

Distributing and Parallelizing Non-canonical Loops

Clément Aubert¹, Thomas Rubiano²,
Neea Rusch¹, Thomas Seiller^{2,3}

¹ Augusta University

² LIPN - Laboratoire d'Informatique de Paris-Nord

³ CNRS - Centre National de la Recherche Scientifique

VMCAI 2023 16 January 2023

Loop Optimization

```
loop (0...n) {  
    task_x  
}  
loop (0...n) {  
    task_y  
}
```

Fission or
distribution



```
loop (0...n) {  
    task_x  
    task_y  
}
```

Fusion or
combination



```
loop (0...n/2) {  
    task_x  
    task_y  
}  
loop (n/2...n) {  
    task_x  
    task_y  
}
```

Splitting

...and many more strategies.

We present a loop optimization algorithm based on **loop fission** transformation, to introduce **parallelization potential** in previously uncovered cases.

Potential for parallelism

- Identify *independent* operations
- Perform those operations in any order as system resources become available

Loop fission (or distribution)

- Break loop into multiple loops
- Each loop has the same iteration range
- Each takes part of original loop's body
- Some duplication may be needed

Conceptually: Distribute loops \Rightarrow parallelize \Rightarrow speedup in execution time

- Is applicable even when iteration space is unknown.
- Can be applied to any kind of loop: `for`, `while`, ...
- Can be applied to languages from high-level to intermediate representation.
- Is suitable for integration with automatic compilation and optimization tools.

Start with a sequential imperative program.

1. Perform dependency analysis using data flow graphs (DFGs).
2. Build a dependency graph.
3. Compute condensation graph and compute its covering.
4. Create loop for each statement in covering.
5. Parallelize distributed loops.

We consider simple deterministic imperative `while` language, with variables, expressions, commands, and parallel command. Program can include:

- Arrays and pure function calls,
- Arbitrarily complex update/termination conditions,
- Loop carried-dependencies, and
- Arbitrarily deep loop nests.

Certain memory accesses are out of scope: pointers, aliasing, etc.

Variables in Command C

We identify variables modified by (Out), used by (In), and occurring (Occ) in C.

E.g., $C ::= t[e_1] = e_2$,

$$\text{Out}(C) = t$$

$$\text{In}(C) = \text{Occ}(e_1) \cup \text{Occ}(e_2)$$

$$\text{Occ}(C) = t \cup \text{Occ}(e_1) \cup \text{Occ}(e_2)$$

We represent and analyze these dependencies using Data Flow Graphs (DFGs).

Data Flow Graph (DFG)

- A DFG is a matrix over a fixed semi-ring.
- Represents a weighted relation on set of variables involved in command C.
- 3 types of dependencies:

∞	dependence	$x \xrightarrow{\text{dependence}} x$
1	propagation	$y \xrightarrow{\text{propagation}} y$
0	reinitialization	$z \quad \quad \quad z$

For each command, we define a mapping from variables of command C to DFG. We write $\mathbb{M}(C)$ for the DFG of C .

Definition: Assignment

Given an assignment C , its DFG is given by:

$$\mathbb{M}(C)(y, x) = \begin{cases} \infty & \text{if } x \in \text{Out}(C) \text{ and } y \in \text{In}(C) \quad (\text{Dependence}) \\ 1 & \text{if } x = y \text{ and } x \notin \text{Out}(C) \quad (\text{Propagation}) \\ 0 & \text{otherwise} \quad (\text{Reinitialization}) \end{cases}$$

Representing DFGs

$$C ::= t[e_1] = e_2$$

$$\text{Out}(C) = \{t\}$$

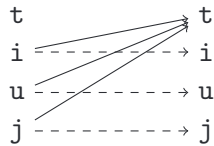
$$\text{In}(C) = \{i, u, j\}$$

$$\text{Occ}(C) = \{t, i, u, j\}$$

$$M(C)$$

$$\begin{array}{c} t \quad i \quad u \quad j \\ \begin{array}{c} t \\ i \\ u \\ j \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 \\ \infty & 1 & 0 & 0 \\ \infty & 0 & 1 & 0 \\ \infty & 0 & 0 & 1 \end{bmatrix} \end{array}$$

$$M(C) \text{ as a graph}$$



All body variables of conditional and loop statements depend on its control expression. We apply loop correction to account for this dependency.

For e an expression and C a command, $\text{Corr}(e)_C$, is $E^t \times O$.

- E^t – column vector with ∞ for variables in $\text{Occ}(e)$ and 0 for other variables.
- O – row vector with ∞ for variables in $\text{Out}(C)$ and 0 for other variables.

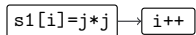
1. Pick a loop at top level.
2. Construct a **dependence graph**, which uses the DFG.
3. Compute its **condensation graph** from dependence graph.
4. Compute a **covering** of the condensation graph.
5. Create a loop per element of the covering.

Dependence Graph

```
while (t[i] > j) {  
W := s1[i]=j*j; // C1  
    s2[i]=1/j;  
    i++; // C3  
}
```

$$\mathbb{M}(W) = \begin{array}{c} \begin{array}{ccccc} & t & i & j & s1 & s2 \\ \begin{array}{l} t \\ i \\ j \\ s1 \\ s2 \end{array} & \begin{bmatrix} 1 & \infty & 0 & \infty & \infty \\ 0 & \infty & 0 & \infty & \infty \\ 0 & \infty & 1 & \infty & \infty \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

$\text{In}(C_1) = \{i, j\}$
 $\text{Out}(C_3) = \{i\}$



Definition: Dependence graph

The *dependence graph* of the loop $W := \text{while } e \text{ do } \{C_1; \dots; C_n\}$ is the graph whose vertices is the set of commands $\{C_1; \dots; C_n\}$, and there exists a directed edge from C_i to C_j if and only if there exists variables $x \in \text{Out}(C_j)$ and $y \in \text{In}(C_i)$ such that $\mathbb{M}(W)(y, x) = \infty$.

Condensation Graph & Covering

Given a dependence graph, its *condensation graph* G_W is the graph whose

- vertices are strongly connected components (SCCs) and
- edges are the edges whose source and target belong to distinct SCCs.

We then find the *proper saturated covering* of G_W . For graph G ,

- *covering* is a collection of subgraphs such that $G = \cup_{i=1}^j G_i$.
- *saturated covering* is a covering such that for all edges with source in G_i , its target belongs to G_i as well.
- It is *proper* if none of the subgraph is a subgraph of another.

Lastly, we construct loop \tilde{W} by inserting a loop for each element in the proper saturated covering.

If \tilde{W} contains multiple loops, parallelize \tilde{W} .

Example

Step 1 of 6

Identify In and Out variables

```
while (j < m) {  
    x = r[i] * A[i][j];    // C1  
    y = A[i][j] * p[j];    // C2  
    s[j] = s[j] + x;        // C3  
    q[i] = q[i] + y;        // C4  
    j++;                    // C5  
}
```

$$\begin{aligned}\text{Out}(C_1) &= \{x\} \\ \text{In}(C_1) &= \{A, i, j, r\} \\ &\vdots \\ \text{Out}(C_3) &= \{s\} \\ \text{In}(C_3) &= \{s, j, x\} \\ &\vdots \\ \text{Out}(C_5) &= \{j\} \\ \text{In}(C_5) &= \{j\}\end{aligned}$$

Example

Step 2 of 6

Construct DFGs for each command

```
while (j < m) {  
  x = r[i] * A[i][j];    // C1  
  y = A[i][j] * p[j];    // C2  
  s[j] = s[j] + x;       // C3  
  q[i] = q[i] + y;       // C4  
  j++;                   // C5  
}
```

$$M(C_1) = \begin{matrix} & \begin{matrix} i & j & m & x & y & A & r & s & p & q \end{matrix} \\ \begin{matrix} i \\ j \\ m \\ x \\ y \\ A \\ r \\ s \\ p \\ q \end{matrix} & \begin{bmatrix} 1 & \cdot & \cdot & \infty & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \infty & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \infty & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \infty & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix} \end{matrix}$$

Example

Step 3 of 6

Compose DFGs of commands $\mathbb{M}(\mathbf{C}_1; \dots; \mathbf{C}_n)$ and apply loop correction $E^t \times O$

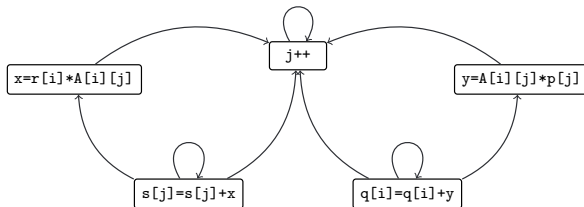
$$\mathbb{M}(\mathbf{C}) = \begin{matrix} & \begin{matrix} i & j & m & x & y & A & r & s & p & q \end{matrix} \\ \begin{matrix} i \\ j \\ m \\ x \\ y \\ A \\ r \\ s \\ p \\ q \end{matrix} & \begin{bmatrix} 1 & \cdot & \cdot & \infty & \infty & \cdot & \cdot & \cdot & \cdot & \infty \\ \cdot & \infty & \cdot & \infty & \infty & \cdot & \cdot & \infty & \cdot & \infty \\ \cdot & \infty & 1 & \infty & \infty & \cdot & \cdot & \infty & \cdot & \infty \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \infty & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \infty \\ \cdot & \cdot & \cdot & \infty & \infty & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \infty & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \infty & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \infty & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \infty \end{bmatrix} \end{matrix}$$

$$\mathbb{M}(\mathbf{C}) = \mathbb{M}(\mathbf{C}_5) \times \dots \times \mathbb{M}(\mathbf{C}_1) + \text{Corr}(\mathbf{e})_{\mathbf{C}}$$

Example

Step 4 of 6

Construct a dependence graph. Vertices are the set of commands $\{C_1; \dots; C_n\}$. Add directed edge from C_i to C_j iff $\exists x, y$, where $x \in \text{Out}(C_j)$ and $y \in \text{In}(C_i)$ and $\mathbb{M}(W)(y, x) = \infty$.



Example

Step 5 of 6

Construct a condensation graph and proper saturated covering.



Example

Step 6 of 6

Distribute loops and parallelize.

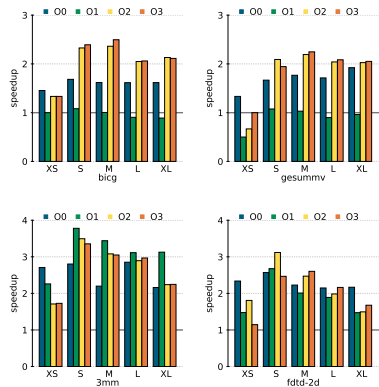
$$\tilde{W} := \text{parallel} \left\{ \begin{array}{l} \text{while } (j < m) \{ \\ \quad x = r[i] * A[i][j]; \\ \quad s[j] = s[j] + x; \\ \quad j++; \\ \} \end{array} \right\} \left\{ \begin{array}{l} \text{while } (j < m) \{ \\ \quad y = A[i][j] * p[j]; \\ \quad q[i] = q[i] + y; \\ \quad j++; \\ \} \end{array} \right\}$$

- Our artifact¹ is a collection of benchmarks.
- Mapped imperative syntax to C language.
- Used OpenMP directives to parallelize.
- Measured on standard benchmark suites, partially converted to `while` loops.
- Compared to an alternative loop transformation tool.



¹Clément Aubert et al. *Distributing and Parallelizing Non-canonical Loops – Artifact*. Version 1.0. Sept. 2022. DOI: 10.5281/zenodo.7080145. URL: <https://github.com/statycc/loop-fission>.


Experimental Results



- Enables transformation and parallelization of loops ignored by alternative methods.
- Non-canonical loops: speedup upper-bounded by the number of parallelizable loops produced by transformation.
- Canonical loops: comparable to alternative methods in speedup potential.
- Demonstrated automatic insertion of parallel directives and practicality of this technique.

Conclusion

- Introduced an automatable loop optimization technique that adds parallelization potential to imperative programs.
- It is loop and language-agnostic – many possible applications.
- We presented the algorithm to perform the loop optimization.
- Experimental results demonstrate expected performance gain – see artifact
- See our paper for proof of preservation of semantic correctness.

 `statycc/loop-fission`