Distributing and Parallelizing Non-canonical Loops

Clément Aubert[1], Thomas Rubiano[2],
Neea Rusch[1], 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.

In This Work

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

Our Technique

- 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.

Technique Overview

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.

Program Under Analysis

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 $\mathtt{C}$

We identify variables modified by (Out), used by (In), and occurring (Occ) in $\mathtt{C}$.

E.g., $\mathtt{C} ::= \mathtt{t[e_1]} = \mathtt{e_2}$,

$$\begin{aligned}
\mathrm{Out}(\mathtt{C}) &= \mathtt{t} \\
\mathrm{In}(\mathtt{C}) &= \mathrm{Occ}(\mathtt{e}_1) \cup \mathrm{Occ}(\mathtt{e}_2) \\
\mathrm{Occ}(\mathtt{C}) &= \mathtt{t} \cup \mathrm{Occ}(\mathtt{e}_1) \cup \mathrm{Occ}(\mathtt{e}_2)
\end{aligned}$$

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	$\mathtt{x} \xrightarrow{\text{dependence}} \mathtt{x}$
1	propagation	$\mathtt{y} \overset{\text{propagation}}{\dashrightarrow} \mathtt{y}$
0	reinitialization	$\mathtt{z} \qquad\qquad \mathtt{z}$

Constructing DFGs

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

Definition: Assignment

Given an assignment $\mathtt{C}$, its DFG is given by:

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

Representing DFGs

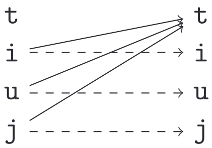
$$\mathtt{C} ::= \mathtt{t[e_1]} = \mathtt{e_2}$$

$$\mathrm{Out}(\mathtt{C}) = \{\mathtt{t}\}$$
$$\mathrm{In}(\mathtt{C}) = \{\mathtt{i}, \mathtt{u}, \mathtt{j}\}$$
$$\mathrm{Occ}(\mathtt{C}) = \{\mathtt{t}, \mathtt{i}, \mathtt{u}, \mathtt{j}\}$$

$\mathbb{M}(\mathtt{C})$

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

$\mathbb{M}(\mathtt{C})$ as a graph



Correction

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

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

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

Algorithm

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

$\mathtt{W} :=$

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

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

$\mathrm{In}(\mathtt{C}_1) = \{\mathtt{i}, \mathtt{j}\}$

$\mathrm{Out}(\mathtt{C}_3) = \{\mathtt{i}\}$

$$C_1 \longrightarrow C_3$$

Definition: Dependence graph

The *dependence graph* of the loop $\mathtt{W} := \mathtt{while\ e\ do}\ \{\mathtt{C}_1; \cdots; \mathtt{C}_\mathtt{n}\}$ is the graph whose vertices is the set of commands $\{\mathtt{C}_1; \cdots; \mathtt{C}_\mathtt{n}\}$, and there exists a directed edge from $\mathtt{C}_i$ to $\mathtt{C}_j$ if and only if there exists variables $\mathtt{x} \in \mathrm{Out}(\mathtt{C}_j)$ and $\mathtt{y} \in \mathrm{In}(\mathtt{C}_i)$ such that $\mathbb{M}(\mathtt{W})(\mathtt{y}, \mathtt{x}) = \infty$.

Condensation Graph & Covering

Given a dependence graph, its *condensation graph* $\mathtt{G_W}$ is the graph whose

- vertices are strongly connected components ($\textsc{scc}$s) and
- edges are the edges whose source and target belong to distinct $\textsc{scc}$s.

We then find the *proper saturated covering* of $\mathtt{G_W}$. 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.

Constructing Output

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

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

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
}
```

$$\begin{aligned}
\mathrm{Out}(\mathtt{C}_1) &= \{\mathtt{x}\} \\
\mathrm{In}(\mathtt{C}_1) &= \{\mathtt{A}, \mathtt{i}, \mathtt{j}, \mathtt{r}\} \\
&\vdots \\
\mathrm{Out}(\mathtt{C}_3) &= \{\mathtt{s}\} \\
\mathrm{In}(\mathtt{C}_3) &= \{\mathtt{s}, \mathtt{j}, \mathtt{x}\} \\
&\vdots \\
\mathrm{Out}(\mathtt{C}_5) &= \{\mathtt{j}\} \\
\mathrm{In}(\mathtt{C}_5) &= \{\mathtt{j}\}
\end{aligned}$$

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
}
```

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

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

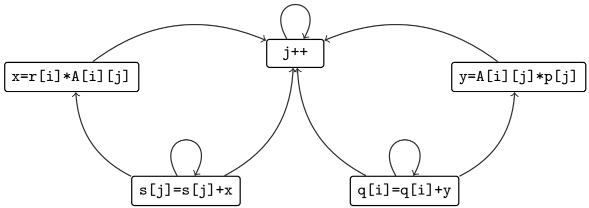
$$\mathbb{M}(\mathtt{C}) = \begin{array}{cc} & \begin{array}{cccccccccc} \mathtt{i} & \mathtt{j} & \mathtt{m} & \mathtt{x} & \mathtt{y} & \mathtt{A} & \mathtt{r} & \mathtt{s} & \mathtt{p} & \mathtt{q} \end{array} \\ \begin{array}{c} \mathtt{i} \\ \mathtt{j} \\ \mathtt{m} \\ \mathtt{x} \\ \mathtt{y} \\ \mathtt{A} \\ \mathtt{r} \\ \mathtt{s} \\ \mathtt{p} \\ \mathtt{q} \end{array} & \begin{bmatrix} 1 & \cdot & \cdot & \infty & \infty & \cdot & \cdot & \cdot & \cdot & \infty \\ \cdot & \infty & \cdot & \infty & \infty & \cdot & \cdot & \infty & \cdot & \infty \\ \cdot & \infty & 1 & \infty & \infty & \cdot & \cdot & \infty & \cdot & \infty \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \infty & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \infty \\ \cdot & \cdot & \cdot & \infty & \infty & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \infty & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \infty & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \infty & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \infty \end{bmatrix} \end{array}$$

$\mathbb{M}(\mathtt{C}) = \mathbb{M}(\mathtt{C}_5) \times \cdots \times \mathbb{M}(\mathtt{C}_1) + \mathrm{Corr}(\mathtt{e})_\mathtt{C}$

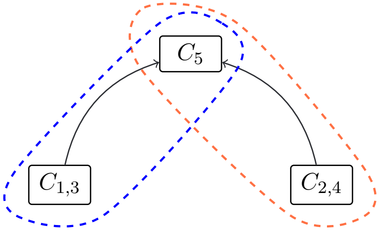
Example

Step 4 of 6

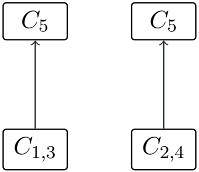
Construct a dependence graph. Vertices are the set of commands $\{\mathtt{C}_1; \cdots; \mathtt{C}_\mathtt{n}\}$. Add directed edge from $\mathtt{C}_i$ to $\mathtt{C}_j$ iff $\exists\, x, y$, where $\mathtt{x} \in \mathrm{Out}(\mathtt{C}_j)$ and $\mathtt{y} \in \mathrm{In}(\mathtt{C}_i)$ and $\mathbb{M}(\mathtt{W})(\mathtt{y}, \mathtt{x}) = \infty$.



Construct a condensation graph and proper saturated covering.



$\Rightarrow$



Example

Step 6 of 6

Distribute loops and parallelize.

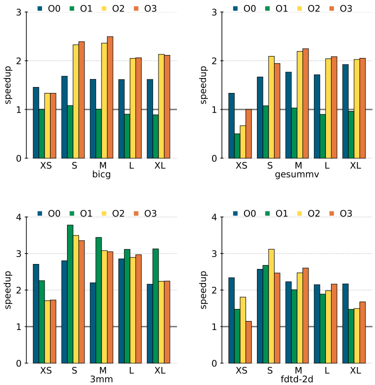
$$\tilde{\mathtt{W}} \coloneqq \quad \mathtt{parallel} \left\{ \begin{array}{l} \texttt{while (j < m) \{} \\ \quad \texttt{x = r[i] * A[i][j];} \\ \quad \texttt{s[j] = s[j] + x;} \\ \quad \texttt{j++;} \\ \texttt{\}} \end{array} \right\} \left\{ \begin{array}{l} \texttt{while (j < m) \{} \\ \quad \texttt{y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

Experimental Evaluation

- Our artifact[1] 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.

[1] Clément 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.