2017 IEEE 24th International Conference on High Performance Computing (HiPC)

Lifting Barriers Using Parallel Polyhedral Regions

Harenome Razanajato, Cédric Bastoul, Vincent Loechner
ICPS/ICube Laboratory
University of Strasbourg and INRIA, France
Email: razanajato@etu.unistra.fr, bastoul@unistra.fr, loechner@unistra.fr

***Abstract*—Nowadays best performing automatic parallelizers and data locality optimizers for static control programs rely on the polyhedral model. Polyhedral compilation consists of three phases: (1) abstracting the input code into a mathematical view; (2) analyzing and transforming this representation into an optimized alternative; (3) generating the corresponding code while ensuring it is semantically equivalent to the input code. During this last phase, state-of-the-art polyhedral compilers generate only one type of parallelism when targeting multicore shared memory architectures: parallel loops via the OpenMP `omp parallel for` directive.**

In this work, we propose to explore how a polyhedral compiler could exploit parallel region constructs. Instead of initializing a new set of threads each time the code enters a parallel loop and synchronizing them when exiting it, the threads are initialized once for all at the entrance of the region of interest, and synchronized only when it is necessary.

Technically, we propose to embed the whole region containing parallel loops in an `omp parallel` construct. Inside the parallel region, the `single` construct is used when some code needs to be executed sequentially; the `for` construct is used to distribute loop iterations between threads. Thanks to the power of the polyhedral dependence analysis, we compute when it is valid to add the optional `nowait` clause, to omit the implicit barrier at the end of a worksharing construct and thus to reduce even more control overhead.

Through a set of experiments on the PolyBench benchmarks, we show that resulting codes can overwhelm the performance obtained by the *Pluto* polyhedral compiler.

***Keywords*-code generation; OpenMP; polyhedral model; synchronizations barriers.**

I. Introduction

Optimizing and parallelizing compilers require complex loop transformations that are usually tackled using the polyhedral model, as in *GCC/GRAPHITE* [1] and *LLVM/Polly* [2]. This powerful model takes as input a static control piece of code, made of loops with affine bounds and statements reading and writing arrays through affine access functions, possibly depending on unknown constant parameters. The statements are raised into the mathematical abstraction as polytopes defining their iteration domains, then analyzed and transformed in order to extract parallelism, improve data locality, and enable vectorization. The applied transformations may include arbitrarily complex sequences of, *e.g.*, loop fusion, fission, peeling, reversal, interchange, skewing, and tiling. Finally, the code generation consists in lowering back this mathematical abstraction into a parallelized and optimized code.

The state-of-the-art polyhedral compiler Pluto [3] generates multi-threaded code by calling the loop generator CLooG [4]. This tool annotates the parallel loops to be generated, such that an OpenMP `omp parallel for` directive is inserted on top of every parallel loop that was identified. This method is pretty effective for simple cases, *e.g.*, if there is one single outer parallel loop. However, it often induces unwanted synchronizations in more complex cases: every time the execution flow encounters such a parallel for construct, the threads are started (even if they are not created again each time in modern OpenMP implementations), and there is an implicit synchronization barrier at the end of the parallel loop.

In this work, we propose to lift the implicit barriers and thread restarts which are known to introduce significant overhead [5], [6]. Our technique embeds the whole piece of code of interest into a single parallel region, and controls explicitly where to put the necessary barriers. This is done automatically thanks to the polyhedral model dependence analysis. We will exploit the following OpenMP [7] constructs and clauses: `parallel`, `for`, `single`, `nowait`, and `barrier`.

This paper is organized as follows. In Sect. II we introduce our proposal with an example, showing that it can improve performance of the resulting parallel code. The background on the polyhedral model and on the OpenMP parallel constructs is presented in Sect. III. We detail our main parallel code generation algorithm in Sect. IV. In Sect. V we show that our proposal improves the performance of many codes from the PolyBench benchmark suite. Finally, we present some related work in Sect. VI, and conclude.

II. Motivating Example

Let us consider the main computation of *atax* from the PolyBench suite [8]. The original code is given in Fig. 1. It is a linear algebra kernel that consists of a matrix transpose and a vector multiplication.

When applying the source-to-source polyhedral compiler Pluto [3] version 0.11.4 (with option `--parallel`) to this code, we get the parallelized and optimized code given in Fig. 2. The computation of array `tmp` has been done first (lines 2–10), with two parallel loops performing the

0-7695-6326-0/17/$31.00 ©2017 IEEE
DOI 10.1109/HiPC.2017.00046

IEEE computer society

Authorized licensed use limited to: INRIA. Downloaded on January 15,2026 at 15:43:40 UTC from IEEE Xplore. Restrictions apply.

```
1 for (i = 0; i < N; i++)
2   y[i] = 0;
3 for (i = 0; i < M; i++) {
4   tmp[i] = 0.0;
5   for (j = 0; j < N; j++)
6     tmp[i] += A[i][j] * x[j];
7   for (j = 0; j < N; j++)
8     y[j]+=A[i][j]*tmp[i];
9 }
```

Figure 1. Atax: original code

```
1 int t1, t2, t3;
2 #pragma omp parallel for
3 for (t2=0;t2<=M-1;t2++)
4   tmp[t2] = 0.0;
5 if (N >= 1) {
6   #pragma omp parallel for private(t3)
7   for (t2=0;t2<=M-1;t2++)
8     for (t3=0;t3<=N-1;t3++)
9       tmp[t2] += A[t2][t3] * x[t3];
10 }
11 #pragma omp parallel for
12 for (t2=0;t2<=N-1;t2++)
13   y[t2] = 0;
14 if ((M >= 1) && (N >= 1)) {
15   for (t2=0;t2<=N+M-2;t2++) {
16     #pragma omp parallel for
17     for (t3=max(0,t2-M+1);t3<=min(t2,N-1);t3++)
18       y[t3]+=A[(t2-t3)][t3]*tmp[(t2-t3)];
19   }
20 }
```

Figure 2. Atax: Pluto optimized code

initialization and the computation. Array `y` is then computed: it is first initialized in a parallel loop (l. 11–13), and then two nested loops perform its computation, the inner one being parallel (l. 15–19). Due to data-dependencies, Pluto did not parallelize the outer loop. Notice that a skewing was applied on this loop nest.

We propose in this paper to remove the unnecessary implicit synchronizations, by embedding this whole code into a single parallel region as shown in Fig. 3. The threads are created once, at entry of the region. Then, they are synchronized only when necessary. In this example, the last two loop nests require a synchronization because of the successive writes to array `y`. But the *nowait* clause can be added to the first two loops: the dependencies from the first to the second parallel loop are verified, since the same threads access the same array elements of array `tmp`; and there is no dependency from the second to the third parallel loop, accessing different arrays.

The performance of those versions of the code, depending on the dataset size, are given in Table I. The baseline (1x) is the speed of the sequential version. The experimental platform is a 6-core Intel processor, as described in Sect. V. Our custom parallel region version is on average 1.97x faster than the Pluto version. This is mainly due to the OpenMP thread re-creation that is avoided inside the last parallel loop of this code.

```
1 #pragma omp parallel
2 {
3   int t1, t2, t3;
4   #pragma omp for nowait
5   for (t2=0;t2<=M-1;t2++)
6     tmp[t2] = 0.0;
7   if (N >= 1) {
8     #pragma omp for private(t3) nowait
9     for (t2=0;t2<=M-1;t2++)
10      for (t3=0;t3<=N-1;t3++)
11        tmp[t2] += A[t2][t3]*x[t3];
12  }
13  #pragma omp for
14  for (t2=0;t2<=N-1;t2++)
15    y[t2] = 0;
16  if ((M >= 1) && (N >= 1)) {
17    for (t2=0;t2<=N+M-2;t2++) {
18      #pragma omp for
19      for (t3=max(0,t2-M+1);t3<=min(t2,N-1);t3++)
20        y[t3]+=A[(t2-t3)][t3]*tmp[(t2-t3)];
21    }
22  }
23 }
```

Figure 3. Atax: synchronization optimized (custom) code

Table I
PERFORMANCE OF THE ATAX BENCHMARK

dataset	pluto version	custom version
medium	0.46x	1.31x
large	1.24x	1.94x
extralarge	1.27x	2.19x

Before presenting our general code generation algorithm, the next section will introduce some background on the polyhedral model and on OpenMP.

III. BACKGROUND

A. Polyhedral Model

The polyhedral model (or polytope model) is a mathematical framework that can be used to analyze and transform Static Control Parts (SCoP) of programs.

SCoP transformation using the polyhedral model requires three steps. First, adequate programs must be raised to a polyhedral representation. Such a representation shall include several characteristics of the input SCoP: the iteration domains of the statements, the original scheduling and the dependencies. In the second step, a new scheduling can be determined. The new scheduling may reorder statement instances as long as no dependency is violated. Finally, a new code implementation that follows the new scheduling is generated.

1) Polyhedral Representation: polyhedral frameworks manipulate affine relation abstractions, which map sets of *input vectors* to sets of *output vectors* with respect to affine constraints. Such relations represent all the various aspects of a program such as iteration domains or dependencies between iterations.

Iteration Domains – The polyhedral model revolves around the concept of *statement instance*, *i.e.* a given ex-

Authorized licensed use limited to: INRIA. Downloaded on January 15,2026 at 15:43:40 UTC from IEEE Xplore. Restrictions apply.

```
1 for (i = 0; i < N; ++i)
2   for (j = 0; j < M; ++j)
3     S1: A[j] = A[j] + B[i];
```

Figure 4. SCoP example

ecution of a statement. Statement instances are identified by their *iteration vectors* which consist of the values of the loop iterators enclosing the statement. The *iteration domain* of a statement is the set of its possible iteration vectors. It may depend on fixed yet unknown values, called *parameters*. The iteration domain $\mathcal{D}$ for a statement S can be represented as a polyhedron:

$$\mathcal{D}_S(\vec{p}) = \left\{ \vec{\imath}_S \;\middle|\; D_S \begin{pmatrix} \vec{\imath}_S \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\} \quad (1)$$

where $\vec{p}$ is the vector of parameters, $\vec{\imath}_S \in \mathbb{Z}^{dim(\vec{\imath}_S)}$ stands for an iteration vector of statement S, and $D_S \in \mathbb{Z}^{m_{D_S} \times (dim(\vec{\imath}_S)+dim(\vec{p})+1)}$ — where m_{D_S} is the number of constraints — is an integer matrix that encodes the constraints.

For instance, considering the statement `S1` from Fig. 4, $(0,0)$ and $(0,1)$ would be two possible *iteration vectors* (assuming $N \geq 1$ and $M \geq 2$), whereas $(0, M+2)$ would not be possible. The *constraints* for the iteration vectors (i,j) of statement `S1` would be:

$$\begin{cases} i & \geq 0 \\ i & < N \\ j & \geq 0 \\ j & < M \end{cases} \Leftrightarrow \begin{cases} i & \geq 0 \\ -i+N-1 & \geq 0 \\ j & \geq 0 \\ -j+M-1 & \geq 0 \end{cases}$$

And the *iteration domain* would be:

$$\mathcal{D}_{S1}\begin{pmatrix} N \\ M \end{pmatrix} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \middle| \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \\ N \\ M \\ 1 \end{pmatrix} \geq \vec{0} \right\}$$

Here, the iteration vector $\vec{\imath}_{S1}$ is composed of the two loop indices (i,j) and the vector of parameters $\vec{p}$ of the two unknown program variables (N,M).

Dependence Relations – Dependences between statement instances of a source statement *S* and a target statement *T* can be represented as a polyhedron. Each integer point in this polyhedron signifies that there is a dependency between the corresponding input and output *iteration vectors*. Such a polyhedron can be defined by the following relation:

$$\delta_{S,T}(\vec{p}) = \left\{ \vec{\imath}_S \rightarrow \vec{\imath}_T \;\middle|\; R_{S,T} \begin{pmatrix} \vec{\imath}_S \\ \vec{\imath}_T \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\} \quad (2)$$

For the SCoP example in Fig. 4 there is a dependency due to the consecutive accesses to array `A`, from iteration (i,j) to iteration (i',j'), when $i' > i$ and $j' = j$. The dependence relation would be:

$$\delta_{S1,S1}(\vec{p}) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} i' \\ j' \end{pmatrix} \middle| \begin{bmatrix} -1 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \\ i' \\ j' \\ N \\ M \\ 1 \end{pmatrix} \geq \vec{0} \right\}$$

This constraints matrix corresponds to $i' > i$ and $j' = j$.

2) Scheduling: scheduling relations determine the temporal ordering between statement instances. To do so, each instance of a statement is associated with a logical date $\vec{t}_S$.

$$\theta_S(\vec{p}) = \left\{ \vec{\imath}_S \rightarrow \vec{t}_S \;\middle|\; T_S \begin{pmatrix} \vec{\imath}_S \\ \vec{t}_S \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\} \quad (3)$$

Parallelizing SCoPs amounts to determining a new scheduling where multiple statement instances have the same logical date and/or where some logical date dimensions are identified as parallel, while ensuring that all dependencies are preserved [3], [9].

3) Code Generation: once a new scheduling has been decided, the corresponding code can be generated. Such algorithms shall produce a code that scans each point of the polyhedra in the order specified by the newly found scheduling [10], [11]. Most recent refinements to the code generation problem can be found in CLooG [4], CodeGen+ [12] and isl [13].

B. Parallel Constructs

Source-to-source automatic parallelization tools, such as Pluto [3] or R-Stream [14], generate parallel code from a sequential program by implementing the whole process presented in the previous subsection. When targeting general purpose shared memory architectures such as multicore CPUs, parallelism is expressed with parallel loop constructs, usually implemented using OpenMP [7]. Our work opens a wider use of classical parallel constructs such as parallel *regions*, and some work sharing and synchronization mechanisms.

1) Parallel Region: parallel computing based on the fork-join model uses special program parts where parallel execution branches begin at their starting point and where the sequential execution is resumed at their ending point. Such program parts are called *parallel regions*. In practice, once a master thread reaches a parallel region, worker threads are requested to execute the code within that region along with the master thread. Worker threads are stopped and the master thread continues the execution alone when and only when all threads have reached the end of the parallel region. In OpenMP, the pragma `omp parallel` is used to specify that the work enclosed in the construct is a parallel region. Parallel regions come with a creation

Authorized licensed use limited to: INRIA. Downloaded on January 15,2026 at 15:43:40 UTC from IEEE Xplore. Restrictions apply.

and management overhead. Moreover, data affinity is not guaranteed between threads of different parallel regions. Hence, optimized parallel implementations should merge parallel regions when it is possible, *e.g.* when a parallel region is enclosed within a loop.

2) Work sharing: master and worker threads execute the same code within a parallel region unless a work sharing construct is specified.

- *Loop constructs* split up loop iterations among the threads according to a scheduling policy. In our work, we may use the *static* scheduling policy. It specifies that iterations are divided equally among threads (except maybe the last thread). A *chunk* number of iterations may be specified. In this case, *chunk* number of contiguous iterations will be allocated to each thread, in a round-robin fashion. In OpenMP, the pragma `omp for` is used to specify work sharing for the loop enclosed in the construct. The shortcut `omp parallel for` may be used to specify a parallel region containing only one loop work sharing construct.
- *Single constructs* specify that a code block is executed by only one thread. In OpenMP, the pragma `omp single` is used to make a single thread execute the work enclosed in the construct.

Work sharing constructs have synchronization barriers implied in the end of the construct: all threads have to wait for all the others before resuming their execution. Optimized parallel implementations should explicitly remove them when they are not necessary.

3) Synchronization: specific constructs allow to add or to remove thread synchronizations within a parallel region.

- *Barrier constructs* specify that all threads executing a parallel region must wait for each other at that point before resuming their execution. In OpenMP, the pragma `omp barrier` is used to specify such a construct.
- *Barrier lifting constructs* specify that the implicit barrier at the end of a work sharing construct can be removed. In OpenMP, the `nowait` clause may be specified along with a work sharing construct to remove its implicit barrier at the end of the work enclosed in the construct.

An optimized parallel implementation may use such constructs to guarantee the correctness of the parallel code or to avoid the synchronization overhead when it is not necessary.

IV. Parallel Region Generation

A. Parallel Regions

Current code generation methods in polyhedral compilers create an OpenMP parallel region for each parallel loop. This means that for each parallel loop: 1) a thread team is created at the start of the loop; 2) the thread team must *always* synchronize at the end of the loop (because it is the end of the parallel region); and 3) the thread team is destroyed at the end of the loop. Although modern OpenMP implementations attempt to be clever in regard to this matter, consequent control overhead still remains [5], [6].

We propose to refine current code generation methods by generating a single parallel region, when it is profitable: if there is only an outer parallel loop, it is useless. But when parallel loops are enclosed in one or many outer sequential loops, the overhead of multiple parallel regions can be avoided by creating a single outer parallel region. Our technique should be applied only on loop nests containing inner parallel loops or when factorizing multiple parallel loops into a single parallel region is possible.

Code generation of SCoPs shall then generate an `omp parallel` directive at the start of the corresponding code and generate `omp for` directives instead of `omp parallel for` on parallel loops. However, using a single parallel region for a whole SCoP raises new redundant execution and synchronization problems. Indeed, statements may not all be identified as potentially parallel. Thus, such parts must be protected with the OpenMP `single` construct to ensure they are executed only once.

Enclosing multiple worksharing constructs in a common parallel region provides the opportunity to introduce the `nowait` clause. This clause specifies that the implicit barrier at the end of a worksharing construct may be omitted. Whenever two adjacent worksharing constructs share no dependency, the `nowait` clause may be used on the first construct. Moreover, the OpenMP specification states that under certain circumstances, the `nowait` clause may be safely used on a `for` construct which precedes another `for` if the latter loop's statement instances depend only on the same logical iteration of the former loop. This is possible if: 1) the sizes of both *iteration domains* are equal, 2) the *chunk size* is either the same for both loops or not specified, 3) both loops are bound to the same parallel region, 4) none of the loops is associated with a `SIMD` construct. Safe use of the `nowait` clause in this fashion also requires the scheduling to be `static`. This is the default behaviour for current known implementations of OpenMP but the specification states that it should be enforced with the `schedule(static)` clause.

Special care must be dedicated to worksharing constructs enclosed in loops. If the worksharing construct of interest is the last of the outer loop, it may *precede* the next worksharing construct in the generated code as well as the first worksharing construct in the outer loop. In this case, determining whether the `nowait` clause can be used requires to analyze multiple dependencies. The minimal requirement is the possibility to use the `nowait` clause between constructs within the loop. If the next worksharing construct is not compatible with the `nowait` clause, a `barrier` construct must be used right before the latter construct instead of right after the former loop. Figure 5

Authorized licensed use limited to: INRIA. Downloaded on January 15,2026 at 15:43:40 UTC from IEEE Xplore. Restrictions apply.

```
1  #pragma omp parallel
2  {
3    for (i = 0; i < N; ++i)
4      #pragma omp for nowait
5      for (j = 0; j < M; ++j)
6        S1: A[j] = A[j] + B[i];
7    #pragma omp barrier
8    #pragma omp for
9    for (i = 0; i < M; ++i)
10     for (j = 0; j < i; ++j)
11       S2: C[i] = C[i] * A[j];
12 }
```

Figure 5. Example of SCoP where the `nowait` clause requires a barrier

presents such an example: the `nowait` clause can safely be used for successive iterations of the loop on `i` for statement *S1* but the very last batch of iterations must be completed before *S2* can be executed.

B. Determining the validity of the `nowait` clause

Obviously, if two worksharing constructs next to each other do not depend on each other, it is valid to add the `nowait` clause to the first worksharing construct regardless of the worksharing construct type or the schedule type for loops. The implicit barrier of the second worksharing construct maintains the synchronization with the remainder of the code.

As explained in the previous subsection, the OpenMP specification describes four conditions for the safe use of `nowait` (as long as the schedule type is set to `static`) when there are dependencies between subsequent loop constructs. The first condition can be verified using the polyhedral model: determining that the sizes of two iteration domains coincide is possible with Ehrhart polynomials [15]. Ensuring that the second, third and fourth conditions are fulfilled is trivial: our code generation algorithm aims to use a single parallel region and thus enforces by design that the worksharing constructs bind to the same parallel region. In the same vein, ensuring that the chunk sizes are identical (or not specified) and that no `SIMD` construct is associated is easy.

If the aforementioned conditions are met, compliant OpenMP implementations must assign the same logical iterations to the same threads. Hence, the last step is to ensure that the only existing dependencies lie between identical logical iterations.

Let $S1$ and $S2$ be two statements such that $S2$ depends on $S1$ as follows:

$$\delta_{S1,S2}(\vec{p}) = \left\{ \vec{i}_{S1} \rightarrow \vec{i}_{S2} \;\middle|\; R_{S1,S2} \begin{pmatrix} \vec{i}_{S1} \\ \vec{i}_{S2} \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\}$$

Assuming d_{S1} is the parallel dimension for $S1$ and d_{S2} is the parallel dimension for $S2$, the following polyhedron links the same logical iterations of $S1$ and $S2$:

$$\mathcal{P}_{S1,S2}(\vec{p}) = \left\{ \vec{i}_{S1} \rightarrow \vec{i}_{S2} \;\middle|\; \left(d_{S1} = d_{S2} \right) \right\}$$

The two previous equations combine to get the expression of dependencies in different logical iterations of $S1$ and $S2$ as: $\delta_{S1,S2} \setminus \mathcal{P}_{S1,S2}$. It immediately follows that the necessary condition for the validity of the `nowait` clause on statement $S1$ is:

$$\delta_{S1,S2} \setminus \mathcal{P}_{S1,S2} \equiv \emptyset$$

C. Code Generation

The general algorithm for parallel region annotation is described in Alg. 1. The input of the algorithm is an optimized (using the polyhedral techniques) Abstract Syntax Tree (AST), encoded as a list of AST nodes. Any loop previously identified as parallel shall be annotated with `#pragma omp for` (instead of `#pragma omp parallel for`) whereas any other statement shall be annotated with `#pragma omp single`. In this polyhedral AST, we temporarily (*i.e.* until the code is actually pretty-printed) introduce macro statements: we consider the body of a worksharing construct as a single statement.

Algorithm 1 Parallel Regions Annotation

Input: node list *n*

```
1: for all statements s in n do
2:   if s is a parallel loop then
3:     annotate s with #pragma omp for
4:   else
5:     annotate s with #pragma omp single
6:   end if
7: end for
8: annotate_nowait(n)
```

At line 8 of Alg. 1 we call algorithm Alg. 2, to annotate as many worksharing constructs as possible with the `nowait` clause. For any given AST node *n*, *n→next* references the next node (if any) at the same AST level. *n→nowait* indicates whether the node of interest requires a barrier. Successive nodes marked with `nowait` are considered to belong to the same nowait-group. If a node is a loop, *n→inner* corresponds to its body. *n→last* is used to either point to the last nowait-group in *n→inner* or to the nowait-group that contains *n* in the case it is not a loop. The value of *n→last* is set by the function `compute_last` (called lines 4 and 9). `last_node` returns the last AST node of a nowait-group. The `dep` function (lines 5 and 15) takes as input two nowait-groups of nodes g_1 and g_2 and returns true if and only if for each pair of nodes $(n_1 \in g_1, n_2 \in g_2)$ there is no dependency that would be violated if the `nowait` clause was added to all nodes of the nowait-group g_1.

The algorithm attempts to leverage as many barriers as possible: it assumes at first that no node requires a barrier

Authorized licensed use limited to: INRIA. Downloaded on January 15,2026 at 15:43:40 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2 `annotate_nowait`

Input: node list n

```
 1: n→nowait = true
 2: if n→inner then
 3:    annotate_nowait(n→inner)
 4:    compute_last(n)
 5:    if !dep(n→last, n→inner) then
 6:       last_node(n→last)→nowait = false
 7:    end if
 8: else
 9:    compute_last(n)
10: end if
11: if n→next then
12:    annotate_nowait(n→next)
13:    if !n→inner or last_node(n→last)→nowait then
14:       n→nowait = false
15:       if dep(n→last, n→next) then
16:          n→nowait = true
17:       end if
18:    end if
19: end if
```

(line 1) and then restores barriers where needed (lines 6 and 14–17). At each level of the AST, the algorithm recurses (line 3) into the bodies of loops (n→*inner*) and on the next node of the level (line 12). The first and last nowait-groups (which can coincide) of the body of a loop are of the utmost importance. The dependencies between these two nowait-groups must be analyzed (line 5) because the last nowait-group will precede the next iteration of the first nowait-group. If this condition is not verified, the last node of the last nowait-group requires a barrier (line 6). If the node n is a loop, checking its dependencies with its successor nowait-group (line 15) and possibly restoring a barrier is only necessary if no barrier has been restored in its body (line 13). When doing so, the barrier is temporarily restored (line 14) so that `dep` (line 15) always considers n→*last* and n→*next* as different nowait-groups (because n→*last* may refer to n if it is not a loop).

The pretty printing phase is akin to current techniques, with some modifications. The differences are: 1) the whole SCoP must be surrounded with the `#pragma omp parallel { ... }` construct, 2) `#pragma omp for` must be used instead of `#pragma omp parallel for`, 3) `#pragma omp single` constructs must be printed where needed, 4) the `nowait` clause must be added to a worksharing construct when the corresponding n→*nowait* is true and 5) `#pragma omp barrier` are printed where needed (after node n when n→*nowait* is false). To simplify the generated code, the pretty printing phase may also decide to omit both the barrier and the preceding nowait clause, when the barrier would be equivalent to the preceding work sharing construct's implicit barrier.

Table II
BENCHMARKS MAIN CHARACTERISTICS

benchmark	#main loop nests	#inner paral. loops	#single	#nowait
adi	2	2	0	0
adi-tile	2	2	0	0
atax	4	1	0	2
bicg	4	1	0	2
cholesky	2	2	2	1
cholesky-tile	2	2	0	0
correlation	9	2	0	1
covariance	7	1	0	2
doitgen	3	3	0	1
doitgen-tile	3	3	0	2
fdtd-2d	6	4	3	2
fdtd-2d-tile	1	1	0	0
floyd-warshall	1	1	0	0
floyd-warshall-tile	1	1	0	0
gemver	3	1	0	0
gramschmidt	8	3	1	1
gramschmidt-tile	5	2	1	1
heat-3d	11	4	7	5
heat-3d-tile	1	1	0	0
jacobi-1d	5	4	7	5
jacobi-1d-tile	1	1	0	0
jacobi-2d	10	4	7	4
jacobi-2d-tile	1	1	0	0
lu	2	1	1	0
lu-tile	2	1	0	0
nussinov	2	1	0	0
nussinov-tile	1	1	0	0
reg_detect	16	3	4	2
reg_detect-tile	1	1	0	0
seidel-2d	1	1	0	0
seidel-2d-tile	1	1	0	0
trisolv	2	1	1	1
trisolv-tile	2	1	0	0
trmm	2	1	0	0

V. BENCHMARKS

We evaluated our approach with benchmarks taken from the PolyBench suite [8]. We ran Pluto version 0.11.4 on all of them, in two flavors: the first one with automatic parallelization enabled (option `--parallel`), and the second one with parallelization and tiling (option `--tile --parallel`). From those two flavors of each benchmark, we selected the ones where our method was applied, *i.e.* where at least one internal parallel loop occurs. This happens on 34 benchmarks out of 50 (2*25), 14 of them being the tiled version. Table II describes the main characteristics of each of these benchmarks: the number of main loop nests that they contain, the number of parallel loops embeded into outer sequential loops, the number of single regions, and the number of nowait clauses that are introduced by our method.

We conducted our experiments on three platforms:

1) An Intel Xeon E5-2620v3 @ 2.40GHz (6 cores, 12 threads), running the Linux 4.11.5 kernel. Intel Turbo Boost and Hyperthreading were dynamically disabled during the execution of the benchmarks. In order to further reduce the variance of the measurements, Linux FIFO

Authorized licensed use limited to: INRIA. Downloaded on January 15,2026 at 15:43:40 UTC from IEEE Xplore. Restrictions apply.

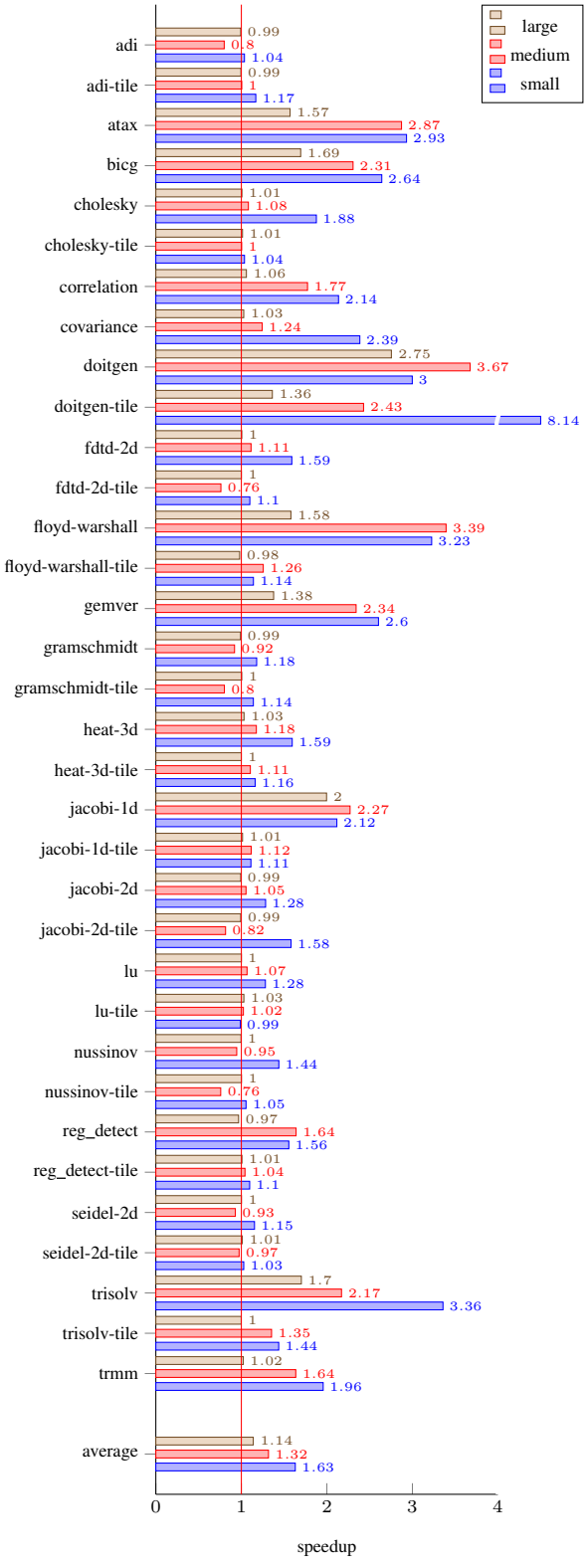

Figure 6. Speedup over Pluto, first platform (6-cores/gcc)

scheduling was enabled via the PolyBench's macro `POLYBENCH_LINUX_FIFO_SCHEDULER`. The compiler is `gcc 7.1.1` using options `-O3 -march=native -fopenmp`.

2) The second platform is a dual socket Intel Xeon E5-2650v3 @ 2.30GHz (2*10 cores, 40 total threads), running Linux 4.4.0. The compiler is `gcc 5.4.0`, using options `-O3 -march=native -fopenmp`. No particular environment variable was set on this platform to get stable measurements.
3) The last platform is the same computer, but using this time the `icc` compiler version `17.0.0` and options `-O3 -march=native -qopenmp`. We had to put the environment variable `OMP_NUM_THREADS` to 20 in order not to use hyperthreading to get more stable measurements. Still, the variance was much higher on this configuration than on the previous ones, so the results on this platform are less reliable: the variance exceeds 5% in about half of the measurements that we made.

The environment variable `OMP_PROC_BIND` was set to `true` on all platforms. The PolyBench scripts that we used perform all time measurements as the average of 3 median measurements out of 5 runs. Notice that we report some speedups that are larger than the number of available cores; this is not a surprise as Pluto is not only a parallelizer but also a data locality optimizer and vectorizer.

Figure 6 presents the acceleration of our version compared to the Pluto version on the 6-cores first platform for the small, medium, and large datasets of the PolyBench suite. We can notice from this figure that our method improves many of these benchmarks: the acceleration is most often greater than 1x. The geometric mean acceleration for all datasets is given on the bottom line. On *smaller* datasets, the benefit of our method is often *greater*: in many cases the ratio between threads creation and synchronizations time towards computation time is higher when computing small datasets. The overall mean acceleration on all dataset sizes available in PolyBench (including mini and extralarge, not shown in the figure) is 1.36x.

However, we noticed that in some of these benchmarks, Pluto did not improve the performance over the sequential version of the code. In order to support our conclusions, we checked that our method improves both the efficient parallelized codes and the inefficient ones. Figure 7 presents both the acceleration of the Pluto version and the one of our version, over the sequential version on the standard dataset. The geometric mean of our version speedup over Pluto, when Pluto performs worse than 1x is 1.95x, and when Pluto performs better than 1x it is 1.09x. So our method improves more the poor performing Pluto codes, which is not a surprise: the synchronization over computation time ratio is usually higher on those codes. Nevertheless, we checked that our method improves the performance of most

Authorized licensed use limited to: INRIA. Downloaded on January 15,2026 at 15:43:40 UTC from IEEE Xplore. Restrictions apply.

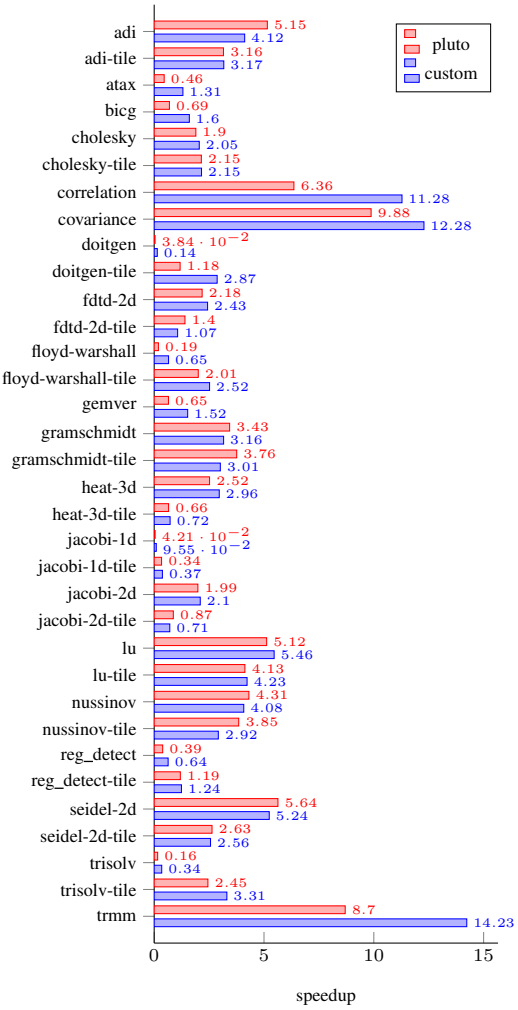

Figure 7. Speedup of the Pluto and our custom versions over the sequential version (first platform (6-core/gcc), medium dataset

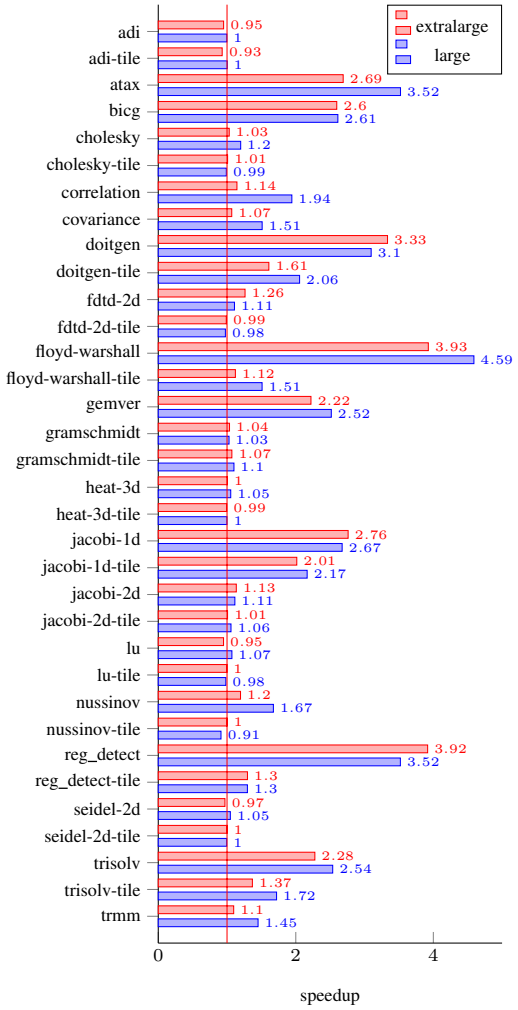

Figure 8. Speedup over Pluto, second platform (40-threads/gcc)

of the Pluto parallel codes, whether Pluto performs well or not. This assessment is confirmed by the measurements on the other dataset sizes and platforms; we did not put all of them in this paper for space reasons.

We also ran those benchmarks on 40 threads in the second configuration (2x 10-cores hyperthreaded), to get the results presented in Fig. 8. The geometric mean of those accelerations is 1.52x on the large dataset, and 1.39x on the extralarge dataset.

Finally, we ran the benchmarks on the 20 threads third configuration, using the icc compiler and the Intel OpenMP runtime. Each benchmark acceleration of our version over the Pluto version is given in Fig. 9 for the large and extralarge datasets. Some benchmarks, marked with *, are not reported since there is a numerical divergence between the different versions, probably due to the icc vectorizer: the vector floating point unit does not have the same precision as the main one. The average acceleration of our version is respectively 1.14x and 1.11x for the large and extralarge datasets. The overall acceleration is a bit lower than the previous ones, most probably due to the more efficient OpenMP runtime. But those measurements are less reliable, as said before, since the variance in time measurements often exceeds the PolyBench default limit (5%).

VI. Related Work

Loop parallelization received a lot of attention in the optimizing compilation community because loops are known to embed a significant part of the overall computation time. Allen and Kennedy's parallelization algorithm computes strongly connected components of the dependence graph to decide about convenient loop distribution to extract parallel loops [16]. Wolf and Lam's perfectly nested loop parallelization algorithm uses a unified representation of a subset of loop transformations, known as unimodular transformations,

Authorized licensed use limited to: INRIA. Downloaded on January 15,2026 at 15:43:40 UTC from IEEE Xplore. Restrictions apply.

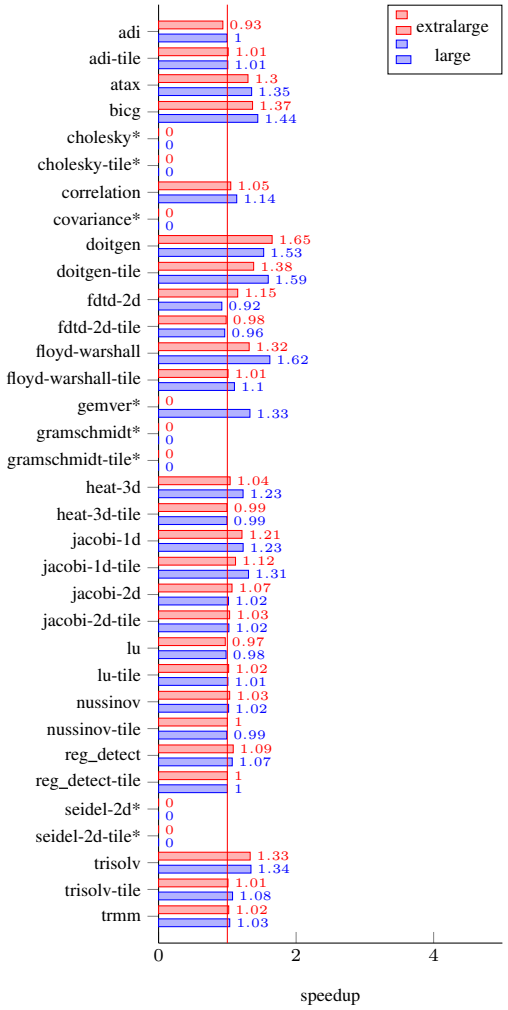

Figure 9. Speedup over Pluto, third platform (20-cores/icc)

to extract parallelism [17]. Feautrier's parallelization algorithm was the first one to propose a general solution to the (innermost) parallelism extraction problem, by computing an affine transformation [18], [9]. Feautrier's work has been extended in may ways, e.g., by Lim and Lam to extract outermost parallel loops [19], [20], or by Bondhugula et al. as a general polyhedral framework to optimize parallelism and data locality [3], [21]. These techniques are now included in many high-level compilers such as Pluto [3], R-Stream [14] or TRACO [22], and also in low-level compilers such as GCC [1], LLVM [2] or IBM XL [23].

Few algorithms are designed to generate synchronizations e.g., Allen-Kennedy[16] and Lim-Lam [19], but most of them aim at generating parallel loop constructs with an implicit barrier synchronization at the end of each parallel loop, such as the `omp parallel for` OpenMP parallel construct [7]. Our technique is complementary to these works: it may take as input the optimized generated code (or its internal polyhedral representation) and build the convenient parallel region construct to minimize the runtime overhead and remove spurious synchronizations.

Synchronization placement and optimization has been the subject of many past works, including barrier placement [24], [25] and removal [26], [27]. Works on reducing synchronization overhead through generating merged SPMD programs from fork-join implementations are the closest to our approach [28], [26], [27]. In particular, we share the idea to put several smaller regions together to remove barriers. However our context and techniques are quite different since in our case the code is not modified (except parallel constructs) and a specific analysis allows to safely remove barriers, while in the SPMD approach, broadcast barriers are removed as a consequence of merging regions and of the choice of work distribution among threads. Zhao et al. also proposed a technique which reduces task creation overhead [29] but in the context of task-parallel programs.

VII. Conclusion

In this paper, we presented a technique to generate wide *parallel regions* rather than separate *parallel loops*. Our approach brings many advantages to most high-level optimizing compilers that are relying on parallel loop construct only. First, it minimizes the overhead induced by starting/stopping computation threads. Second, it allows to remove unnecessary and costly synchronizations. Lastly, threads may take advantage of data locality between loops. Our method is not competing but is *complementary* to existing parallelization frameworks: its input is an already optimized code and its output is an even more efficient optimization. It exploits the polyhedral representation of programs and a custom code generation phase to factorize parallel loops into wider and deeper loop regions where superfluous synchronizations have been removed. We conducted a wide experimental study showing that our approach is nearly always beneficial and brings a significant gain over the state-of-the-art Pluto compiler, from 1.14 to 1.63 speedup in average, depending on the dataset size. A clear conclusion of our study is that high-level polyhedral compilers should now target parallel regions rather than collections of independent parallel loops.

Ongoing work aims at co-designing an automatic parallelization algorithm along with our parallel region generation technique, since a more appropriate parallelism form is likely to bring additional optimization opportunities.

References

[1] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, and R. Upadrasta, "Graphite two years after: First lessons learned from real-world polyhedral compilation," in *GCC Research Opportunities Workshop (GROW'10)*, Pisa, Italy, 2010.

Authorized licensed use limited to: INRIA. Downloaded on January 15,2026 at 15:43:40 UTC from IEEE Xplore. Restrictions apply.

[2] T. Grosser, A. Größlinger, and C. Lengauer, "Polly – performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, 2012.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," *SIGPLAN Notices*, vol. 43, no. 6, pp. 101–113, 2008.

[4] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT'13, IEEE International Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, France, September 2004, pp. 7–16.

[5] J. M. Bull, "Measuring synchronisation and scheduling overheads in openmp," in *Proceedings of First European Workshop on OpenMP*, vol. 8, 1999, p. 49.

[6] M. Bull, F. Reid, and N. M. Donnell, "Epcc openmp microbenchmark suite, version 3.1," 2015. [Online]. Available: https://www.epcc.ed.ac.uk

[7] OpenMP Architecture Review Board, "Openmp 4.5 specifications." [Online]. Available: http://openmp.org

[8] L.-N. Pouchet, "Polybench/c 4.1: The polyhedral benchmark suite," 2015. [Online]. Available: http://polybench.sourceforge.net

[9] P. Feautrier, "Some efficient solutions to the affine scheduling problem. part ii. multidimensional time," *International journal of parallel programming*, vol. 21, no. 6, pp. 389–420, 1992.

[10] W. Kelly, W. Pugh, and E. Rosser, "Code generation for multiple mappings," in *Frontiers of Massively Parallel Computation, 1995. Proceedings. Frontiers' 95., Fifth Symposium on the*. IEEE, 1995, pp. 332–341.

[11] F. Quilleré, S. Rajopadhye, and D. Wilde, "Generation of efficient nested loops from polyhedra," *International Journal of Parallel Programming*, vol. 28, no. 5, pp. 469–498, 2000. [Online]. Available: http://dx.doi.org/10.1023/A:1007554627716

[12] C. Chen, "Polyhedra scanning revisited," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 499–508, 2012.

[13] T. Grosser, S. Verdoolaege, and A. Cohen, "Polyhedral ast generation is more than scanning polyhedra," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 37, no. 4, p. 12, 2015.

[14] B. Meister, N. Vasilache, D. Wohlford, M. M. Baskaran, A. Leung, and R. Lethin, "R-stream compiler," in *Encyclopedia of Parallel Computing*, 2011, pp. 1756–1765.

[15] P. Clauss, "Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs," in *Proceedings of the 10th International Conference on Supercomputing*, 1996, pp. 278–285.

[16] J. Allen and K. Kennedy, "Automatic translation of FORTRAN programs to vector form," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 4, pp. 491–542, Oct. 1987.

[17] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 4, pp. 452–471, Oct. 1991.

[18] P. Feautrier, "Some efficient solutions to the affine scheduling problem. i. one-dimensional time," *International journal of parallel programming*, vol. 21, no. 5, pp. 313–347, 1992.

[19] A. W. Lim and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine transforms," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997, pp. 201–214.

[20] A. Lim, "Improving parallelism and data locality with affine partitioning," Ph.D. dissertation, Stanford University, 2001.

[21] U. Bondhugula, A. Acharya, and A. Cohen, "The pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests," *ACM Trans. Program. Lang. Syst.*, vol. 38, no. 3, pp. 12:1–12:32, Apr. 2016.

[22] W. Bielecki, M. Palkowski, and T. Klimek, "Free scheduling for statement instances of parameterized arbitrarily nested affine loops," *Parallel Comput.*, vol. 38, no. 9, pp. 518–532, Sep. 2012.

[23] U. Bondhugula, O. Günlük, S. Dash, and L. Renganarayanan, "A model for fusion and code motion in an automatic parallelizing compiler," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT'10. Vienna, Austria: ACM, Sep. 2010, pp. 343–352.

[24] A. Aiken and D. Gay, "Barrier inference," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998, pp. 342–354.

[25] A. Darte and R. Schreiber, "A linear-time algorithm for optimal barrier placement," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '05, 2005, pp. 26–35.

[26] C.-W. Tseng, "Compiler optimizations for eliminating barrier synchronization," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95, 1995, pp. 144–155.

[27] H. Han, C.-W. Tseng, and P. Keleher, "Eliminating barrier synchronization for compiler-parallelized codes on software dsms," *International Journal of Parallel Programming*, vol. 26, no. 5, pp. 591–612, Oct 1998.

[28] R. Cytron, J. Lipkis, and E. Schonberg, "A compiler-assisted approach to spmd execution," in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '90, 1990, pp. 398–406.

[29] J. Zhao, J. Shirako, V. K. Nandivada, and V. Sarkar, "Reducing task creation and termination overhead in explicitly parallel programs," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10, 2010, pp. 169–180.

Authorized licensed use limited to: INRIA. Downloaded on January 15,2026 at 15:43:40 UTC from IEEE Xplore. Restrictions apply.