Introduction to parallel loops

- **Loop statements** are used for implementing repeated computation, but when used extensively or carelessly, they produce performance inefficiencies.

- Modern CPUs provide critical performance improvement through **parallelism**, but software must be written specifically to utilize this available multicore hardware.

This research presents a novel algorithm for automatic program parallelization based on loop splitting. Using flow-dependency analysis inspired by Implicit Computational Complexity theory, the algorithm detects opportunities for splitting loops horizontally into smaller, parallelizable loops, then automatically applies this optimization. The transformation is **semantic-preserving**, which ensures the program behavior remains unchanged.

We hypothesize combining this algorithm with OpenMP [1], an existing state-of-the-art shared memory multiprocessing programming model, will provide noticeable performance gains for resource-intensive computational tasks.

Our technique pictorially

**Step 1.** Identify flow-independencies

**Step 2.** Split computation horizontally

**Step 3.** Parallellize

Implementation and examples

Development is currently underway for an open-source tool implementing this technique on a subset of C programming language.

Before optimization

```c
void main() {
    int a[100], b[100];
    for(int i = 0; i < n; ++i){
        a[i] = i;
        b[i] = i * 2;
    }
}
```

After optimization

```c
void main() {
    int a[100], b[100];
    # pragma omp parallel for
    for(int i = 0; i < n; ++i)
        a[i] = i;
    # pragma omp parallel for
    for(int i = 0; i < n; ++i)
        b[i] = i * 2;
}
```

Pseudo code

```
OPTIMIZE(c_program : str) -> [str]

Parse C program into Abstract Syntax Tree (AST)

Recursively for each loop in AST:
1. Find primary dependencies of loop body variables
2. Build a directed graph of dependencies
3. Compute strongly connected components (SCC)
4. Separate SCCs into subgraphs
5. Split the loops based on subgraphs
6. Insert each split loop into AST

Reconstruct C program from AST using C generator

Return optimized program
```

Source code: https://github.com/statycc/pyalp

Preliminary benchmarks

Program wall time, as measured on example `two_arrays`, and GCC optimization level –O0, showing average speedup of factor 3.4, on executing machine: Darwin kernel v. 20.6.0, i386 processor, with 4 cores and 16 GB RAM.

Conclusion

State of the art tools for loop optimization techniques have limitations, e.g., `while` loops are not optimized [2,3], and algorithm redesign and manual effort is required to create potential for parallelism [4].

Our technique has potential to address these deficiencies: it is loop-type agnostic, identifies parallelization opportunities automatically, and has already demonstrated preliminary performance gains.

Open questions and future work

- Development of cost-benefit analysis to ensure gain
- Implementation need to be extended to support for richer C language syntax, but algorithm and prototype already exist
- Completing further benchmarking and measurements

References


