A Novel Loop Fission Technique Inspired by Implicit Computational Complexity

Clément Aubert, Thomas Rubiano, Neea Rusch, Thomas Seiller

▶ **To cite this version:**

Clément Aubert, Thomas Rubiano, Neea Rusch, Thomas Seiller. A Novel Loop Fission Technique Inspired by Implicit Computational Complexity. 2022. hal-03669387v1

HAL Id: hal-03669387

https://hal.science/hal-03669387v1

Preprint submitted on 16 May 2022 (v1), last revised 19 Sep 2022 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Novel Loop Fission Technique Inspired by Implicit Computational Complexity*

Clément Aubert[1] [0000−0001−6346−3043], Thomas Rubiano[2], Neea Rusch[1] [0000−0002−7354−5330], and Thomas Seiller[2,3] [0000−0001−6313−0898]

[1] School of Computer and Cyber Sciences, Augusta University
[2] LIPN – UMR 7030 Université Sorbonne Paris Nord
[3] CNRS

Abstract. This work explores an unexpected application of Implicit Computational Complexity (ICC) to parallelize loops in imperative programs. Thanks to a lightweight dependency analysis, our algorithm allows splitting a loop into multiple loops that can be run in parallel, resulting in gains in terms of execution time similar to state-of-the-art automatic parallelization tools when both are applicable. Our graph-based algorithm is intuitive, language-agnostic, proven correct, and applicable to all types of loops, even if their loop iteration space is unknown statically or at compile time, if they are not in canonical form or if they contain loop-carried dependency. As contributions we deliver the computational technique, proof of its preservation of semantic correctness, and experimental results to quantify the expected performance gains. Our benchmarks also show that the technique could be seamlessly integrated into compiler passes or other automatic parallelization suites. We assert that this original and automatable loop transformation method was discovered thanks to the "orthogonal" approach offered by ICC.

Keywords: Analysis and Verification of Parallel Program · Automatic Parallelization · Loop Transformation · Implicit Computational Complexity

1 Original Approaches to Automatic Parallelization

1.1 Use Cases for Correct Automatic Parallelization

The demand for perpetually more performant systems has historically driven innovation in hardware, by increasing numbers of transistors and improving clock speeds, but with Dennard scaling and the end of Moore's law in sight, the focus is steadily shifting toward obtaining gains through high-performance and parallel computing [35, Chapter 1]. Existing parallel programming APIs, such as OpenMP [21], PPL [27], and oneTBB [18], facilitate this progression; but several

* This research is supported by the Thomas Jefferson Fund of the Embassy of France in the United States and the FACE Foundation. Th. Rubiano and Th. Seiller are also supported by the Île-de-France region through the DIM RFSI project "CoHOp".

outstanding issues remain: classic algorithms are written sequentially without parallelization in mind and require reformatting to fit the parallel paradigm. Suitable sequential programs with opportunity for parallelization must be modified, often manually, by inserting parallelization directives. This is a time-consuming and error-prone process: numerous dependencies and multiple levels of function calls must be analyzed to identify regions that can be safely parallelized, before inserting the correct directives—a challenging task to programmers accustomed to sequential execution environment. Lastly, the state explosion resulting from parallelization makes it impossible to exhaustively test the code running on parallel architectures [8].

The need for parallelization extends beyond software presently developed: to leverage the potential speedup available on modern hardware, all programs—including legacy software—should instruct the hardware to take advantage of its available processors. This induces demand for automatic transformations of large bodies of software to semantically equivalent parallel programs. Since inserting *incorrect* parallel directives is easy, and can lead to performance degradation or alteration in the program's behavior, *correct* automatic parallelization can be used in lieu of human ingenuity, and benchmarked to approximate the potential gain. A useful automatic parallelization tool needs to deliver two things: a cost-benefit analysis, to show it generates speedup on parallel architectures, and proof of preservation of semantic correctness [13, Section 3.5].

Compilers offer an ideal integration point for many program analyses and optimizations. Automatic parallelization is already a standard feature in developing industry compilers, optimizing compilers, and specialty source-to-source compilers. Tools that perform local transformations, generally on loops, are frequently conceived—but not necessarily implemented—as compiler passes. How those passes are intertwined with sequential code optimizations can sometimes be problematic [12]: as an example, OpenMP directives are by default applied early in the compilation, or even at the front-end, and hence the parallelized source code cannot benefit from sequential optimizations such as unrolling. Furthermore, compilers tend to make conservative choices and often miss opportunities to parallelize, e.g., on complex scientific and engineering codes [12,17].

Given this background, the need to parallelize source code automatically and at scale is evident, but the existing approaches are not perfect. The contribution presented in this paper offers an incremental improvement in this direction: it introduces an automatable and graph-based computational method for semantic-preserving loop transformation. By producing a transformed program, shown to be amiable to integration or pipelining with existing automatic parallelization tools, it fits the described landscape by offering potential to improve versatility and richness of various existing parallel compilation toolchains.

1.2 Leveraging ICC for Correct and Universal Transformation

The technique presented in this paper is founded on Implicit Computational Complexity (ICC) theory [15]: this work is part of a series [6] that explores how ICC can provide new—sometimes orthogonal—approaches to problems such as

code optimization [28,29] or static analysis [5]. Critical to this approach is a strong mathematical backbone that allows to "embed" in the program itself a guarantee of its resource usage, using e.g., bounded recursion [10,26] or type systems [9,23]. This orthogonal approach sometimes allows avoiding difficulties other techniques must address and offers gain in terms of e.g., speed, but possibly at the price of precision [5, Sect. C].

More precisely, the presented technique demonstrates how a dependency analysis mechanism, first introduced in our previous work [28], can be further leveraged to obtain *loop-level parallelism*: a form of parallelism concerned with extracting parallel tasks from loops. We identify an original way of performing *loop fission*, an optimization technique that breaks loops into multiple loops with the same condition or index range, each taking only a part of the original loop's body. Our original technique possesses three notable properties:

Suitable to loops with unknown iteration spaces —program analysis does not require knowing loop iteration space statically nor at compile time, making it applicable to loops not in canonical form, which are often ignored (Sect. A).

Loop-agnostic —the technique requires practically no structure from the loops: in particular, they can be `while`, `do ... while` or `for` loops, have arbitrarily complex update and termination conditions, loop-carried dependencies, and arbitrarily deep loop nests.

Language-agnostic —the method can be used on any imperative language, making it flexible and suitable for realization and integration with tools and languages ranging from high-level to intermediate representations.

The intent of our work is not to replace polyhedral models [20]—that are also pushing to remove some restrictions [11]—, advanced dependency analysis or tools developed for very precise cases (such as loop tiling [12]), nor to build concrete competing implementations. Our goal is to illustrate how ICC has potential to introduce novel and orthogonal optimization techniques: we take as a positive sign the fact that we can facilitate the discovery of equivalent parallel implementations that are not reachable through a pre-established set of correct transformation rules, as complementary to existing methods. This also benefit our approach, making scheduling or optimization of caching out of the scope of this work, since they can be deferred to the tool implementing our algorithm.

1.3 Our Contribution: From Theory to Benchmarks

Our contribution spans from theoretical foundations to concrete measurements, to deliver a complete perspective on the design and expected real-time efficiency of the introduced method. We present three contributions:

1. The design of a loop fission transformation—Sect. 3.1—that analyzes dependencies of loop condition and body variables; establishes cliques between statements, and splits independent cliques into multiple loops.

2. The correctness proof—Sect. 3.2—that guarantees the semantic preservation of loop transformation.
3. Experimental results—Sect. 4—that evaluate the potential gain of the proposed technique, including for loops with unknown iteration spaces, and demonstrates its integratability with existing parallelization frameworks.

This paper is organized as follows: Sect. 2 defines the "building blocks" for our program transformation method—its language and how dependencies are computed—, Sect. 3 details the loop fission algorithm, Sect. 4 presents the experimental results, and Sect. 5 concludes.

2 Background: Language and Dependency Analysis

2.1 A Simple While Imperative Language With Parallel Capacities

We work with a simple imperative `WHILE`-language, with semantics similar to `C`, extended with a `parallel` command, similar to e.g., OpenMP's directives [21], allowing to execute its arguments in parallel. The grammar is given by:

$$\begin{aligned} \mathit{var} &::= \mathtt{i} \mid \mathtt{j} \mid \ldots \mid \mathtt{s} \mid \mathtt{t} \mid \ldots \mid \mathtt{x}_1 \mid \mathtt{x}_2 \mid \ldots \mid \mathtt{z_n} \mid \mathit{var}[\mathit{exp}] && \text{(Variables)} \\ \mathit{exp} &::= \mathit{var} \mid \mathit{val} \mid \mathit{op}(\mathit{exp}, \ldots, \mathit{exp}) && \text{(Expression)} \\ \mathit{com} &::= \mathit{var} = \mathit{exp} \mid \mathtt{if}\ \mathit{exp}\ \mathtt{then}\ \mathit{com}\ \mathtt{else}\ \mathit{com} \mid \\ &\quad \mathtt{while}\ \mathit{exp}\ \mathtt{do}\ \mathit{com} \mid \mathtt{use}(\mathit{var}, \ldots, \mathit{var}) \mid \mathtt{skip} \mid \\ &\quad \mathit{com};\mathit{com} \mid \mathtt{parallel}\{\mathit{com}\}\{\mathit{com}\} \cdots \{\mathit{com}\} && \text{(Command)} \end{aligned}$$

A variable represents either an undetermined "primitive" datatype, e.g., not a reference variable, or an array, whose indices are given by an expression. An expression is either a variable, a value (e.g., integer literal) or the application to expressions of some operator *op*, which can be e.g., relational (e.g., `==`, `<`) or arithmetic (e.g., `+`, `-`). We let `V` (resp. `e`, `C`) ranges over variables (resp. expression, command), write e.g., `if e then C` for `if e then C else skip`, and sometimes replace the semicolon with a new line. We assume commands to be correct, e.g., with operators correctly applied to expressions, no out-of-bounds errors, etc.

A `WHILE` program is thus a sequence of statements, each statement being either an *assignment*, a *conditional*, a *while* loop, a *function call*[4]or a *skip*. *Statements* are abstracted into *commands*, which can be a statement, a sequence of commands, or multiple commands to be run in parallel. The semantics of `parallel` is the following: variables appearing in the arguments are considered local, and the value of a given variable `x` after execution of the `parallel` command is the value of the last modified local variable `x`. This implies possible race conditions, but our transformation will be robust to those: we will assume given `parallel`-free

[4] The `use` command represents any command which does not modify its variables but use them and should not be moved around carelessly (e.g., a `printf`). In practice, we currently treat all function calls as `use`, even if the function is pure.

programs, and will introduce `parallel` commands that either uniformly update the variables accross commands, or update them in only one command.

For convenience we define the following sets of variables.

Definition 1. *Let* C *be a command, we let* Out(C) *(resp.* In(C), Occ(C)*) be the set of variables* modified *by (resp.* used *by,* occuring *in)* C*, as follows:*

C	Out(C)	In(C)	Occ(C)
$\mathtt{x} = \mathtt{e}$	x	$\mathrm{Occ}(\mathtt{e})$	$\mathtt{x} \cup \mathrm{Occ}(\mathtt{e})$
$\mathtt{t[e_1]} = \mathtt{e_2}$	$\mathtt{t}$	$\mathrm{Occ}(\mathtt{e_1}) \cup \mathrm{Occ}(\mathtt{e_2})$	$\mathtt{t} \cup \mathrm{Occ}(\mathtt{e_1}) \cup \mathrm{Occ}(\mathtt{e_2})$
`if` e `then` $\mathtt{C_1}$ `else` $\mathtt{C_2}$	$\mathrm{Out}(\mathtt{C_1}) \cup \mathrm{Out}(\mathtt{C_2})$	$\mathrm{Occ}(\mathtt{e}) \cup \mathrm{In}(\mathtt{C_1}) \cup \mathrm{In}(\mathtt{C_2})$	$\mathrm{Occ}(\mathtt{e}) \cup \mathrm{Occ}(\mathtt{C_1}) \cup \mathrm{Occ}(\mathtt{C_2})$
`while` e `do` C	$\mathrm{Out}(\mathtt{C})$	$\mathrm{Occ}(\mathtt{e}) \cup \mathrm{In}(\mathtt{C})$	$\mathrm{Occ}(\mathtt{e}) \cup \mathrm{Occ}(\mathtt{C})$
`use`$(\mathtt{x}_1, \dots, \mathtt{x}_n)$	$\emptyset$	$\{\mathtt{x}_1, \dots, \mathtt{x}_n\}$	$\{\mathtt{x}_1, \dots, \mathtt{x}_n\}$
`skip`	$\emptyset$	$\emptyset$	$\emptyset$
$\mathtt{C_1};\mathtt{C_2}$	$\mathrm{Out}(\mathtt{C_1}) \cup \mathrm{Out}(\mathtt{C_2})$	$\mathrm{In}(\mathtt{C_1}) \cup \mathrm{In}(\mathtt{C_2})$	$\mathrm{Occ}(\mathtt{C_1}) \cup \mathrm{Occ}(\mathtt{C_2})$

for $\mathrm{Occ}(\mathtt{x}) = \mathtt{x}$ $\quad \mathrm{Occ}(\mathtt{t[e]}) = \mathtt{t} \cup \mathrm{Occ}(\mathtt{e})$

$\mathrm{Occ}(val) = \emptyset \quad \mathrm{Occ}(\mathtt{op(e_1, \dots, e_n)}) = \mathrm{Occ}(\mathtt{e_1}) \cup \dots \cup \mathrm{Occ}(\mathtt{e_n})$

Our treatment of arrays is an over-approximation: we consider the array as a single entity, and that changing one value in it changes it completely. This is however satisfactory: since we will not split loop "horizontally" (e.g., splitting the iteration space between threads) but "vertically" (e.g., splitting the tasks between threads), we want each thread in the `parallel` command to have "full control" of the array it modifies, and not to synchronize its writes with other commands.

2.2 Datalow Graphs for Loop Dependency Analysis

The loop transformation algorithm relies fundamentally on its ability to analyze data-flow dependencies between loop condition and variables in the loop body, to identify opportunities for loop fission. In this section we sketch the principles of this dependency analysis, founded on the theory of *data-flow graphs*, and how it maps to the presented `WHILE`-language. This dependency analysis was influenced by large body of works related to static analysis [1,22,25], semantics [24,34] and optimization [28]; but is presented here in self-contained and compact manner.

Definition of Data-Flow Graphs A data-flow graph for a given command C is a weighted relation on the set Occ(C). Formally, this is represented as a matrix over a semi-ring, with the implicit choice of a denumeration of Occ(C).

Definition 2. *A* data-flow graph *(*DFG*) for a command* C *is a* $|\mathrm{Occ}(\mathtt{C})| \times |\mathrm{Occ}(\mathtt{C})|$ *matrix over a fixed semi-ring* $(\mathcal{S}, +, \times)$*, with* $|\mathrm{Occ}(\mathtt{C})|$ *the cardinal of* Occ(C)*. We write* $\mathbb{M}(\mathtt{C})$ *the* DFG *of* C*, and explain how to construct it below.*

To avoid resizing matrices whenever additional variables are considered, we identify $\mathbb{M}(\mathtt{C})$ with its embedding in a larger matrix, i.e., we will abusively call the DFG of C any matrix of the form $\mathbb{M}(\mathtt{C}) \oplus \mathrm{Id}$, implicitly viewing the additional rows/columns as variables not in Occ(C). We will use weighted relations, or

weighted bi-partite graphs, to illustrate these matrices. Examples will use the semiring $(\{0, 1, \infty\}, \max, \times)$, which is the specific semiring considered in later sections to represent dependencies: ∞ represents *dependence*, 1 represents *propagation*, and 0 represents *reinitialization* or independence. Fig. 1 introduces these notions and the graphical conventions used throughout this paper. Note that in the case of dependencies, $\mathrm{In}(\mathtt{C})$ is exactly the set of variables that are source of a "dependence" arrow, while $\mathrm{Out}(\mathtt{C})$ is the set of variables that either are targets of dependence arrows or were reinitialized.

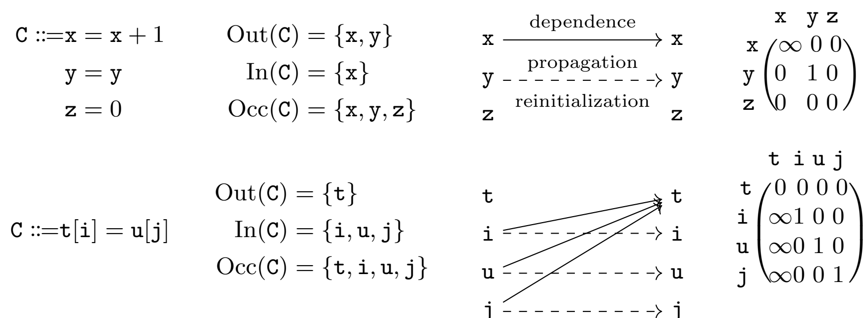


Fig. 1. Program examples, sets, and representations of their dependences

2.3 Constructing Data-Flow Graphs (DFGs)

The DFG of a command is computed by induction on the structure of the command.

Base cases (assignment, skip, use) The DFG for assignments are obtained by straightforward generalization of the cases illustrated in Fig. 1, and $\mathbb{M}(\mathtt{skip})$ is the "empty matrix" with 0 rows and columns.[5]

To account for $\mathtt{use}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$, we introduce a variable $\mathtt{e}$—standing for *effect*—not being part of the language, and let $\mathbb{M}(\mathtt{use}(\mathtt{x}_1, \ldots, \mathtt{x}_n))$ be the matrix with coefficients from each $\mathtt{x}_i$ and $\mathtt{e}$ to $\mathtt{e}$ equal to ∞, and 0 coefficients otherwise.

Composition and multipaths The definition of DFG for a (sequential) *composition* of commands is an abstraction that allows treating a block of statements as one command with its own DFG.

Definition 3. $\mathbb{M}(\mathtt{C}_1; \mathtt{C}_2; \ldots; \mathtt{C}_n)$ *is the matrix product* $\mathbb{M}(\mathtt{C}_1)\mathbb{M}(\mathtt{C}_2)\cdots\mathbb{M}(\mathtt{C}_n)$.

For two graphs, the product of their matrices of weights is represented in a standard way, as a graph of length 2 paths; as illustrated in Fig. 2.

[5] Identifying the DFG with its embeddings, it is hence the identity matrix of any size.

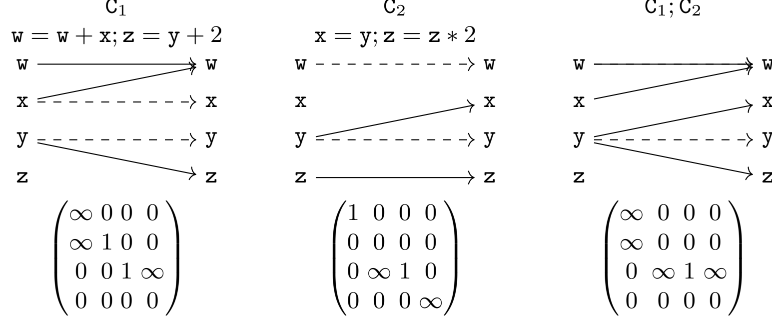


Fig. 2. Data-Flow Graph of Composition.

Conditionals. We define the DFG of if e then $\mathtt{C}_1$ else $\mathtt{C}_2$ from the DFG of the commands $\mathtt{C}_1$ and $\mathtt{C}_2$. First, consider a situation where both commands $\mathtt{C}_1$ and $\mathtt{C}_2$ are potentially executed. In that case, the statement should be represented by the overapproximation $\mathbb{M}(\mathtt{C}_1) + \mathbb{M}(\mathtt{C}_2)$. However, all the modified variables in $\mathtt{C}_1$ and $\mathtt{C}_2$ (e.g., $\mathrm{Out}(\mathtt{C}_1) \cup \mathrm{Out}(\mathtt{C}_2)$) depends on the variables used in e (e.g., occuring in e). For this reason, letting E be the vector with coefficient equal to ∞ for the variables in e and 0 for all the other variables, O be the vector containing the variables in $\mathrm{Out}(\mathtt{C}_1) \cup \mathrm{Out}(\mathtt{C}_2)$, and $(\cdot)^t$ be the matrix transpose, we define $\mathrm{Corr}(\mathtt{e}) = (EO)^t$, and have—as illustrated previously [28, Fig. 3]—:

Definition 4. $\mathbb{M}(\texttt{if e then } \mathtt{C}_1 \texttt{ else } \mathtt{C}_2) = \mathbb{M}(\mathtt{C}_1) + \mathbb{M}(\mathtt{C}_2) + \mathrm{Corr}(\mathtt{e})$.

While Loops. To define the DFG of a command while e do C from $\mathbb{M}(\mathtt{C})$, we need, as for conditionals, the *loop correction* $\mathrm{Corr}(\mathtt{e})$, to account for the fact that all the modified variables in C depends on the variables used in e:

Definition 5. $\mathbb{M}(\texttt{while e do C}) = \mathbb{M}(\mathtt{C}) + \mathrm{Corr}(\mathtt{e})$.

This is different from our previous treatment of while loop [28, Definition 5], that required to compute the transitive closure of $\mathbb{M}(\mathtt{C})$: for this particular transformation, this is not needed, as all the relevant dependencies are obtained immediately—this also guarantee that loop-carried dependencies [38] do not refrain from parallelizing the body of the loops, in our analysis.

3 Loop Fission Algorithm

3.1 Algorithm, Presentation and Intuition

Leveraging the presented dependency analysis, we can now define the specifics of our loop transformation technique, given in Algo. 1 and explained below.

Given a loop $\mathtt{C} := \texttt{while e do } \{\mathtt{C}_1; \cdots; \mathtt{C}_n\}$, we first compute $\mathbb{M}(\mathtt{C}_1; \ldots; \mathtt{C}_n)$, and add the loop correction $\mathrm{Corr}(\mathtt{e})$. The dependence graph of the loop C is

Algorithm 1 Loop fission

Input: loop $w = \{\texttt{while} \in \texttt{WHILE}\}$
vertices ← empty list
In, *Out* ← of loop body variables
parent ← parent statement of w
for all *in*, *out* ∈ c(*In*, *Out*) **do**
  if *in* is dependency of *out* **then**
    vertices += vertex(*in*, *out*)
end for
for all *cond* ∈ loop conditions **do**
  vertices += (*cond*, body stmts)
end for
digraph ← from(*vertices*)
sccs ← reduce(*digraph*)
dag ← condensation(*digraph*, *sccs*)
subgraphs ← from *dag*
if |*subgraphs*| > 1 **then**
  parent remove(w)
  for all *scc* ∈ *sccs* **do**
    parent insert loop from scc
  end for

then defined as as the graph where the set of vertices is the set of commands $\{\texttt{C}_1; \cdots ; \texttt{C}_n\}$, and there exists a directed edge from $\texttt{C}_i$ to $\texttt{C}_j$ if and only if there exists variables $\texttt{x} \in \text{Out}(\texttt{C}_j)$ and $\texttt{y} \in \text{In}(\texttt{C}_i)$ such that $\mathbb{M}(\texttt{C})(\texttt{x}, \texttt{y}) = \infty$. Note that all the commands are the sources of dependence edges whose target is the commands modifying the variables occuring in $\texttt{e}$ thanks to the correction.

The remainder of the loop transforming principle is simple: for each loop in the analyzed program, after evaluating data dependencies in the loop condition and variables in the loop body, it produces a graph representing the dependencies between commands; then determines the cliques in the graph and forms *strongly connected components* (SCCs); the SCCs are separated into subgraphs to produce the final split loops, when applicable, and contain a copy of the loop header and update commands.

Let us consider a program and its corresponding graph, in Fig. 3, where strongly connected components (SCCs) are shown as dotted rectangles: the *condensation graph* is the graph whose vertices are SCCs and edges are the edges whose source and target belong to distinct SCCs. Our algorithm splits this graph into its branches, introducing duplications, and outputs a parallelizable version of the initial program as follows. It first splits the graph into branches: for this, we work not with the dependency graph directly but with the acyclic graph obtained from its decomposition into SCCs. This graph is known as the *condensation* of the initial graph. The splitting algorithm produces automatically the corresponding program and its *covering* (Definition 6), presented in Fig. 4 and 5.

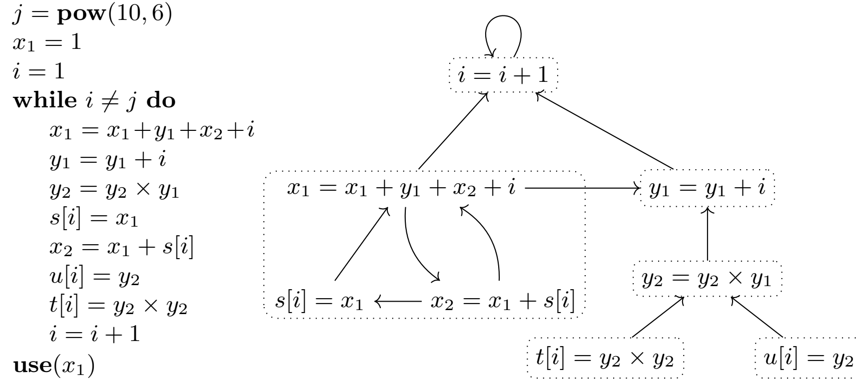
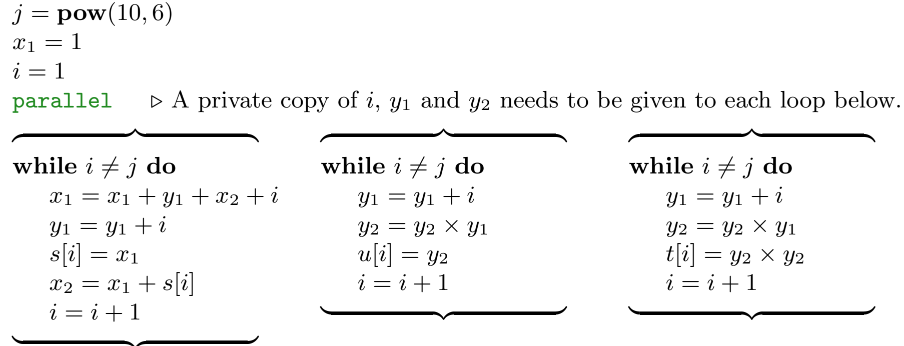


Fig. 3. `WHILE`-language program and its corresponding dependency graph.



use(x_1) ▷ The copies of i, y_1 and y_2 can be destroyed, all have the same values.

Fig. 4. Transformed program

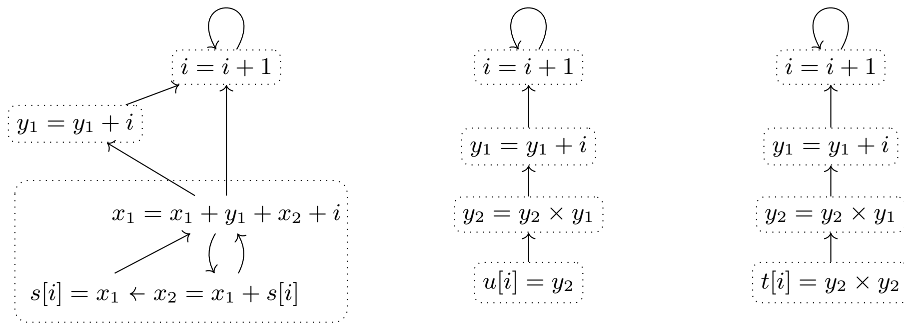


Fig. 5. Transformed program—covering

3.2 Correctness of the Algorithm

Formally, the transformation is described by means of particular kinds of *coverings* of the dependency graph. Let us start with a formal definition.

Definition 6 ([14]). *A covering of a (directed) graph G is a collection of subgraphs $G_1, G_2, \ldots, G_k$ such that $G = \cup_{i=1}^{k} G_i$.*

A saturated covering *is a covering $G_1, G_2, \ldots, G_k$ such that for all edge in G with source in G_i, its target belongs to G_i as well.*

Given a loop $\texttt{C} := \texttt{while e do } \{\texttt{C}_1; \cdots ; \texttt{C}_n\}$, we explained how one can compute its dependency graph $D(\texttt{C})$ on the set of vertices $\{1, 2, \ldots, n\}$ representing the commands in the body of C. Given a saturated covering $G_1, G_2, \ldots, G_k$ of the graph G, we can define a sequence of loops and prove that the semantics of the initial loop is preserved. This is done by showing that for any variable x appearing in the initial loop, its final value is unchanged.

Definition 7. *Let $\texttt{C} := \texttt{while e do } \{\texttt{C}_1; \cdots ; \texttt{C}_n\}$ be a command, $D(\texttt{C})$ its dependency graph, and $G_1, G_2, \ldots, G_k$ a saturated covering of the graph G. We define $\tilde{\texttt{C}} = \texttt{C}^1; \ldots ; \texttt{C}^k$ where the command $\texttt{C}^j$ is defined from G_j by $\texttt{C}^j = \texttt{while e do } \{\texttt{C}_{i_1}; \cdots ; \texttt{C}_{i_m}\}$ where $\{i_1, \ldots, i_m\}$ is the set of vertices of G_j.*

Theorem 1. *The transformation $\texttt{C} \rightsquigarrow \tilde{\texttt{C}}$ preserves the semantic.*

Proof (sketch). We show that for every variable x, the value of x after the execution of C is equal to the value of x after the execution of $\tilde{\texttt{C}}$. Variables are considered local to each loop $\texttt{C}^j$ in $\tilde{\texttt{C}}$, so we need to avoid race condition. To do so, we prove the following more precise result: for each variable x and each loop $\texttt{C}^j$ in $\tilde{\texttt{C}}$ in which the value of x is modified, the value of x after executing C is equal to the value of x after executing $\texttt{C}^j$.

The previous claim is then straightforward to prove, based on the property of the covering. One shows by induction on the number of iterations k that for all the variables $x_1, \ldots, x_h$ appearing in $\texttt{C}^j$, the values of $x_1, \ldots, x_h$ after k loop iterations of $\texttt{C}^j$ are equal to the values of $x_1, \ldots, x_h$ after k loop iterations of C. Note some other variables may be affected by the latter but the variables $x_1, \ldots, x_h$ do not depend on them (otherwise, they would also appear in $\texttt{C}^j$ by definition of the dependence graph and the covering). □

4 Experimental Results

This section aims at experimentally substantiating two claims:

1. Our algorithm can parallelize loops that are completely ignored by—to our knowledge (Sect. A)—all the other automatic parallelization tools, and result in appreciable gain (Sect. 4.2),
2. Our code transformation provides a gain similar to the automatic parallelization tool `AutoPar-Clava`—which "compare[s] favorably with closely related auto-parallelization compilers" [4, p. 1]—when both are applicable, and can be integrated in automatic parallelization pipelines (Sect. 4.3).

Taken together, those results confirm that our original loop fission can easily be integrated in pre-existing tools and improve the performances of the resulting code. We begin by briefly explaining our benchmarking strategy.

4.1 Benchmarking Strategy

The primary goal of our benchmarking strategy was to evaluate the potential performance gain using the described algorithm and parallelization. The PolyBench/C suite [30] was selected for this purpose because it contains programs in the `C` programming language, which naturally maps to the syntax of `WHILE` presented in Sect. 2, the `parallel` command being represented as OpenMP directives. The suite has been crafted to offer different opportunities for loop transformations, and comes with built-it timing utilities, which we use, to obtain accurate and comparable results. One drawback is the suite's sole focus on "canonical" `for` loops, which prevented evaluating transformations of other kinds of loops that our algorithm can split. This issue was resolved by introducing an additional case study, measuring the performance of parallelized `while` loops, discussed in Sect. 4.2.

To avoid evaluation of non-transformable programs, the suite was then reduced to the programs that were either suitable for loop fission or already had the intended form. There were 6 such programs. Next these programs were transformed manually to their post-fission form. Since the proposed technique also involves parallelization, we defined two parallelization strategies: one where OpenMP directives were inserted manually, and another using an automated parallelizing source-to-source compiler, `AutoPar-Clava` [4]. The two approaches are complementary and serve different purposes: the manual method enables finding optimal directives; the automatic approach shows these tools can be pipelined to obtain fully automatic parallelizing compilation toolchain.

The PolyBench/C benchmark timing script runs each program 5 times, taking the average of 3 runs. The script also measure variance between the averaged times, which in this study was constrained never to exceed 5%. Speedup is the ratio of sequential and parallel executions, $S = T_{\text{Seq}}/T_{\text{Par}}$, where a value greater than 1 indicates parallel is outperforming the sequential execution. In presentation of these results, the original sequential programs are always considered the baseline.

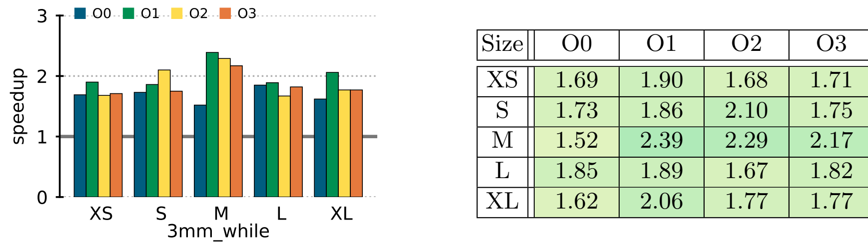
The benchmarks were ran on multiple `gcc` compiler optimization levels (`O0-O3`), and data sizes supplied with the suite (`MINI - EXTRALARGE`) using a Linux 4.19.0-20-amd64 #1 SMP Debian 4.19.235-1 (2022-03-17) x86_64 GNU/Linux machine, with 4 Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz processors. Our open source benchmarking is available at https://github.com/statycc/icc-fission.

4.2 Case Study: Parallel `while` Loops

The agnostic treatment of the various kinds of loops, including loops with unknown iteration spaces, are some of the highlights of the presented technique. It is however difficult to compare this approach to other techniques, because most loop transformation and parallelization tools focus only on `for` loops and

PolyBench/C does not include `while` loop programs. The difficulty parallelizing `while` loops arises from the need to synchronize evaluation of the loop recurrence and termination condition, with improper synchronization resulting in overshooting the iterations [33]; rendering such loops effectively sequential. Our technique addresses this challenge by recognizing the independence between loops resulting from loop fission, thus producing parallelizable loop chains. A good candidate for demonstrating this concept is the benchmark program `3mm`, from which we constructed a semantically equivalent program using `while` loops, `3mm_while`.

In general, special care is needed when inserting parallelization directives for loops with unknown iteration spaces. Use of `single` directive prevents overshooting the loop termination condition and need for synchronization between threads, enabling parallel execution by multiple threads on individual loop statements. Fig. 6 demonstrates this strategy yields a consistent speedup, with geometric mean of 1.8 across data sizes and `gcc` compiler optimization levels. While this is expectedly slightly lower than speedup of 3.1 obtained on an equivalent `for` loop program—texttt3mm is in Table 1—this is an encouraging result because `while` loops are the most demanding types of loops to optimize, and generally completely ignored. Similar examples illustrating that our analysis can apply to e.g., loop-carrying dependency loops or loops whose iteration space is not known at compilation time could be similarly crafted.



Size	O0	O1	O2	O3
XS	1.69	1.90	1.68	1.71
S	1.73	1.86	2.10	1.75
M	1.52	2.39	2.29	2.17
L	1.85	1.89	1.67	1.82
XL	1.62	2.06	1.77	1.77

Fig. 6. Speedup of `3mm` program implemented using `while` loops.

4.3 PolyBench Results

Fig. 7 visualizes the speedup obtained on manually parallelized and transformed programs. Included in the figure are only those programs to which loop fission could be applied. Parallelization directives were applied only to outer loops to reduce parallelization overhead. Loop chains without flow dependencies were placed inside a `parallel` block and parallelized using `nowait`. Shared variables were marked `private` and `reduction` clauses were used when applicable.

While the performance gains are obvious, a few cases require explanation. Speedup is not always obtained on very small data sizes (`bicg`, `gesummv`), because these are very fast programs—less than 0.01 ms in clock time—with low iterations

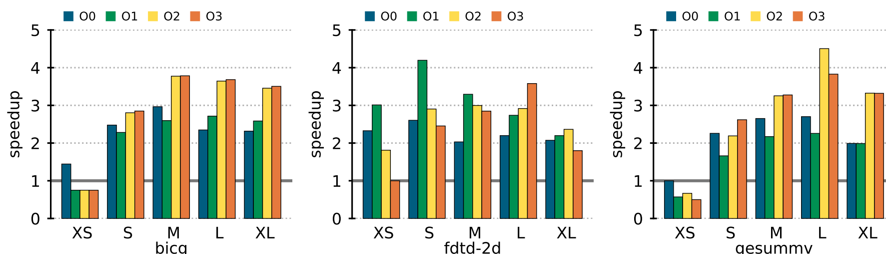


Fig. 7. Speedup obtained on Polybench/C benchmarks, after loop fission and manual parallelization, on different data sizes and compiler optimization levels.

counts, and parallel synchronization produces more overhead to not result in gain. Nonetheless, any program implemented without parallelism in mind would obtain potential speedup, equalling up to the number of available processors.

Manual vs. Automatic Parallelization Finally, we compared three parallelization strategies: 1. applying `AutoPar-Clava` to the original code, 2. applying `AutoPar-Clava` to the code as transformed by our algorithm, 3. inserting manually the OpenMP directive in the code resulting from our algorithm. Indeed, while `Autopar-Clava` does not perform loop transformations, it offers a viable alternative for placing the parallelization directives automatically. Our results, presented in Table 1 (in Sect. B) makes it evident that all 3 alternative approaches offer comparable results.[6] This illustrates that, when applicable, our technique produces results "as good as" a state-of-the-art parallelization tool, and that it can actually be leveraged in its flow.

5 Conclusion

By reasoning about programs at a high-level and in terms of graphs, we obtained an adaptable and correct code transformation, that can be used in a large variety of situations—insensible to the actual tools or programming language provided it is in an imperative style—and that offers notable gain on optimizations that are frequently overlooked. Particularly, the ability to reason about loops with unknown iteration spaces or loop-carried dependencies is significant, as this property is not supported in the described form by similar existing tools (Sect. A). Furthermore, our code transformation is lightweight, automatable, suitable to various forms of implementations, and proven correct.

From there, multiple perspectives are open: integrating our method in existing tools should not raise difficulties, but will require cost-benefit analysis along, at

[6] Except in the case of `fdtd-2d`, where the automatic tool was unable to produce safe parallelization directives, and no timing result could be obtained.

least, two axes. The first one is to determine if splitting the loop is performed in the correct type of environment: since e.g., parallelizing only the inner-most loop with OpenMP is detrimental to performances [35, Chapter 3, Nested], taking into account the tool's limitations will be crucial to result in actual gains. Second, the cost of duplication some commands can void the gain obtained from parallelizing some loops: luckily, our cost analysis [5] functions on the same imperative language, re-uses some of the tools introduced here, and could help in evaluating the degree of splitting in terms of cost-benefit analysis. Last but not least, Theorem 1 would still be valid for this "conditional" loop-splitting algorithm: further discussion and an extended example including these considerations can be found in the appendix, Sect. C.

Acknowledgments The authors wish to express their gratitude to João Bispo for explaining how to integrate `AutoPar-Clava` in their benchmark.

References

1. Abel, A., Altenkirch, T.: A predicative analysis of structural recursion. J. Funct. Program. **12**(1), 1–41 (2002). https://doi.org/10.1017/S0956796801004191
2. Amini, M.: Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators. Theses, Ecole Nationale Supérieure des Mines de Paris (Dec 2012), https://pastel.archives-ouvertes.fr/pastel-00958033
3. Amini, M., Creusillet, B., Even, S., Keryell, R., Goubier, O., Guelton, S., Mcmahon, J.O., Pasquier, F.X., Péan, G., Villalon, P.: Par4All: From Convex Array Regions to Heterogeneous Computing. In: IMPACT 2012 : Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012. Paris, France (Jan 2012), https://hal-mines-paristech.archives-ouvertes.fr/hal-00744733
4. Arabnejad, H., Bispo, J., Cardoso, J.M.P., Barbosa, J.G.: Source-to-source compilation targeting openmp-based automatic parallelization of C applications. J. Supercomput. **76**(9), 6753–6785 (Sep 2020). https://doi.org/10.1007/s11227-019-03109-9
5. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: mwp-analysis improvement and implementation: Realizing implicit computational complexity. In: Felty, A. (ed.) 7th International Conference on Formal Structures for Computation and Deduction (FSCD). LIPIcs, Schloss Dagstuhl (Mar 2022), https://hal.archives-ouvertes.fr/hal-03596285, to appear
6. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: Realizing Implicit Computational Complexity (Mar 2022), https://hal.archives-ouvertes.fr/hal-03603510, accepted to the 28th International Conference on Types for Proofs and Programs
7. Bae, H., Mustafa, D., Lee, J., Aurangzeb, Lin, H., Dave, C., Eigenmann, R., Midkiff, S.P.: The cetus source-to-source compiler infrastructure: Overview and evaluation. Int. J. Parallel Program. **41**(6), 753–767 (2013). https://doi.org/10.1007/s10766-012-0211-z
8. Baier, C., Katoen, J., Larsen, K.: Principles of Model Checking. MIT Press (2008)
9. Baillot, P., Terui, K.: Light types for polynomial time computation in lambda-calculus. In: LICS. pp. 266–275. IEEE Computer Society (2004). https://doi.org/10.1109/LICS.2004.1319621

10. Bellantoni, S.J., Cook, S.A.: A new recursion-theoretic characterization of the polytime functions (extended abstract). In: Kosaraju, S.R., Fellows, M., Wigderson, A., Ellis, J.A. (eds.) STOC. pp. 283–93. ACM (1992). https://doi.org/10.1145/129712.129740
11. Benabderrahmane, M., Pouchet, L., Cohen, A., Bastoul, C.: The polyhedral model is more widely applicable than you think. In: Gupta, R. (ed.) Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings. LNCS, vol. 6011, pp. 283–303. Springer (2010). https://doi.org/10.1007/978-3-642-11970-5_16
12. Bertolacci, I.J., Strout, M.M., de Supinski, B.R., Scogland, T.R.W., Davis, E.C., Olschanowsky, C.: Extending openmp to facilitate loop optimization. In: de Supinski, B.R., Valero-Lara, P., Martorell, X., Bellido, S.M., Labarta, J. (eds.) Evolving OpenMP for Evolving Architectures - 14th International Workshop on OpenMP, IWOMP 2018, Barcelona, Spain, September 26-28, 2018, Proceedings. LNCS, vol. 11128, pp. 53–65. Springer (2018). https://doi.org/10.1007/978-3-319-98521-3_4
13. Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J.: Parallel Programming in OpenMP. Morgan Kaufmann, Oxford, England (Oct 2000)
14. Chung, F.R.K.: On the coverings of graphs. Discret. Math. **30**(2), 89–93 (1980). https://doi.org/10.1016/0012-365X(80)90109-0
15. Dal Lago, U.: A short introduction to implicit computational complexity. In: Bezhanishvili, N., Goranko, V. (eds.) ESSLLI. LNCS, vol. 7388, pp. 89–109. Springer (2011). https://doi.org/10.1007/978-3-642-31485-8_3
16. Dave, C., Bae, H., Min, S., Lee, S., Eigenmann, R., Midkiff, S.P.: Cetus: A source-to-source compiler infrastructure for multicores. Computer **42**(11), 36–42 (2009). https://doi.org/10.1109/MC.2009.385
17. Holewinski, J., Ramamurthi, R., Ravishankar, M., Fauzia, N., Pouchet, L.N., Rountev, A., Sadayappan, P.: Dynamic trace-based analysis of vectorization potential of applications. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 371–382. PLDI '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2254064.2254108
18. Intel: oneTBB documentation (2022), https://oneapi-src.github.io/oneTBB/
19. Intel Corporation: Intel C++ Compiler Classic Developer Guide and Reference, https://www.intel.com/content/dam/develop/external/us/en/documents/cpp_compiler_classic.pdf
20. Karp, R.M., Miller, R.E., Winograd, S.: The organization of computations for uniform recurrence equations. J. ACM **14**(3), 563–590 (1967). https://doi.org/10.1145/321406.321418
21. Klemm, M., de Supinski, B.R. (eds.): OpenMP Application Programming Interface Specification Version 5.2. OpenMP Architecture Review Board (Nov 2021), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf
22. Kristiansen, L., Jones, N.D.: The flow of data and the complexity of algorithms. In: Cooper, S.B., Löwe, B., Torenvliet, L. (eds.) New Computational Paradigms, First Conference on Computability in Europe, CiE 2005, Amsterdam, The Netherlands, June 8-12, 2005, Proceedings. LNCS, vol. 3526, pp. 263–274. Springer (2005). https://doi.org/10.1007/11494645_33
23. Lafont, Y.: Soft linear logic and polynomial time. Theor. Comput. Sci. **318**(1), 163–180 (2004). https://doi.org/10.1016/j.tcs.2003.10.018

24. Laird, J., Manzonetto, G., McCusker, G., Pagani, M.: Weighted relational models of typed lambda-calculi. In: LICS. pp. 301–310. IEEE Computer Society (2013). https://doi.org/10.1109/LICS.2013.36
25. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Hankin, C., Schmidt, D. (eds.) Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001. pp. 81–92. ACM (2001). https://doi.org/10.1145/360204.360210
26. Leivant, D.: Stratified functional programs and computational complexity. In: Van Deusen, M.S., Lang, B. (eds.) Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 325–333. ACM Press (1993). https://doi.org/10.1145/158511.158659
27. microsoft: Parallel patterns library (ppl) (2021), https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl?view=msvc-170
28. Moyen, J., Rubiano, T., Seiller, T.: Loop quasi-invariant chunk detection. In: D'Souza, D., Kumar, K.N. (eds.) Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings. LNCS, vol. 10482. Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_7
29. Moyen, J., Rubiano, T., Seiller, T.: Loop quasi-invariant chunk motion by peeling with statement composition. In: Bonfante, G., Moser, G. (eds.) Proceedings 8th Workshop on Developments in Implicit Computational Complexity and 5th Workshop on Foundational and Practical Aspects of Resource Analysis, DICE-FOPARA@ETAPS 2017, Uppsala, Sweden, April 22-23, 2017. EPTCS, vol. 248, pp. 47–59 (2017). https://doi.org/10.4204/EPTCS.248.9, http://arxiv.org/abs/1704.05169
30. Pouchet, L.N., Yuki, T.: PolyBench/C 4.1, https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/
31. Prema, S., Nasre, R., Jehadeesan, R., Panigrahi, B.: A study on popular auto-parallelization frameworks. Concurrency and Computation: Practice and Experience **31**(17), e5168 (Feb 2019). https://doi.org/10.1002/cpe.5168
32. Quinlan, D., Liao, C., Panas, T., Matzke, R., Schordan, M., Vuduc, R., , Yi, Q.: Rose user manual: A tool for building source-to-source translators draft user manual (version 0.9.11.115), http://rosecompiler.org/uploads/ROSE-UserManual.pdf
33. Rauchwerger, L., Padua, D.A.: Parallelizing while loops for multiprocessor systems. In: Proceedings of the 9th International Symposium on Parallel Processing. p. 347–356. IPPS '95, IEEE Computer Society, USA (1995)
34. Seiller, T.: Interaction graphs: Full linear logic. In: Grohe, M., Koskinen, E., Shankar, N. (eds.) Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016. pp. 427–436. ACM (2016). https://doi.org/10.1145/2933575.2934568
35. Suomela, J.: Programming parallel computers (2022), https://ppc.cs.aalto.fi/
36. TylerMSFT: Parallel Patterns Library (PPL), https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl
37. Vitorović, A., Tomašević, M.V., Milutinović, V.M.: Manual parallelization versus state-of-the-art parallelization techniques. In: Hurson, A. (ed.) Advances in Computers, vol. 92, pp. 203–251. Elsevier (2014). https://doi.org/10.1016/B978-0-12-420232-0.00005-2
38. Wikipedia contributors: Loop dependence analysis — Wikipedia, the free encyclopedia (2022), https://en.wikipedia.org/w/index.php?title=Loop_dependence_analysis&oldid=1080087398, [Online; accessed 15-May-2022]

A Limitations of Existing Automatic Parallelization Tools

We focus here on presenting the types of loop that other “popular” [31] auto-parallelization frameworks for `C` *cannot* parallelize but that our algorithm could split. In particular, we do not discuss loops containing function calls that have side effects or control-flow modifiers (such as `break;` or `continue;`): neither our algorithm nor the underlying dependency mechanisms of the discussed tools—to the best of our knowledge—can accommodate those.

Most tools can process only “canonical loops”, defined e.g., in OpenMP’s specification [21, 4.4.1 Canonical Loop Nest Form]: essentially, their structure is of the form `for (init-expr; test-expr; incr-expr) structured-block`, with `incr-expr` being a (single) increment or decrement by a constant or a variable, and `test-expr` being a single comparison between a variable and a variable or a constant. Additional constraints on loop dependences are sometimes needed, e.g., the absence of loop-carried dependency [38] for cetus.

It is always hard to infer the absence of support, but we evaluated the lack of formal discussion or example of e.g., `while` loop to be sufficient to determine that the tool could not process `while` loops, unless of course they can trivially be transformed into `for` loops of the required form [37, p. 236]. We refer to a recent study [31, Section 2] for more detail on those notions and on the limitations of some of the tools discussed below.

It seems further that some tools cannot parallelize loops whose body contains e.g., `if` or `switch` statements [31, p. 18], but we have not investigated this claim further: however, our algorithm can handle `if`—and `switch` too, if it was part of our syntax—present in the body of the loop seamlessly.

Name	`for` loop	`while` loop	`do` ... `while` loop
cetus	In canonical form*	No	
Par4all	Unknown†		
ROSE	In canonical form§	No	
icc	Only if countable‡		No
PPL	Unclear¶		
Openmp	In canonical loop nest form**	No	
AutoparClava	Same limitations as openmp		

∗ *[cetus] currently handles all canonical loops of the form*
`for(i = lb; i < ub; i + = inc)` ([16, p. 39])

> *[a] loop is marked as parallel if no scalar variable carries dependences and all dependence arcs in the graph show non-loop-carried dependences with respect to the loop.* ([7, p. 761])

† Unfortunately, the project’s documentation is currently not accessible and the publications related to this project [2,3] do not discuss loop limitations.

§ Even if the manual boldly claims

> *The implementation can successfully optimize arbitrary loop structures, including complex, non-perfect loop nest* ([32, p. 123])

it later on specifies:

> *[Rose] utilizes traditional techniques developed to optimize loop nests in Fortran programs. When optimizing C or C++ applications, this package only recognizes and optimizes a particular for-loop that corresponds to the DO loop construct in Fortran programs. Within the ROSE source-to-source compiler infrastructure, such a loop is defined to have the following formats:*
> `for (i = lb; i <= ub; i+ = positiveStep)`
> *or* `for (i = ub; i >= lb; i+ = negativeStep)` ([32, p. 124])

‡ Loops can be formed with the usual `for` and `while` constructs, provided the loop iteration is *countable*:

> *The loop iterations must be countable; in other words, the number of iterations must be expressed as one of the following:*
> – *A constant.*
> – *A loop invariant term.*
> – *A linear function of outermost loop indices.*
>
> *In the case where a loops exit depends on computation, the loops are not countable.* ([19, p. 2126])

¶ The documentation [36] does not discuss which types of loop are supported clearly, but this tool seems to support only `for` (and `for each`) C++ loops.

∗∗ For more detail, refer to OpenMP's documentation [21]. In short,

> *The canonical loop nest form allows the iteration count of all associated loops to be computed before executing the outermost loop.* ([21, Section 4.4.2])

B Benchmarking Parallel Versions of PolyBench/C

Table 1 presents the gain obtained by parallelizing six different programs from the PolyBench/C suite, for different levels of optimization and data sizes, for the following 4 program categories:

1. original: unmodified and sequential programs (used as a base to measure the speedup),
2. original-autopar: original programs, where OpenMP directives are automatically inserted by `AutoPar-Clava` [4],
3. fission-manual: programs after loop fission, where OpenMP directives are inserted manually,
4. fission-autopar: programs after loops fission, where OpenMP directives are automatically inserted by `AutoPar-Clava`.

Benchmark		O0			O1			O2			O3		
Name	Data size	orig. auto	fiss. auto	fiss. man	orig. auto	fiss. auto	fiss. man	orig. auto	fiss. auto	fiss. man	orig. auto	fiss. auto	fiss. man
3mm	XS	2.28	2.29	2.31	2.46	2.50	2.75	1.73	1.73	1.98	1.81	1.96	2.04
	S	2.76	2.79	2.58	3.91	3.92	3.95	4.19	4.19	4.21	3.42	3.41	3.47
	M	2.24	2.23	2.23	3.63	3.61	3.63	3.42	3.40	3.43	3.45	3.44	3.47
	L	3.56	3.49	3.48	3.97	3.85	3.91	4.16	4.05	3.98	4.40	4.45	4.44
	XL	2.35	2.25	2.31	3.96	4.06	3.76	2.92	2.87	2.88	2.76	2.78	2.85
bicg	XS	0.53	0.45	1.44	0.23	0.23	0.75	0.23	0.20	0.75	0.28	0.26	0.75
	S	2.08	1.82	2.48	1.68	1.37	2.28	1.57	1.45	2.80	1.73	1.69	2.85
	M	3.30	2.84	2.96	3.56	2.43	2.60	4.10	3.37	3.77	4.20	3.48	3.78
	L	2.74	2.34	2.35	3.96	2.63	2.71	4.54	3.63	3.64	4.56	3.68	3.68
	XL	2.71	2.30	2.32	3.77	2.60	2.59	4.27	3.46	3.46	4.30	3.50	3.50
deriche	XS	2.03	2.08	1.63	2.24	2.19	2.26	2.36	2.54	2.29	2.20	2.22	2.35
	S	2.33	2.28	1.76	2.20	2.33	1.94	2.42	2.53	2.01	2.34	2.29	2.08
	M	2.73	2.74	1.96	3.03	3.05	2.44	3.10	3.20	2.38	2.95	3.03	2.56
	L	1.73	1.73	1.39	1.70	1.55	1.72	1.75	1.75	1.65	1.72	1.71	1.86
	XL	0.95	0.95	0.89	0.84	0.84	0.84	0.86	0.86	0.84	0.86	0.86	0.87
fdtd-2d	XS	2.17		2.33	2.13		3.01	1.42		1.81	0.77		1.00
	S	2.60		2.60	3.47		4.20	2.28		2.90	1.59		2.45
	M	1.93		2.03	1.39		3.29	1.27		3.00	0.80		2.85
	L			2.20			2.74			2.92			3.58
	XL			2.07			2.20			2.36			1.80
gesummv	XS	1.50	0.90	1.00	1.00	0.57	0.57	1.00	0.57	0.67	0.75	0.43	0.50
	S	2.71	2.33	2.26	2.58	1.55	1.66	2.36	2.04	2.19	2.82	2.62	2.62
	M	3.07	2.80	2.65	3.27	2.17	2.17	3.19	3.25	3.25	3.29	3.29	3.28
	L	3.08	2.85	2.70	3.48	1.95	2.26	4.23	4.51	4.51	3.56	3.78	3.83
	XL	2.27	2.10	1.99	3.16	2.02	1.99	3.16	3.32	3.32	3.15	3.31	3.32
mvt	XS	1.73	1.73	2.17	1.43	1.43	2.00	0.83	0.83	1.25	0.83	1.00	1.25
	S	2.73	2.75	2.85	3.64	3.56	3.56	2.54	2.49	2.85	2.51	2.38	2.71
	M	3.10	3.06	3.44	4.33	4.33	4.38	3.14	3.14	3.16	3.15	3.19	3.20
	L	2.28	2.30	2.22	3.53	2.97	3.49	2.53	2.50	2.51	2.51	2.57	2.50
	XL	1.41	1.35	1.32	2.28	1.99	2.38	1.33	1.53	1.52	1.55	1.48	1.53

Table 1. Comparing speedup between original sequential programs and transformed parallel programs, for various data sizes and compiler optimization levels.

C Cost-Benefit Considerations for Loop Fission

For the example transformations presented in Sect. 3.1, it may be that duplicating the command $y_2 = y_2 \times y_1$ degrades the speed gain of the parallelization to the point that it is not worth splitting the loop. Based on a more precise dependency analysis, such as the MWP-analysis [5], which produces a weighted dependency graph whose weights *provide quantitative information* on the dependency (linear, weak polynomial, polynomial, super-polynomial). The splitting algorithm will be adapted to use this information in order to allow for adaptive splittings. In the above case, one may then obtain the cover and corresponding algorithm presented in Sect. C. Note that Theorem 1 would still apply to this “conditional” loop-splitting algorithm.

$j = \mathbf{pow}(10, 6)$
$x_1 = 1$
$i = 1$
`parallel` ▷ A private copy of i and y_1 needs to be given to each loop below.

while $i \neq j$ **do**	**while** $i \neq j$ **do**
$x_1 = x_1 + y_1 + x_2 + i$	$y_1 = y_1 + i$
$y_1 = y_1 + i$	$y_2 = y_2 \times y_1$
$s[i] = x_1$	$u[i] = y_2$
$x_2 = x_1 + s[i]$	$t[i] = y_2 \times y_2$
$i = i + 1$	$i = i + 1$

$\mathbf{use}(x_1)$ ▷ The copies of i and y_1 can be destroyed, all have the same values.

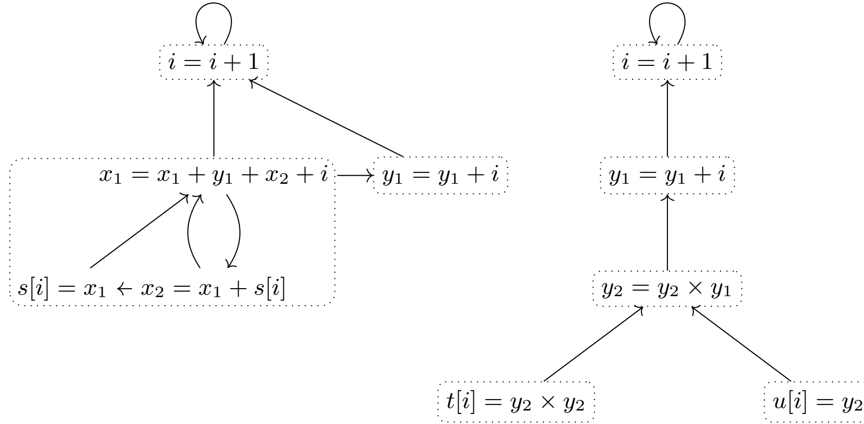


Fig. 8. Optimized Example