1 Introduction

One of the most effective ways for improving performance of an existing software is to transition sequential code to parallel code. Unfortunately, correct concurrent code requires a lot of additional complexity. This means such transition is very costly in terms of development time (and usually, concurrency related issues that are only revealed later).

One approach is to focus only on limited aspects of the program, hopefully achieving significant performance gain in relatively small effort. When aiming for automatic process, loops in the program are a natural choice: they are usually costly in terms of run-time, and are easier to transform to parallel form.

Our project is a tool that attempts to boost performance of existing programs by automatically modifying the program's code to perform loops in parallel, whenever possible[2]. The transformation is performed by partitioning, meaning - there would be multiple threads running the same code, each runs over different partition of the loop inductive variable. The second part of the project is to analyze the results and understand the effect of running the tool. In a high level view, the project design can be described by those three major steps:

- **Analyze** the program source code and find loops that can be transformed to run in parallel.
- **Modify** the source code: apply loop parallelization.
- **Profile** the resultant program using a sample input provided by the end-user.

2 Dependency analysis

2.1 Motivation and Introduction

Before we can change loops behavior from linear to parallel, we must have confidence that the program result will not change. Changes in the output usually mean breaking functionality, which we do not want.
Since our parallelization approach is based on loop partitioning, outcome of the code inside a loop can be affected if two different loop iteration are accessing the same data. For example, let's take a look at the following code:

```
for(int i = 1; i < 10000; ++i)
{
   A[i] = B[i] * 5;
}
```

It is trivial to see that the data read and written in each iteration is different, so there would be no problem if we modified the program to open multiple threads, each thread handling a partition of the loop bounds. On the other hand, what happens in the following example?

```
for(int i = 1; i < 10000; ++i)
{
   A[i] = B[i] * 5;
    B[i + 1] = A[i];
}
```

Can we parallelize this loop?

To solve this question, we will look for loop-carried dependencies, which are cross-iteration dependencies with respect to array access and private variables. In the example above, we can see that since $A[i]$ depends on $B[i]$, which itself depends on $A[i-1]$, we have a dependency between loop iterations.

We are using LLVM's[6] DependenceAnalysis pass[3], which goal is to successfully analyze loop dependencies in *most* cases. The assumption is that most array references in real-world code are simple (index-wise) and can be covered by a small set of special tests (described above)[5]. When we are not sure or cannot know, we assume dependency (a few examples: non-linear array reference; variables with dynamic user input). The analysis is **conservative**, meaning the code is always correct but there could be false positives.
We will take a few simplifying assumptions:

- We are only working with integer variables for loop index variables and variables used for array reference access
- The loops are **normalized**, meanings - they have an inductive variable that goes between 1 and N (see the "Implementation" section to see why this assumption is legitimate).

2.2 Definitions

Definition 1 ***Index*** *is a loop variable.*

Definition 2 ***Subscript*** *is the pair of expressions that appear in a certain coordinate in a pair of array references.*

For example:

- In $A[i] = A[i]$, $< i, i >$ is a subscript.
- In $A[j+1] = A[i]$, $< j+1, i >$ is a subscript.

When dealing with multi-dimension array reference, there would be multiple subscripts, one for each dimension.
For example, the subscripts in the expression $A[i,j,k] = A[i+1,j,k-1]$ would be $< i, i+1 >, < j, j >$ $, < k, k-1 >$.

Definition 3 *The relation between the elements in each subscript pair can be described by* ***distance*** *and* ***direction****.*

For example:
For the subscript $< i, i+1 >$, the distance is 1 and the direction is $<$. Notice that since array references does not have to be linear (e.g. $A[i*i+5]$), distance is not always available. When dealing with subscripts vector, we will use distance and direction vectors, where each subscript has a corresponding direction and distance.

Direction vectors can be merged using Cartesian product, as described below.

Definition 4 *When dealing with multidimensional arrays, we say a subscript position is* ***separable*** *if its indices do not occur other subscripts. If two different subscripts contain the same index, we say they are* ***coupled****.*

For example:

- $< i, i >, < j, j >$ are separable, as no variables are shared between the pairs.
- $< i, k >, < i, i+k >$ are coupled, as the variable i appears in both subscripts.

When testing a **coupled group** of subscripts, we will handle coupled subscripts as a group, otherwise we might suffer imprecise output.

For example:

```
for (int i = 1; i < 100; ++i) {
  A[i + 1][i] = B[i] + C;
  D[i] = A[i][i] * E;
}
```

In this case, as all subscripts are coupled, we must have a system of equations with all subscripts, not two stand-alone equations.

Given instruction S and T that access the same memory location, there are 4 types of dependencies:

Definition 5 *If S is a write and T is a read, this is a* ***flow dependence***

Definition 6 *If S is a read and T is a write, this is an* ***anti-dependence***

Definition 7 *If S is a write and T is a write, this is an* ***output dependence***

Definition 8 *If S is a read and T is a read, this is an* ***input dependence***

For our purpose, input dependence is not usually important and we will ignore it.

Definition 9 *Given a subscript, the* ***subscript complexity*** *can be classified to* ***ZIV*** *(no indices),* ***SIV*** *(one index) or* ***MIV*** *(multiple indices) according to the number of loop index variables in the expression.*

2.3 Algorithm

Algorithm 1 Loop Dependence Analysis

```
1:  Partition subscripts into separable and coupled groups
2:  Classify each subscript as ZIV, SIV or MIV
3:  for each separable subscript do
4:      Apply single subscript test according to its complexity
5:      if independence established then
6:          Output "no dependence" and halt
7:  for each coupled group do
8:      Apply multiple subscript test
9:      if independence established then
10:         Output "no dependence" and halt
11: Merge all direction vectors computed in the previous steps into a single set of direction vectors
```

Input: a pair of array references and loop bounds.
Output: "no dependence" or direction vector.

2.3.1 Notes

In steps 3 and 4, it is correct to halt if we prove independence in any group: we are dealing with multi-dimension array reference, so if we do not refer to the same element in one dimension, we obviously do not refer the same data.

2.4 Dependency Tests

The approach is to test subscript-by-subscript, calculating the dependency between each pair. Eventually, intersect the results of of the tests to give a binary answer (no dependence, or - dependence found, and if available - dependency distance or direction).
One approach is the GCD test, which can handle any subscript pair, but has a limited precision due to a few limitations. Alternatively, when certain conditions are met, we can use other kinds of tests such as ZIV, SIV and MIV that give more efficient and precise answer.

2.4.1 GCD Test

In its most general form, testing dependency of a subscript pair means finding out if there are possible values for which both array references might access the same array index.
Calculating the possible values is simple - compare the subscript pair elements, with the loop index variables as parameters within the bounds of the loop. Since we are dealing only with integers, only integer solutions are acceptable for the resulting equation.
For example:

```
for(int i = 1; i < 100; ++i)
{
   A[i] = A[i - 1] + 1;
}
```

In this case, it is easy to see there is a dependency. Our equation would be $i = i' - 1$. It is easy to see that, for example, $i = 1, i' = 2$, and $1 \geq 1, 2 \geq 100$ so the solution is within the loop bounds.
Let's take a look at a more complex case:

```
for(int i = 0; i < 100; ++i)
{
   for(int j = 0; j < 100; ++j)
    {
       A[6 * i + 5 * j - 10] = ...;
        ... = A[i];
    }
}
```

Our equation is not as simple as before: $6 * i + 5 * j - 10 = i'$, or $6 * i - 1 * i' + 5 * j = 10$. In the general case, after switching sides we are looking at the problem of finding all integer solutions for the following equation: $\sum_{i=1}^{n} a_i * x_i = c$
Where $x_1 ... x_n$ are the loop variables, $a_1 ... a_n$ are the coefficients, and c is a constant integer. Solution exists for this Diophantine equation if and only if the greatest common divisor (GCD) of all coefficients $gcd(a_1, a_2, ...)$ divides c. Unfortunately, there are a few disadvantages for using the GCD test:

- GCD test ignores loop bounds, which could lead to false positives (see example below)
- The GCD is often 1, in which case the solution is conservative and could be a false positive
- GCD test result does not provide distance or direction

```
for(int i = 0; i < 10; ++i)
{
   A[i] = ...; /* S1 */
    ... = A[i - 100]; /* S2 */
}
```

In this case, GCD test will return a positive answer since: $gcd(1, -1)$ divides 100. It is easy to see that there is no dependency in this loop as $0 \geq i \geq 10$, therefore $S1$ and $S2$ cannot possibly access the same memory. This is a simple example that demonstrates the limitations of the GCD test.

2.4.2 ZIV (Zero Index Variable)

This is the simplest case. Both subscripts are loop invariants (the relevant detail - the subscripts are not affected by the loop index). In this case, the test is to see if, according to our knowledge of the program, the pair elements might be equal.

Examples: Let a, b variables that are not the loop variable. They can be function parameters from unknown source, result of I/O operation, user input, etc.

- $A[b] = A[b+1]$ - we can tell for sure there is no dependency.
- $A[a] = A[b]$ - possible dependency as we do not know the values of a and b.

2.4.3 SIV (Single Index Variable)

The subscripts contain one induction variable. This could be a part of a more complex expression (e.g. if j is the loop index, it could be $A[j]$ or $A[j*2+5]$).
Let i, i' be the loop variable, let a_1, a_2, c_1, c_2 be integer scalars, let U, L be the upper and lower bounds of the loop variable.
Assuming the subscript pair contains a polynom of degree 1, we can express the subscript as $< a_1 * i + c_1, a_2 * i' + c_2 >$. Essentially, it is possible to get an exact answer by finding all integer solution to the following Diophantine equation
$a_1 * i + c_1 = a_2 * i' + c_2, L \geq i, i' \geq U$
Dependence exists for all integer solutions of the equation, so if there is a solution within the range of the loop, we can prove dependence. Unfortunately, solving general integer programming equations is a hard problem. For a few common cases detailed below we can find a solution quite efficiently. For other cases, we will run an exact SIV test.

2.4.4 Strong SIV subscript

A subscript of the form $< a * i + c_1, a * i + c_2 >$, where the coefficients are equal, is called a strong SIV subscript. In this case, we define the dependence distance to be $d = \frac{c_1 c_2}{a}$.
Loop dependency exists if d is an integer and $|d| \leq U - L$. The direction can also be calculated according to d.

2.4.5 Weak SIV subscript

A subscript of the form $< a_1 * i + c_1, a_2 * i + c_2 >$, is called a weak SIV subscript. In this case, exact dependency calculation is possible by solving the Diophantine equation described above.

2.4.6 Weak-zero SIV subscript

This is a special case of weak SIV subscript, where either a_1 or a_2 are zero (but not both - in this case it is simply a ZIV case).
In this case, without loss of generality, let $a_2 = 0$, and the equation is now $a_1 * i + c_1 = c_2$. Therefore, solution exists if $i = \frac{c_2 - c_2}{a_1}$ is an integer and is within the loop bounds.

2.4.7 Weak-crossing SIV subscript

Another special case of weak SIV subscript, where $a_1 = -a_2$. It is called "crossing" because when looking at the problem geometrically, the dependence point (if exist) is the point where the lines representing each subscript on a 2-d plane cross each other. In this case, the solution for the dependence equation is $i = \frac{c_2 - c_1}{2 * a_1}$. If we have a solution i that is multiplication of 0.5 and within the loop bounds, dependency is proven.

2.4.8 MIV (Multiple Index Variable)

The subscripts contain multiple index variables. In general cases, dependency could be found using GCD test. In a few common cases, there are more efficient solutions - as described below.

2.4.9 RDIV (Restricted Double Index Variable)

This is a special case of MIV, where the subscript pair is in the form $< a_1 * i + c_1, a_2 * j + c_2 >$. For this special case, the exact SIV test can be extended to test for dependence in this test.

2.4.10 Symbolic RDIV

This test provides another way to handle the MIV case where the subscript pair is in the form of $< a_1 * i + c_1, a_2 * j + c_2 >$. The basic idea is that $c_2 - c_1$, the difference between the constant terms of each subscript expression, can be formed symbolically and simplified. The result may be used like a constant.

The basic condition for dependency, as usual, is that $c_1 + a_1 * i = c_2 + a_2 * j$ for some $L_1 \geq i \geq U_1, L_2 \geq j \geq U_2$. Without loss of generality, we'll assume $a_1 \geq 0$, and we may now break this into cases:

Assume $Sign(a_1) = Sign(a_2)$. In this case, the expression $a_1 * i - a_2 * j$ reaches maximum value of $i = U_1, j = 1$, and minimum value for $i = 1, j = U_2$. Therefore, dependency exists only if: $a_1 - a_2 * U_2 \geq c_2 - c_1 \geq a_1 * U_1 - a_2$.

Assume $Sign(a_1) \neq Sign(a_2)$. In this case, the expression $a_1 * i - a_2 * j$ reaches maximum value of $i = U_1, j = U_2$, and minimum value for $i = 1, j = U_2$. Therefore, dependency exists only if: $a_1 - a_2 \geq c_2 - c_1 \geq a_1 * U_1 - a_2 * U_2$.

The upside of this test is that i and j could be the same variable, so it can serve as a fallback when the RDIV test does not disprove dependence. The downside - the test does not find distance or direction, it can only be used to disprove dependency.

2.4.11 Merging the Results

Merging the direction vectors of all tests (in the last step of the algorithm) is practically a Cartesian product, as each separable and coupled subscript group contains a unique subset of indices.

For example:

```
int c = ...;
int A[...][...];

for(int i = 1; i < 10; ++i)
{
   for(int j = 1; j < 10; ++j)
    {
      A[i + 1][j] = A[i][j] + c;
   }
}
```

In this example, the first subscript pair $< i+1, i >$ returns the direction "<". The second pair $< j, j >$ returns the direction "=". The resulting Cartesian product is a direction vector: $\{<, =\}$.
Another, more complex example:

```
for(int i = 1; i < 10; ++i)
{
   for(int j = 1; j < 9; ++j)
    {
      A[i + 1][j] = B[i] + NUM;
        C[j] = A[1][10 - j];
   }
}
```

In this example, the first subscript pair $< i+1, i >$ returns the direction "<". The second pair $< j, 10-j >$ returns the direction vector "$\{< \ . =, >\}$". The resulting Cartesian product is now a more complex vector: $\{(<, <), (<, =), (<, >)\}$.

2.5 Implementing Dependence Analysis

Our implementation of dependency analysis relies on the LLVM framework, and is written as an LLVM pass. We are using its dependency analysis pass, together with a few more required passes. Combining the information from those passes, we can output a list of loops that can be parallelized.

2.5.1 Supporting Passes and Transformations

Those are multiple LLVM passes used by both the DependenceAnalysis pass and our pass for analyzing the code.

- The "**Loop Information**" pass is a basic pass that finds loops in the code, by analyzing the call flow graph.
- The "**Canonicalize Induction Variables**" pass makes working with loops easier, as all loops are transformed to have a single induction variable which starts at zero and steps by one, as well as changing pointer arithmetic operations to work with array subscripts.
- The "**Canonicalize natural loops**" greatly helps analysis by providing loop pre-header, and making sure all exit blocks from the loop only have predecessors from inside the loop.
- The "**Scalar Evolution**" pass is used to analyze and categorize scalar expressions in loops. It provides valuable information required for any loop analysis.

2.5.2 The DependencyAnalysis Pass

The dependency analysis pass, probably for performance reason, is only performed on demand. Meaning - the pass itself does nothing when invoked on function. Instead, it is invoked when another pass is calling its *Depends* method. The dependence test input is two instructions, referred to as "source" and "destination". The reason for those names - those are the source and the destination nodes in the dependence graph, and the output of the test is the directed edge between them (if there is one). Only simple store and load instructions (simple means âĂŞ not volatile or atomic operations) are tested, the rest are ignored.
The pass is performing number of tests on the source and destination instructions, trying to get an output that is as precise as possible (e.g. SIV test is much preferable over a GCD test). If precise methods are not suitable (or failed to work), less precise tests will be used. The pass is implementing the dependence analysis algorithm described above and returns whatever dependencies that were found.

If possible, a direction vector is returned (but it is not guaranteed!). It is possible that no dependency was found, but it was not proven that there is no dependency. In this case, the result will be *confused*, meaning - we do not know.

2.5.3 The IndependentLoop Pass

The IndependentLoop pass is the one we implemented for the project. This is a function pass, which means - it is working within the scope of a function. For each loop in the function, dependency analysis is performed on each pair of relevant instructions.

If no dependencies are found, the source line for the beginning and the end of the loop is printed (as LLVM is working with compiled bitcode, we are depending on debug symbols to find the source line numbers). Since we wanted to make sure there are no degradation bugs in the target program, we are counting "confused" results as a dependency.

For simplicity, nested loops are ignored, meaning - we will not try to perform parallelization of loops with depth of more than 1.

We found working with LLVM bitcode and outputting reference to source lines to be quite a challenge. Even when the target files are compiled with debug symbols, we could not reliably provide the exact lines in the source code of each loop's closing brackets. We had to compensate for this limitation by implementing additional Python code as part of the tool run.

3 From Sequential to Parallel

3.1 OpenMP

Following the dependency analysis, we have the information about potential loops we can transform to parallel. For the actual transformation, we used the **OpenMP** framework[4].

3.2 Implementation

We created a Python tool that puts all the pieces together: dependence analysis, parallelization of eligible loops, and profiling the results.

Input: List of target source files to parallel, command for building the source files, sample input used for profiling.

Output: New target source files with parallelized loops.

The process is as follows:

1. Compile target files to LLVM bitcode
2. Perform dependence analysis on the bitcode-compiled target files
3. Inject *OpenMP* parallelization code for all loops that are safe to parallel
4. Inject timing code before and after each loop that was parallelized
5. Compile and run the target code
6. Read and parse the timing output
7. Revert the modified files. Inject parallelization code again, but now only for loops where performance enhancement was observed

Test Name	Description
Application: TinyJPEG	This application benchmark performs decoding of JPEG images with fixed encoding of
Kernel: Ray Tracing	This is a kernel-type benchmark of a very simple and brute-force ray tracer.
Kernel: k-means Clustering	This is a kernel-type benchmark of a simple off-line clustering algorithm.
Kernel: Message-Digest 5	This is a kernel-type benchmark measuring the time it takes to MD5 some predefined b

Table 1: Programs from the Starbench suite used for evaluation

4 Evaluation

During development, we used a few short program for testing. This gave us some confidence in the code, but we wanted to evaluate the tool against a few real-world numerical programs as well.
We selected a standard benchmark named **Starbench parallel benchmark suite**[1], which includes multiple numerical programs, implemented in C/C++/Fortran, covering aspects such image processing, artificial intelligence, compression and hashing. The benchmark offers three different implementation of each program - sequential, multi-threaded, and a specialized programming language named OPMSS.
We used the sequential implementations as a baseline, for both comparing the run time results and making sure the correctness of the program (meaning, the same output) was preserved.
See table 1 for complete list of tests.

4.1 Results and Analysis

Evaluation of the tool on the benchmark programs showed no performance improvement. For all non-trivial tests, results with and without optimization were almost identical.
Analyzing the optimization process we used, it was easy to understand the results - our tool did not parallelize any significant part of the program. The conservative approach we took was very important for avoiding incorrect programs, but on the other hand - any loop that contained non-trivial code was ruled out. Therefore, the impact of the parallelization was very little.

For example, looking at the c-ray (Ray Tracing) test, the following loop is parallelized:

```c
for(i=0; i<NRAN; i++) urand[i].x = (double)rand() / RAND_MAX - 0.5;
```

This loop initializes an array of size 1024 with random numbers. Obviously, with a modern processor this loop takes almost no time; parallelizing it only applies the overhead of forking and joining the program. On the other hand, this line of code is doing the actual, heavy lifting:

```c
render_scanline(xres, yres, i, (uint32_t*)((void*)pixels + i*xres*sizeof(uint32_t)),
    rays_per_pixel);
```

When running the optimization on a hand-crafted test, we could see that it has the potential to provide a very significant performance boost. Unfortunately, for real-world programs, it seems like more work is required.

5 Discussion

This was a very interesting project that involved a lot of new materials: dependence analysis, LLVM and OpenMP, all of them were new for of us. This was challenging, but we learned a lot in the process. The evaluation results were a bit disappointing, we hoped that the result will be able to improve real-world programs as well, but we understood an important insight about the hardship of automatic optimization.

5.1 Future Development

We thought of a few possible directions could improve the tool's results. One obvious direction is to request the **programmer's assistance** - hints from the programmer (e.g. code annotations) could assist in finding code to parallelize. On the other hand, if they had the intention to create a concurrent program in the first place, they could have used OpenMP already.
Since we wanted to stick to automatic measures, we could **transform the code** - in many cases, it is possible to apply code transformations to change dependent loops to independent loops. For example, if the dependence is caused because of a single iteration, some times it is possible to split the loop code so this iteration will run in a different code block and the rest of the loop will be parallelized. We found a lot of such examples during our research, from simple code modification to sophisticated algorithms that identify existing patterns (e.g. binary search) and replace them with parallel code. Unfortunately, we did not have time to experiment in this direction.

6 Appendix: Using the Tool

6.1 Requirements

6.1.1 Requirements for Running the Tool

- Linux-based operating system (we are using shell commands as part of the program). We used **Ubuntu** for our development environment, but any Linux will do
- LLVM version 3.6 (must have the correct version, as LLVM is not always backward-compatible
- OpenMP framework
- Python 2.7x
- C compiler

6.1.2 Requirements for Building the Tool and the Samples

- Modern C++ compiler
- Clang compiler (version 3.6) - for building the samples
- LLVM 3.6 header files
- CMake version 2.8 or above

6.2 Running the Tool

From the command line, run the following command:

```
python ./autoPar.py --run_cmd=BENCHMARK_RUN_CMD --build_cmd=BUILD_CMD --src_dir=CODE_SRC_DIR
```

Where:

- **runcmd** - command to run the benchmark
- **buildcmd** - command to build the target files
- **srcdir** - the target source directory

```
python ./autoPar.py --run_cmd="bash ../benchmark/run_benchmark.sh" --build_cmd="bash
    ../benchmark/builed.sh" --src_dir="../benchmark/"
```

It is also possible to run just the dependency analysis pass using the **opt** command line tool:

```
opt-3.6 -load <Independent Loop Pass Binary> -basicaa -mem2reg -simplifycfg -loop-simplify
    -loop-rotate -instcombine -indvars -indloop <Target File> -o /dev/null
```

References

[1] Michael Andersch, Ben Juurlink, and Chi Ching Chi. A benchmark suite for evaluating parallel programming models. In *Proceedings 24th Workshop on Parallel Systems and Algorithms*, 2011.

[2] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A. Padua. Automatic program parallelization, 1993.

[3] Preston Briggs. Llvm dependence analysis pass.

[4] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[5] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. *SIGPLAN Not.*, 26(6):15–29, May 1991.

[6] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.