Automatic Loop Parallelization

PAAV 2015 Final Project

Submitted by: Yogev Vaknin, Itay Polack

Lecturer: Noam Rinetzky

Agenda

- Motivation
- Introduction: loop parallelization
- Short introduction to dependency analysis
- Short introduction to OpenMP
- Project implementation
- Evaluation
- Discussion

Motivation

- We would like to improve software performance
- We want a **significant** improvement
- We want **minimal development effort**
- The program output **must not change**
- The program performance **must not degrade**

Idea

- Focus on specific program aspects that are -
  - Common
  - Resource demanding (usually)
  - Easy to improve (some times)
- Loops!
  - Common structure
  - Many iterations running the same code
  - Long loops have significant performance cost

How?

- Many existing optimizations
  - Extract loop invariant code outside the loop
  - Loop unrolling
  - Perform the loop concurrently
  - … and many more!
- **Perform the loop concurrently**
  - Many identical threads
  - Each one is running a partition of the loop iterations

Loop Parallelization

Before

- Run the loop for N iterations

After

- Create X threads, each one is running N / X iterations
- Wait for all threads to finish

Example - Code

Before

```
for (int i = 0; i < 100000; ++i) {
  A[i] = B[i] / 2;
}
```

After

```
void loop_thread(int start, int end)
{
  for (int i = start; i < end; ++i) {
   A[i] = B[i] / 2;
  }
}
```

```
pthread_create( &thread1, NULL, loop_thread, (void*) {
.start = 0, .end = 10000 });
pthread_create( &thread2, NULL, loop_thread, (void*) {
.start = 10001, .end = 20000 });
// ... Open as many threads as required ...
pthread_join( thread1, NULL);
pthread_join( thread2, NULL);
```

The Dependency Challenge

- Many loops have cross-iteration dependencies
- Example:

```
int i, fib[1000];
fib[0] = 1, fib[1] = 1;
for (i = 2; i < 1000; ++i) {
   fib[i] = fib[i - 1] + fib[i - 2];
}
```

- Each iteration depends on the previous two iterations
- What happens if we try to parallelize this loop?

What is a Dependency?

- Two different instructions accessing the same memory means there is a **dependence** between them
- Given two program instructions, T and S, that access the same memory location:
  - If S is a write and T is a read, this is a **flow dependence**
  - If S is a read and T is a write, this is an **anti-dependence**
  - If S is a write and T is a write, this is an **output dependence**
  - If S is a read and T is a read, this is an **input dependence**

Dealing with Cross-Iteration Deps

- Replace dependent code with independent code
  - For example: implicit formula for Fibonacci sequence
  - Efficient, but hard to implement and fits very few cases
- Attempt to resolve cross dependencies
  - For example: if a single iteration causes dependence, run it beforehand and parallelize the rest
  - Efficient in many cases, but hard to implement and sometimes risky
- **Ignore dependent loops**
  - Simple!

Dependency Testing: Approach

- Conservative
  - "No dependence" result means: we proved there are no cross iteration dependencies
  - "Dependence found" result means: there *might* be a dependency
- Practical
  - Try to cover **most** cases by focusing on common and easy-to-solve scenarios
  - Assume dependency in complex cases
- Result is correct, but may be less than optimal

Dependency Testing: Method

- Focus on **array reference** operations
- Scalar variables can be treated as a single-element arrays
- Pointer access can sometimes be handled by alias analysis

Basic Definitions

- **Index** is a loop variable

For example:

for (j = 0; j < 100; ++j) { … }

j is the *index*

- **Subscript** is a *pair* of expressions that appear in certain coordinate in a pair of array references

For example:

A[i, j, k] = A[i + 1, j, k - 1] + 100;

The *subscripts* are <i, i + 1>, <j, j>, <k, k - 1>

Basic Definitions

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

For example:

For the subscript <j, j + 1>, the distance is **1** and the direction is **<**

- When there are multiple subscripts, we will use a **direction/distance vector**, where each element matches the corresponding subscript
- Merging direction vectors (in our domain): Cartesian product
  - There might be dependencies in multiple directions at the same time

Dependency Tests

- Given a subscript pair -
  - Is there a dependency?
  - Can we calculate the dependency direction?
  - Can we calculate the dependency distance?
- The general case is **undecidable**
- A complex example:

```
scanf("%d", &var);
for (i = 0; i < 100; ++i) {
  for (j = 0; j < 100; ++j) {
    A[var + pow((i + j), 5)] = A[pow(i, j) + i * i - 5000];
  }
}
```

Dependency Tests

- Most programs use simple subscripts
  - Assume integers only
  - Assume linear subscript expressions only
  - For the rest, assume dependency
- Problem can be reduced to solve a simple equation: are there index variables for which the subscript elements get the same value?
- For example:

```
for (int i = 0; i < 100; ++i) {
  A[i] = A[2 * i + 1];
}
```

Can be reduced to the following **Diophantine equation**: i= i' * 2 + 1

Dependency Tests: GCD Test

- Solving a system of Diophantine equations is NP-complete
- But, we can use a simple algorithm to know if such solution exists: GCD test
- Given an equation of the form $a_1 * x_1 + a_2 * x_2 + \ldots + a_n * x_n = c$, for $x_1 \ldots x_n$ loop indices, $a_1 \ldots a_n$ coefficients, and c constant, solution exists if and only if the greatest common divisor (GCD) of all coefficients divides c
- Limited
  - Ignores loop bounds, leading to false positives
  - The GCD is 1 in many case, leading to false positives
  - Binary answer, no direction or distance

GCD Test: Examples

```
// gcd(5, 10) does not divide 139 - no dependency
for (i = 1; i < 100; ++i) {
  for (j = 1; j < 100; ++j) {
    A[5 * i] = A[-10 * j + 139];
  }
}
```

```
// gcd(1, -1) divides 100, but there is no dependency due to loop bounds
for (i = 0; i < 10; ++i) {
  A[i] = x;
  x = A[i - 100];
}
```

Dependency Tests: More Tests

- GCD covers a lot of cases, but is quite limited
- Banerjee test is an variation on the GCD test that can imply bounds
- When certain constraints are met, specific tests can be used, depending on the **subscript complexity**
  - ZIV subscript is a subscript with no indices
    - Both expressions are loop invariants
  - SIV subscript contains a single index
  - MIV subscript contains multiple indices

Dependency Tests: Specific Tests

- ZIV, SIV and MIV tests assume specific subscript form, for which they can sometimes provide distance and direction vectors
- For example, the Weak-SIV test assume the subscripts to be in the form of $< a1 \ * i + c1, a2 \ * i + c2 >, a1 = 0 \ OR \ a2 = 0$
- In this case, dependence exists if $i = \frac{c2 - c1}{a1}$ exists, is an integer, and within the loop bounds
- Since most expressions used in real-world programs are simple, those tests cover most of the cases
  - About 85% according to study performed on many scientific Fortran programs

Separable and Coupled Subscripts

- A subscript is **separable** if its indices do not occur in other subscripts
- If two different subscripts contain the same index they are **coupled**
- Dealing with coupled subscripts requires additional care, as naïve approach could cause imprecision
- For example, in this code we have multiple subscripts sharing the same index variable:

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

- Separate calculation might lose crucial data and lead to incorrect output

Separable and Coupled Subscripts

- All coupled subscripts must be tested together
- For example: A[j – 1][i + 1][I + 3][k] = A[j + 2][j][i][k + 1]
- Separable subscripts: <k, k + 1>
- Coupled subscripts: <j – 1, j + 2>, <i + 1, j>, <i + 3, i>
- For the coupled subscripts, all group must be tested together:

$$\begin{cases} j - 1 = j' + 2 \\ i + 1 = j' \\ i + 3 = i' \end{cases}$$

Dependence Analysis: Suggested Algorithm

- Input: a pair of array references and loop bounds

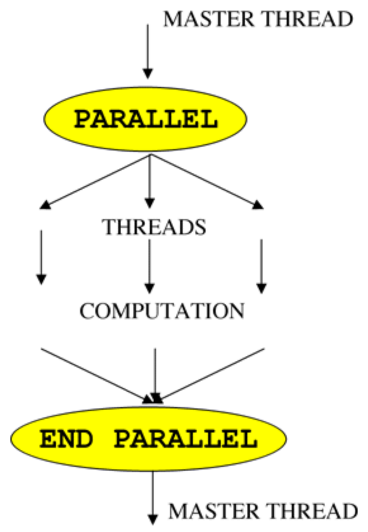
1. Partition subscripts into separable and coupled groups
2. Classify each subscript as ZIV, SIV or MIV
3. For each separable subscript: apply dependency test
    1. If independence established, return “no dependence” and halt
4. For each coupled group: apply dependency test
    1. If independence established, return “no dependence” and halt
5. Merge all direction vectors computed in previous steps into a single step of direction vectors

OpenMP: Introduction

- Framework for shared-memory and multithread programming
- Simple and flexible interface for developing parallel applications
- Supports C, C++ and Fortran
- Supports many architecture and operating systems
    - Even supercomputers!
- Transforms sequential code to parallel code using simple compiler directive and commands

OpenMP: Model

- Fork-join model
- Master thread, parallel sections
  - Master thread spawns threads when reaching parallel section
  - Parallel sections are performed simultaneously
  - Master thread wait for all threads to finish before proceeding
- Performance cost
  - Fork and join operations
  - Possibly less efficient usage of cache



OpenMP: Hello World Example

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

int main (int argc, char *argv[]) {
  int th_id, nthreads;
  #pragma omp parallel private(th_id)
  {
    th_id = omp_get_thread_num();
    printf("Hello World from thread %d\n", th_id);
    #pragma omp barrier
    if ( th_id == 0 ) {
      nthreads = omp_get_num_threads();
      printf("There are %d threads\n",nthreads);
    }
  }
  return EXIT_SUCCESS;
}
```

OpenMP: Loop Example

```
#include <cmath>
int main()
{
  const int size = 256;
  double sinTable[size];

  #pragma omp parallel for
  for(int n=0; n<size; ++n)
    sinTable[n] = std::sin(2 * M_PI * n / size);

  // the table is now initialized
}
```

Putting It All Together

- Our tool is putting everything together, aiming for automatic process of parallelizing loops
- **Input**: a list of target source files, input for profiling the resulting program
- **Process**:

1. Analyze the source code for loops that can be transformed to work in parallel
2. Use OpenMP for transforming eligible loops
3. Build and profile the transformed programs using the provided input
4. Select loops to parallel only if the performance improved

Implementation: Technical Details

- The tool was implemented in Python
- Dependency analysis was written as LLVM pass, implemented in C++
- Requirements:
  - Linux-based operating system
  - LLVM 3.6
  - OpenMP
  - Python
  - C compiler

Implementation: Dependency Analysis

- Written in C++, as LLVM pass
  - Analysis only
- Uses LLVM's DependenceAnalysis pass for finding dependencies
  - For each loop, apply dependence testing for each pair of store/load instructions
- Output loops that are eligible to run in parallel
  - Meaning: independence was proven
- Very cautious
  - Skip nested loops
  - Only suggest loops it is completely sure about

Implementation: Transforming to Parallel

- Written in Python
- Transformation is done in source level, by injecting special code
- Apply transformation for each loop suggested by the LLVM pass
  - Use OpenMP’s special pragma for the parallelization: “pragma omp parallel for”

Implementation: Profiling

- Written in Python
- Inject timing code before and after each loop that is candidate for parallelization
- Run each program twice: with and without parallelization
  - Rely on user's input
  - Collect timing information and evaluate effect of parallelization
  - Effect is evaluated per loop, not per program

Evaluation

- Sample programs we created as part of the development process
- **Starbench**: parallel benchmark suite
  - A collection of numerical programs, covering multiple subjects
    - AI algorithms
    - Image processing
    - Compression
    - Etc.

Evaluation Results

- **No improvement over the baseline**
- No significant loops, in terms of performance, were parallelized
  - The conservative nature of our dependence analysis made us ignore most loops, and all significant ones
  - Due to OpenMP’s overhead, there is no gain when trying to parallelize small or insignificant loops

Discussion

- In its current form, the automatic process provides little to no benefit for real-world scenarios
  - Complex code inside the loop will almost always lead to dependency assumption
- Idea: Improve the dependency testing – can we get less false positives?
- Idea: Remove dependencies by changing the code
  - Very simple in some cases, feasible in many cases
  - By applying such changes, we might reduce dependencies
- Great learning experience
  - Challenging project
  - We learned a lot about LLVM, OpenMP, and the subject of dependence analysis