experiments and results

We implemented this methodology on an lane departure warning system. The source code used is ≈2000 lines of code, and was chosen as a representation of computer vision algorithms used currently in automotive systems. As these algorithms pertain to the safety of a driver, it is extremely important to maintain functional correctness while improving its performance through parallelization.

The output obtained was cross-compiled for an i.MX 6 Sabrelite board running embedded Linux kernel 3.0.15. The output obtained was functionally the same for all inputs as compared to the output of sequential code while giving algorithm performance benefit of ≈300%.

5. Conclusion

Multiple automatic parallelization techniques exist, but each one has its own limitations, thus limiting their use to specific applications. We present the YUCCA tool, which addresses lexical analysis in order to find opportunities for parallelization. This paper focuses on:

1) complex loop analysis for optimal utilization of multi-core technology, and
2) ensuring functional correctness, in order to address the primary concerns of safety critical applications.

Using the XNOR test (section 4.1.4), the enhanced range test (section 4.2.3), and the pointer analysis methodologies presented in section 4.3, we analyze variables within conditional branches. Additionally, we handle unknown loop bounds as well as variable pointers within a loop. This paper also covers the usage of OpenMP constructs in order to parallelize loops, maximize resource utilization, and achieve improved throughput.

5.1 Moving forward

We are in the process of optimizing our variable pointer analysis module in order to make it more robust. Additionally, our ability to address function calls within loops is still in its nascent stage. Development in this field would benefit if we could have a standardized test set as well as corresponding benchmarks (both in terms of performance, as well as in terms of source code).

References


Acknowledgements

We would like to thank Subodh Pachghare, Shiva Ghose, Priti Ranadive, Aditi Sahasrabudhe, Smitha K P, Chaitanya Rajguru, Dr. Vinay Vaidya from CREST at KPIT Technologies for their contribution towards review and valuable inputs.