Received: 1 December 2017 | Revised: 1 November 2018 | Accepted: 23 December 2018
DOI: 10.1002/cpe.5168

WILEY

RESEARCH ARTICLE

A study on popular auto-parallelization frameworks

S. Prema[1] | Rupesh Nasre[2] | R. Jehadeesan[1] | B.K. Panigrahi[1]

[1]Electronics and Instrumentation Group, Indira Gandhi Center for Atomic Research, Homi Bhabha National Institute, Chennai, India
[2]Department of Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India

Correspondence
Rupesh Nasre, Department of Computer Science and Engineering, Indian Institute of Technology Madras, Chennai-600 036, India.
Email: rupesh@cse.iitm.ac.in

Funding information
Department of Atomic Energy, Government of India

Summary

We study five popular auto-parallelization frameworks (Cetus, Par4all, Rose, ICC, and Pluto) and compare them qualitatively as well as quantitatively. All the frameworks primarily deal with loop parallelization but differ in the techniques used to identify parallelization opportunities. Due to this variance, various aspects, such as certain loop transformations, are supported only in a few frameworks. The frameworks exhibit varying abilities in handling loop-carried dependence and, therefore, achieve different amounts of speedup on widely used PolyBench and NAS parallel benchmarks. In particular, Intel C Compiler (ICC) fares as an overall good parallelizer. Our study also highlights the need for more sophisticated analyses, user-driven parallelization, and meta-auto-parallelizer that provides combined benefits of various frameworks.

KEYWORDS

loop-carried dependence, loop transformations, privatization, vectorization

1 | INTRODUCTION

Quest for performance has made multicore processors mainstream. It becomes vital for the programmers and algorithm developers to exploit these ubiquitous advanced architectures with parallel programming. Exploiting the potential of multicore processors through parallel programming is a significant challenge. Amid several approaches to tame this challenge, a promising and programmer-friendly approach is *automatic parallelization*.[1-3] Auto-parallelizers eliminate the need for a programmer to transform a sequential code into a parallel code, which is quite attractive. Source-to-source transformations with insertion of parallel directives are performed by auto-parallelizers, such as Cetus,[4-7] Par4all,[8,9] Pluto,[10,11] Parallware,[12,13] Rose,[14,15] Intel C Compiler,[16] LLVM (Low Level Virtual Machine) Polly,[17] ParaWise,[18] ParaGraph,[19] SUIF (Stanford University Intermediate Format),[20-22] and Polaris.[23,24] Although these existing parallelizers offer considerable benefits, they still fall short of fully replacing the manual transformations. Several parallelizers do not utilize all the static information available, whereas several others fall short of modeling precision. It leads to either missed parallelism opportunities or unwanted parallelization of sequential codes. As a consequence, compared to the original sequential version, the auto-parallelized code may lead to parallelization overheads and may exhibit poorer performance.[25]

Earlier studies[26,27] of the parallelizing compilers using Perfect benchmarks performed a detailed analysis of the code restructuring techniques. The techniques include induction variable elimination, scalar expansion, forward substitution, strip mining, and loop interchange. The studies found that some of the programs showed improvements and that scalar expansion led to positive results. An effectiveness study on the Polaris compiler with OpenMP (Open Multi-Processing)[28] parallel code using Perfect benchmarks proclaimed a performance lag in small parallel loops. It also illustrated the importance of reduction operation, which resulted in a moderate (10%) performance improvement. Another study using an automatic parallelizing compiler to parallelize CFD (computational fluid dynamics) codes revealed that automatic code conversion along with user-inserted directives depicted considerable performance improvement.[29] A few works evaluated the overall performance of parallelizing compilers with improved loop timings.[30,31] PETRA, a portable performance evaluation tool, found that parallelizers are reasonably successful in about half of the given science-engineering programs. It also illustrated that some of the algorithms are data-dependence free, but the complex program constructs lead to a conservative analysis.[25]

Imai[32] claims that without any loop transformation, a program does not have much coarse-grain parallelism between different loops. Recent research has therefore proposed automatic tuning techniques to be essential parts of the auto-parallelization tools[33-35] with the aim to overcome performance degradation. The studies mentioned above do not evaluate the effect of loop transformation techniques, such as tiling, fission, fusion,

unrolling, and peeling on parallelization. Recent work has proposed loop transformation and optimization techniques to be the primary parts of parallelizers.[10,11] However, no detailed study on the support of different programming constructs by popular tools is currently available.

To the best of our knowledge, empirical evidence of these techniques and their effectiveness in parallelization of popular benchmarks is sorely missing. Such a study would be valuable to identify the relative pros and cons of various widely used auto-parallelization frameworks and would also help a user choose a specific framework in a given scenario to maximize the performance gains.

This work makes the following contributions.

- We provide a comprehensive performance evaluation of modern parallelizing compilers and their underlying parallelization techniques. This paper underlines that the key role of automatic parallelization of sequential programs combined with loop transformation techniques from the advent is to replace tedious and error-prone manual parallelization. In our study, we consider actively maintained five C+OpenMP-based auto-parallelization projects: Cetus, Par4all, Rose, ICC, and Pluto. We use two widely used benchmark suites: PolyBench and NAS parallel benchmarks (NPB). The suites consist of 38 programs that stress-test different aspects of the parallelization frameworks.
- It is necessary to investigate the ability of the tools in addressing loop parallelization issues. In particular, the presence of loop-carried dependence is a critical characteristic that limits the effectiveness of a parallelizer.[36] For a study such as ours, investigating the effect of loop-carried dependence and its removal by loop transformations are vital to understand the behavior and the sophistication of various auto-parallelizing compilers. This work also illustrates the role of different loop transformations that lead to better speedup.
- We collect empirical evidence for the effect of various transformations on PolyBench and NPB. In this work, we apply different source-to-source transformations for the identified predicaments confronted by the parallelizers and proposed possible solutions. We present the overall program performance and the effect of different techniques adopted by the frameworks in detecting execution overheads and overall speedup. During this process, we evaluate both the parallelization and the loop transformation techniques employed by different frameworks. We identify challenges faced by the auto-parallelizers and opportunities for potential improvements.

The remainder of this paper is structured as follows. Section 2 introduces each framework under study. Section 3 performs a qualitative comparison of the frameworks. Section 4 provides a brief description on parallelization of PolyBench and NPB benchmarks and their grouping based on different kinds of dependences. It also discusses the support by frameworks for different dependences occurring in benchmarks. In addition, it reveals the parallelization issues and their limitations along with the experimental results and discussion on the effect of loop transformation. We compare and contrast with the related work in Section 5 and conclude in Section 6.

2 | OVERVIEW OF AUTO-PARALLELIZER CHARACTERISTICS

In this section, we introduce the five parallelization frameworks in our study along with their individual methodologies.

2.1 | Cetus

Cetus is a source-to-source built-in C parser written in ANother Tool for Language Recognition (ANTLR) with intermediate representation (IR) classes and optimization passes. Cetus has a more memory-efficient IR, and it parses the source code quickly.[6] It has an array data-dependence testing framework that includes loop-based affine array-subscript testing. Cetus uses Banerjee-Wolfe inequalities[37,38] as the default dependence test, and it also supports the range test.

Cetus[4-6] performs analysis and transformation passes that include a set of general passes and parallelization passes. It performs the following checks to identify if a loop is parallelizable: (i) whether the loop is a canonical loop, (ii) whether function calls within the loop are without side effects, (iii) whether control-flow modifiers are not present within the loop body, and (iv) whether the loop increment is an integer constant. Cetus includes the following general passes and parallelization passes.

- Symbolic analysis that manipulates symbolic expressions, which is essential while dealing with real programs.
- Points-to alias and use-def chain analyses.
- Function inlining in place of complete interprocedural analysis and transformation.[5] However, parallel coverage tends to be less when using selective inlining.
- Several parallelization passes, such as induction variable recognition and substitution, reduction recognition and transformation, scalar and array privatization,[39-41] data dependence analysis,[41] and loop parallelizer and code generator,[4] are available in Cetus.

Not every loop should be parallelized. Cetus performs profitability tests for eliminating the smaller loops that are likely to cause overheads. The framework does not parallelize loops with a workload below a certain threshold. If the workload expression is not evaluated at compile time, the technique uses a runtime OpenMP IF construct technique, called the *model-based profitability test*. Cetus does not support nested parallelism and parallelizes only the outer loop. In addition, Cetus has an easy-to-use GUI that makes the parallelization process user friendly.

2.2 | Par4all

Par4all[8,9] is a source-to-source compiler that supports C and FORTRAN as the input language and can generate OpenMP, CUDA (Compute Unified Device Architecture),[42] or OpenCL (Open Computing Language)[43] code. It follows the PIPS framework (Parallelization Infrastructure for Parallel Systems).[44] PIPS and programmable pass manager together perform various program analyses, locate parallel loops, and generate parallel codes.[8] It delivers an advantage from its interprocedural capabilities, such as memory effects,[45,46] reduction detection, and parallelism detection, and from polyhedral analyses, such as convex array regions[47] and preconditions. Par4all exploits POCC (Polyhedral Compiler Collection)[48] for optimizing loops. It supports memory access transformations to improve locality.[8]

To obtain a good balance between performance and portability, Par4all relies on an intermediate representation of the input program. It allows increasing code portability, enabling intermediate code inspection and editing to simplify both debugging and low-level code optimizations. Par4all is very efficient for low-level optimizations and code generation purposes, primarily when it collects coding rules (eg, no pointers) and hints (eg, allocation of functions onto resources) that simplify its work to apply parallelization operations. Similar to Cetus, Par4all does not support nested parallelism. However, unlike Cetus, Par4all may choose to parallelize an inner loop depending upon an inbuilt heuristic cost function.

2.3 | Rose

Rose[14,15] is an open-source compiler framework to build source-to-source code transformations for C, C++, FORTRAN, OpenMP, and UPC (Unified Parallel C) applications. Rose consists of multiple front-ends, a mid-end operating on its internal intermediate representation (IR) and backends regenerating (unparsed) source code from IR. The IR encompasses an abstract syntax tree (AST), control flow graphs (CFG), and a symbol table with various interfaces for quickly building source-to-source translators. Rose includes several program analyses, including control-flow and data-flow analyses and data dependence and system dependence analyses. Rose has developed a wide range of transformations and optimizations via manipulating AST, including partial redundancy elimination, constant folding, inlining, outlining, automatic parallelization, and various loop transformations. Rose comprises the autoPar tool, which is an implementation of automatic parallelization using OpenMP. It can automatically insert OpenMP 3.0 directives into serial C/C++ codes. Rose can perform several loop transformations, such as fusion, fission, interchange, unrolling, and blocking privatization and reduction.

2.4 | ICC

ICC, a compiler from Intel,[16] is capable of performing powerful code transformations, including auto-parallelization. The code transformations and optimizations in ICC mainly include code restructuring and interprocedural optimizations, automatic parallelization, and vectorization. It comprises high-level and scalar optimizations, such as loop control and data transformations. Unlike other tools, ICC is not a source-to-source parallelizer. It examines the data flow of the code fragments and generates multithreaded code. ICC relies on an analytic engine for deciding when it is profitable to parallelize a loop. Through a user-defined threshold, ICC tries to distribute the overhead of creating multiple threads versus the amount of work at hand to be shared among the threads. ICC requires the number of iterations to be known before entry into a loop; if not, the compiler treats the loop to be computationally less intensive for parallelization. Also, the compiler performs data-flow analysis to ensure correct parallel execution.

2.5 | Pluto

The polyhedral model–based tool Pluto[10,11] is designed for programs that deal with linear algebra, linear programming, and high-level transformations. When the data access functions and loop bounds are affine combinations of the enclosing loop variables and parameters, then the polyhedral model is relevant for loop analysis and transformations. Pluto uses a scanner, a parser, and a dependence tester from the LooPo infrastructure, which is a polyhedral source-to-source transformer, including implementations of various polyhedral analyses and transformations from the literature. The portion of code to be parallelized is called the SCoP (Static Control Part).[49] It contains essential information to build a complete source-to-source framework in a polyhedral model.[50] The Clan tool[49,51] acts as a front-end and automatically translates an SCoP to an OpenScop,[50] which is a polyhedral, matrix-based representation. CLooG (Chunky Loop Generator)[52] works as the back-end compiler that improves code locality in the generated parallel code with several program transformation facilities and scheduling.[53] It creates the loop by scanning the Z-polyhedra output from Clan. Pluto uses ISL (Integer Set Library)[54] by default to compute dependences. Dependence analysis can be performed using Candl (Chunky ANalyzer for Dependences in Loops).[55] Pluto performs affine transformation analysis for efficient loop tiling and loop fusion along with other optimizations such as loop unrolling.

3 | QUALITATIVE STUDY OF AUTO-PARALLELIZATION FRAMEWORKS

In this section, we provide a qualitative comparison of various features supported by the frameworks under study. We first present a simple example parallelized by all the frameworks and then highlight differences in their support functionality.

(A)

```
for(i = 0; i < n; i++)
{
    A[i] = B[i] + C;
    D[i] = A[i] + E;
}
```

(B)

```
#pragma omp parallel for \\
if((10000<(1L+(4L*n)))) \\
private(i)
for (i=0; i<n; i ++ )
{
A[i]=(B[i]+C);
D[i]=(A[i]+E);
}
```

```
...
cmpq   $10000, %rax
setg   %dl
movq   -144(%rbp), %rax
movq   %rax, -64(%rbp)
movq   -128(%rbp), %rax
movq   %rax, -56(%rbp)
movl   %edx, %eax
xorl   $1, %eax
movzbl %al, %edx
leaq   -64(%rbp), %rax
movl   $0, %ecx
movq   %rax, %rsi
movl   $main._omp_fn.0, %edi
call   GOMP_parallel
...
```

(C)

```
#pragma omp parallel for
 for(i=0; i <= n-1; i+=1) {
   A[i] = B[i]+C;
   D[i] = A[i]+E;
 }
```

```
...
movq  %rax, -104(%rbp)
movq  -144(%rbp), %rax
movq  %rax, -64(%rbp)
movq  -128(%rbp), %rax
movq  %rax, -56(%rbp)
leaq  -64(%rbp), %rax
movl  $0, %ecx
movl  $0, %edx
movq  %rax, %rsi
movl $main._omp_fn.0, %edi
call GOMP_parallel
...
```

(D)

```
#pragma omp parallel for \\
private(i) firstprivate(n,C,E)
 for (i=0; i <= n-1; i+=1) {
    A[i] = B[i] + C;
    D[i] = A[i] + E;
 }
```

```
...
call    GOMP_parallel
movq    -64(%rbp), %rax
movq    %rax, -136(%rbp)
movq    -56(%rbp), %rax
movq    %rax, -128(%rbp)
...
```

(E)

```
LOOP BEGIN at sample.c(34,3)
<Peeled>
LOOP END
LOOP BEGIN at sample.c(34,3)
   remark #17109 : LOOP WAS AUTO-PARALLELIZED
   remark #17101 : parallel loop shared= { } private={ }
   firstprivate={ B C A E D i } lastprivate={ } \\
   firstlastprivate={ } reduction={ }
LOOP END
```

```
...
call    __kmpc_ok_to_fork
testl   %eax, %eax
je      ..B1.15       # Prob 50%
movl    $L_main_27__hpo_threaded_loop0_2.26, %edx
lea     (%rsp), %rcx
movl    $.2.3_2_hpo_loc_struct_pack.28, %edi
...
movq    $0, 8(%rsp)
call    __kmpc_fork_call
...
```

(F)

```
if (n >= 1) {
  lbp=0;
  ubp=n-1;
#pragma omp parallel for \\
private(lbv, ubv, t2)
  for (t1=lbp;t1<=ubp;t1++) {
    A[t1] = B[t1] + C;;
    D[t1] = A[t1] + E;;
  }
}
```

```
...
movl  %eax, -16(%rbp)
movq  -120(%rbp), %rax
movq  %rax, -192(%rbp)
movq  -112(%rbp), %rax
...
leaq  -192(%rbp), %rax
movl  $0, %edx
movq  %rax, %rsi
movl $main.omp_fn.0, %edi
call GOMP_parallel_start
...
```

FIGURE 1 Parallelization by various frameworks. A, Original code; B, Cetus; C, Par4all; D, Rose; E, ICC; F, Pluto

Figure 1A shows a sequential for-loop with loop-independent (intra-iteration) dependence. We illustrate how this small example is transformed by each of the frameworks, namely, Cetus (version 1.3.1), Par4all (1.4.6), Rose (0.9.9.47), ICC (15.0.0), and Pluto (0.11.4). Note that ICC does not output the transformed source code but works at the IR level. For each of the other transformed OpenMP codes, we also present a snippet of the corresponding Intel assembly code generated by gcc. Table 1 lists the compiler options used to obtain the parallelized code with the aid of various parallelizers.

As shown in Figure 1, all the tools can parallelize this loop and insert the OpenMP pragma for the for-loop. However, there are variations in the transformations even for this simple example. These variations stem from the techniques used by the tools, which we discuss below.

TABLE 1 Compiler options for auto-parallelization of sequential code

Parallelizer	Compiler Option to Obtain Parallelized Code
Cetus	via GUI
Par4all	`p4a <input>`
Rose	`autoPar -c <input>`
ICC	`icc -parallel -par-report:3 <input>`
Pluto	`polycc –parallel <input>`

Cetus: The transformed code from Cetus shown in Figure 1B contains an if condition, which is an artifact of a cost model that Cetus uses to identify if a loop is parallelizable (refer Section 2.1). This forbids parallelization of codes with very few iterations and accesses and compensates for the overhead of thread creation, etc. The assembly code shows the comparison with the constant value of 10 000, which is the default threshold cost.

Par4all: Unlike others, Par4all does not explicitly mark the loop index *i* to be a private variable. While this does not change the behavior, as by default, the loop indices are considered private, this shows that Par4all does not look for opportunities for privatization. This forces the runtime to conservatively assume variables to be shared, necessitating synchronization. It may lead to overall reduced performance. The assembly code listed in Figure 1C shows that the OpenMP compiler inserts a call to the GOMP/parallel function.

Rose: Rose transforms the code similar to others as depicted in Figure 1D, except for the usage of firstprivate, which marks variables *n*, *C*, and *E* as private, but copies their initial values from the outer scope (from outside the parallel scope). The assembly listing shows that there are several movq instructions *after* the call to GOMP/parallel, which is a deviation compared to other parallelizers.

ICC: ICC reports only the diagnostic information. However, this feature can be controlled using `-par-report:n`. When `n = 3`, it informs the auto-parallelizer to report diagnostic messages for loops successfully and unsuccessfully auto-parallelized, plus additional information about any proven or assumed dependences inhibiting auto-parallelization. Similar to Rose, ICC also uses the `firstprivate` clause. The assembly code for ICC looks quite different compared to that by other frameworks; this is primarily due to ICC working at a lower-level code form (IR) compared to other tools (source). It uses KMPC library routines to create threads (refer Figure 1E). Interestingly, it also inserts a comment (`# Prob 50%`) about branch prediction.

Pluto: Pluto does not use a cost model and, in fact, inserts a condition $n >= 1$, as shown in Figure 1F. Thus, it does not care for the thread creation overhead. However, the number of private variables used by Pluto is relatively large. The assembly code reveals that there are several movq instructions added for the additional private variables. Efficient execution of such a transformed code heavily relies on compiler optimizations for removing the unnecessary temporaries.

To summarize, different frameworks apply different transformations to the same code and differ in their abilities to use a cost model, or to identify thread-private variables, or to work at a lower level of program representation. Two mechanisms are involved in efficient loop parallelization: loop transformations[56] and dependence analysis,[57] which we discuss in the next two subsections.

3.1 | Loop transformation techniques

Table 2 illustrates common loop transformation techniques and their support in the parallelizers under study. *Privatization* creates a private copy of a shared variable, thereby removing data races.[58] Privatization reduces thread interference but needs to be applied carefully to ensure correctness. In *reduction variable recognition*, each thread is provided with a copy of the read/write conflicting variable to operate on and to produce a partial result, which is then combined with other threads' copies to produce a global result.[58] Threads need to cooperate with each other to enable parallel computation. Reduction is applicable only when the underlying computation follows certain algebraic properties such as associativity (in practice, tools also expect the computation to follow commutativity). Unlike other loop transformations, the parallelism in reduction iteratively reduces. Following are the loop optimization techniques that play a vital role in making effective use of parallel processing capabilities. *Loop fission* helps in attaining partial parallelization of the loop by splitting it into multiple loops over the same index range, with each taking only a part of the original loop's body.[37] The auto-parallelizing tools need to ensure that the original data dependences within and across iterations are preserved after fission. Loop fission

TABLE 2 Tool support matrix for loop transformation techniques (✓—parallelized (default), ✗—not parallelized, Priv.—privatization, Red.Var.—reduction variable recognition, ☑ —parallelized after getting enabled via the command line)

Tool	Priv. T1	Fission T2	Peeling T3	Fusion T4	Red.Var. T5	Tiling T6	Unrolling T7
Cetus	✓	✗	✗	✗	✓	✗	✗
ICC	✓	✓	✓	☑	✓	☑	☑
Par4all	✓	✗	✗	✓	✓	☑	☑
Pluto	✓	✗	✓	✓	✗	☑	☑
Rose	✓	☑	✗	☑	✓	☑	☑

can improve both spatial and temporal localities. *Loop peeling* is a transformation technique that helps in removing in-loop dependences, by splitting the dependent portion of the loop outside the parallel scope.[38,56,59] Peeling can help improve overall parallelization of the loop by separating its parallelizable and dependent portions. *Loop tiling* is another technique to improve temporal locality. It partitions a loop's iteration space into smaller chunks or blocks, to help ensure that the data used in a loop (tile) stay within a cache.[60] Tiling has been widely used by various auto-parallelizers for optimizing matrix-based codes. We observe from Table 2 that all the frameworks in our study support privatization of variables by default (T1). On the other hand, all the frameworks except Pluto inherently recognize reduction variables (T5). However, certain transformations (by default) are uniquely supported. For instance, ICC performs loop fission (T2). Some of the loop transformations are supported when we enable those via a command line. For example, tiling (T6) and unrolling (T7) are auto-performed via command-line arguments by all the frameworks except Cetus. On the whole, Pluto and Par4all support five of the seven transformations, Rose supports six (except loop peeling), and ICC supports all of them.

3.2 | Effect of transformations on dependences

In this subsection, we discuss how the frameworks handle various kinds of dependences. Amdahl's law[61] suggests that the sequential part of the program limits the amount of achievable parallelism. Typically, sequential execution is an artifact of data dependence, which forces an execution order. We categorize dependences as loop-independent[58,62] and loop-carried.[58,62] Each of these dependences could be scalar or vector (array-based).

Table 3 shows how the frameworks under study behave in parallelizing simple for-loops in the presence of intra-iteration and loop-carried scalar dependences. For each dependence-framework pair, we mention if the pair is supported (✓), and if so, what transformation technique (T1-T7) the framework uses for enabling the support. We observe that the tools primarily rely on privatization (T1), fusion (T4), and reduction variable recognition (T5). Privatization helps in handling all the dependence problems. Loop fusion[37] reduces the index variable maintenance overhead by merging two identical number of loop iterations to a single loop and is used by Par4all and Pluto. It also improves locality. All the frameworks, except Pluto, utilize reduction variable recognition, which removes loop-carried dependences with scalar variables. The reason for Pluto's poor support for scalar variables is because it is primarily targeted toward polyhedral transformations of array accesses, rather than individual shared items.

We now study the support of loop-carried dependence problems by the frameworks. Testing of loop-carried dependence problems encompasses loop-carried data, anti, and output dependences, along with their combinations. Figure 2 depicts these different scenarios, which can be used to test various frameworks by changing the distance vector $\alpha = 1, 2, 3$. The two cases mentioned in the Figure depict the forward and backward dependences and, accordingly, have different loop-index initialization and guard. Since programs with loop-carried dependences are not uncommon (more on real-world benchmarks in Section 4), we studied how various frameworks react to the presence of these dependences. Table 4 summarizes our observations. Overall, Cetus, Par4all, and Rose did not parallelize codes with loop-carried dependence. We note here that these three frameworks could support loop-carried dependence due to vector occurring in either the inner loop or the outer loop of a nested for (but not both the loops). These frameworks parallelize the dependence-free loop (inner/outer). Hence, we conclude the result as nonparallelized for the nested loops. On the other hand, Pluto and ICC support all the loop-carried dependence problems of Case 1 (backward dependence). However, ICC did not

TABLE 3 Effect on loop-independent and loop-carried scalar dependences (T1—privatization, T4—fusion, T5—reduction)

Dependence Problem	Cetus	Par4all	Pluto	Rose	ICC
Loop-independent (scalar)	✓T1	✓T1	✗	✓T1	✓T1
Loop-independent (vector)	✓T1	✓T1T4	✓T1T4	✓T1	✓T1
Loop-carried (scalar)	✓T1T5	✓T1T5	✗	✓T1T5	✓T1T5

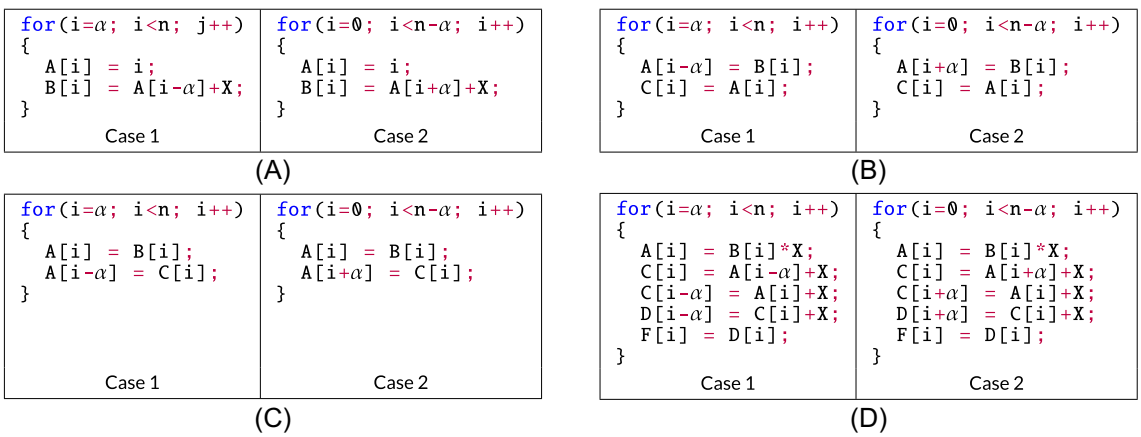


FIGURE 2 Templates of loop-carried dependence problems (Case 1 is backward dependence; Case 2 is forward dependence). A, Loop-carried data dependence; B, Loop-carried anti dependence; C, Loop-carried output dependence; D, Loop-carried data+anti+output dependence

TABLE 4 Support of auto-parallelizer frameworks for different dependence problems using synthetic programs (✓—parallelized, ✗—not parallelized, T1—privatization, T2—loop fission, T3—loop peeling)

	Tools	Cetus			Par4all			Rose			ICC			Pluto		
Dependence Problems	**Distance Vectors**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**
Loop-carried data	Case 1	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T3	✓ T1 T3	✓ T1 T3
	Case 2	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T3	✓ T1 T3	✓ T1 T3
Loop-carried anti	Case 1	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T3	✓ T1 T3	✓ T1 T3
	Case 2	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T3	✓ T1 T3	✓ T1 T3
Loop-carried output	Case 1	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T3	✓ T1 T3	✓ T1 T3
	Case 2	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T3	✓ T1 T3	✓ T1 T3
Loop-carried data+anti+output	Case 1	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T3	✓ T1 T3	✓ T1 T3
	Case 2	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✗	✗	✗

parallelize loop-carried output forward dependence (Case 2), whereas Pluto did not parallelize loop-carried data+anti+output forward dependence. Both these frameworks utilize privatization (T1) and loop peeling (T3) for handling the dependence problems. ICC additionally makes use of loop fission (T2) along with other optimizations (see Table 2).

We now study the behavior of the frameworks on programs having nested loops with loop-carried dependence. Figure 3 depicts six such cases: backward i, backward j, backward i and j, and their forward counterparts. These simple templates suffice to assess the versatility of each framework. Table 5 summarizes our findings, with distance vector $\alpha = 1, 2, 3$. We observe that Pluto supports all these cases of loop-carried dependence problems. This is an artifact of the powerful polyhedral model underlying Pluto's transformations. As we discuss in the next section, Pluto does perform well on PolyBench benchmarks. On the other hand, ICC supports all the cases of loop-carried dependence only when $\alpha = 1$. When $\alpha = 2, 3$, it could handle Cases 2 and 5 only (backward and forward dependences in the *inner* loop). Thus, when the dependence occurs due to the *outer* loop of the nested for when $\alpha = 2, 3$, ICC failed to parallelize. This is interesting because it indicates that the parallelization was tuned for distance vector of unity.* In the transformation, Pluto and ICC use privatization (T1) and loop peeling (T3) for handling the dependence. ICC additionally makes use of loop fission (T2) along with other optimizations for Cases 2 and 5, that is, when the dependence occurs due to the inner loop of the nested for.

4 | QUANTITATIVE STUDY OF AUTO-PARALLELIZATION FRAMEWORKS

Using our carefully crafted synthetic programs, we have identified various scenarios under which the frameworks are able to auto-parallelize. In this section, we investigate the frameworks for the performance of their transformed codes. We use two popular benchmark suites: PolyBench[63,64] and NAS parallel benchmarks (NPB).[65-68] PolyBench is a suite of linear algebra kernels targeted for auto-parallelizers based on the polyhedral model. NPB is a widely used suite of computational fluid dynamics (CFD) applications for assessing the effectiveness of large-scale parallel executions. We carried out our study as a three-step process.

*We believe that ICC supports the unit length distance vector alone as that case was found to be common in practice.

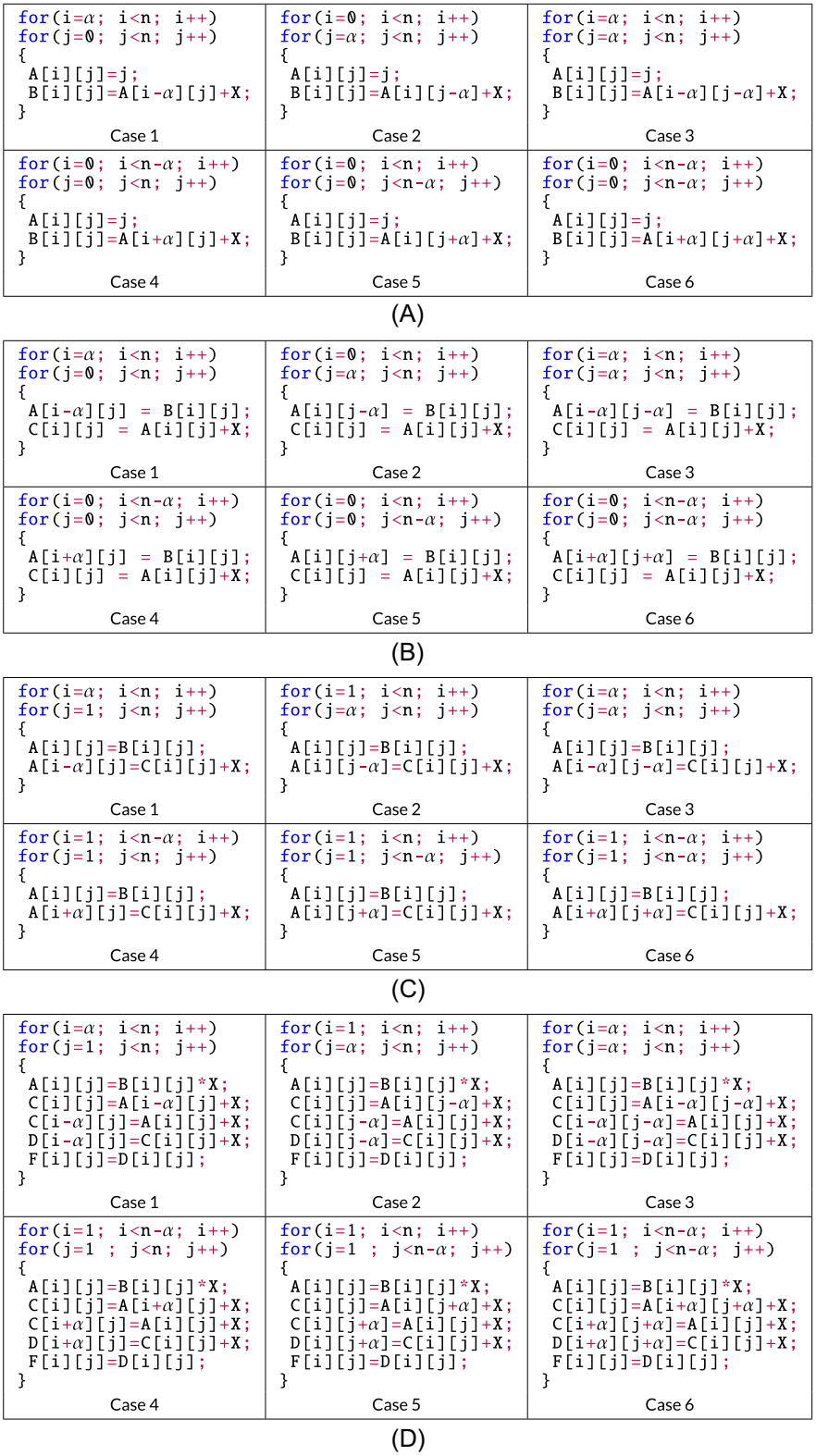


FIGURE 3 Templates of loop-carried dependence problems in nested loops. The six cases are backward i, backward j, backward i and j, forward i, forward j, and forward i and j. A, Loop-carried data dependence problems; B, Loop-carried anti dependence problems; C, Loop-carried output dependence problems; D, Loop-carried data+anti+output dependence problems

- We parallelized each PolyBench and NPB benchmark (38 in total) using each of the five frameworks and obtained the support matrix indicating which benchmark could be successfully parallelized by each framework.
- We analyzed each nonparallelized benchmark and investigated the reasons behind it. We report those later in this section.
- To assess if the transformation was effective, we executed the original sequential program, and the corresponding transformed parallel program, to find the relative speedup. We then analyzed the results, which we discuss below.

TABLE 5 Support of auto-parallelizer frameworks for different dependence problems in nested loops (✓—parallelized, ✗—not parallelized, T1—privatization, T2—loop fission, T3—loop peeling)

Tools	Cetus			Par4all			Rose			ICC			Pluto		
Distance Vector	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
Case 1	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T3	✗	✗	✓ T1 T3	✓ T1 T3	✓ T1 T3
Case 2	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T3	✓ T1 T3	✓ T1 T3
Case 3	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T3	✗	✗	✓ T1 T3	✓ T1 T3	✓ T1 T3
Case 4	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T3	✗	✗	✓ T1 T3	✓ T1 T3	✓ T1 T3
Case 5	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T2 T3	✓ T1 T3	✓ T1 T3	✓ T1 T3
Case 6	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓ T1 T3	✗	✗	✓ T1 T3	✓ T1 T3	✓ T1 T3

All the experiments are carried out on an Intel Xeon E5-2650 v2 machine with 32 cores clocked at 2.6 GHz with 100 GB of RAM, 32 KB of L1 data cache, 256 KB of L2 cache, and 20 MB of L3 cache. The machine runs CentOS 6.5 and 2.6.32-431 kernel, with GCC version 4.4.7, ICC version 15.0.0, and OpenMP version 4.0. Various command-line parameters used to invoke the parallelization frameworks are shown in Table 1. We vary the number of threads as 64, 32, and 16 and compare the speedup against that of the sequential version. The achieved speedup depends upon several factors, such as the amount of total work, load imbalance across threads, overhead of thread creation and synchronization, etc, and is often smaller than the ideal speedup (K× with K threads). To have uniformity in the configuration concerning other frameworks, we retained the default behavior of all the frameworks during evaluation.

4.1 | Auto-parallelization of PolyBench

We executed the PolyBench/C 4.2 benchmarks via the five frameworks under study. PolyBench/C is a suite of 30 programs with numerical computations from different application domains, such as linear algebra, image processing, physics simulation, dynamic programming, and statistics. PolyBench contains several nested loop structures with complex dependences. It makes the kernels challenging to be auto-analyzed by the tools.

We observed that out of the 30 benchmarks, the sequential execution times of two benchmarks (`gesummv` and `deriche`) were negligible; hence, we consider the remaining 28 compute-intensive programs for our experimentation. Table 6 lists which frameworks were able to parallelize which PolyBench codes successfully. Overall, we observe that none of the frameworks parallelized all 28 programs. Pluto could parallelize the largest number (25) of benchmarks. Par4all and Rose could successfully parallelize the same set of 22 benchmarks each, whereas ICC parallelized 18 of them. Pluto could parallelize all the benchmarks with loop-carried dependence due to the vector, and other frameworks could support a subset of them, but they do not work when complex dependences are present. There is no PolyBench benchmark that could not be parallelized using any of these frameworks. This indicates that the constructs and the dependences appearing in PolyBench benchmarks are well covered across the auto-parallelization frameworks. As a corollary, it is feasible to design a meta-parallelizer to improve parallelization coverage, which uses techniques from various existing auto-parallelizers.

We carefully analyzed the scenarios that forbid the frameworks from parallelizing certain benchmarks. Cetus, Par4all, and Rose failed to parallelize benchmarks with loop-carried dependences. Cetus relies on function inlining for improving the scope for intraprocedural optimizations.[5] However, Cetus does not inline functions by default. Hence, it failed in parallelizing some of the loop-independent dependence problems. Clearly, this is a serious limitation of the frameworks (with default behavior). ICC reported the existence of parallel dependence but insufficient computational work. Pluto worked with vectors but did not parallelize for-loops when the ℓ-value of the statement is a scalar variable. It failed to parallelize the benchmarks with loop-carried dependence due to scalar.[†]

[†]We also encountered some implementation issues, which can be fixed by the framework developers. For instance, Cetus does not support multiple "\" (line continuation character) in a single preprocessor macro definition. As a fix, we split such definitions into multiple macro definitions with a single "\" character. Similarly, in Rose, *conditional* preprocessor macros are not correctly interpreted.

TABLE 6 Characteristics of PolyBench benchmarks and the parallelization results by various frameworks (DM = data mining, LA = linear algebra, ✓—parallelized, ✗—not parallelized)

Benchmarks	Domain	Stmts	Dep	RAW	WAW	WAR	Problem Size	Seq. Exec. Time T_S	Cetus	Par4all	Rose	ICC	Pluto
correlation	DM	22	77	42	18	17	E4	4219.65	✗[b]	✓	✓	✓	✓
covariance	DM	15	34	16	11	7	E4	4220.66	✗[b]	✓	✓	✓	✓
gemm	LA	5	6	2	2	2	E4	3620.39	✓	✓	✓	✓	✓
gemver	LA	7	13	7	3	3	E5	403.94	✗[b]	✓	✓	✓	✓
syr2k	LA	5	6	2	2	2	E4	6482.79	✓	✓	✓	✓	✓
syrk	LA	5	6	2	2	2	E4	4060.29	✓	✓	✓	✓	✓
syrk	LA	5	6	2	2	2	E4	4060.29	✓	✓	✓	✓	✓
2 mm	LA	10	13	6	4	3	E4	20 081.44	✗[b]	✓	✓	✓	✓
3 mm	LA	15	19	10	6	3	E4	24 170.55	✗[b]	✓	✓	✓	✓
atax	LA	6	12	6	4	2	E5	64.49	✓	✓	✓	✓	✓
bicg	LA	6	10	4	4	2	E5	57.93	✓	✓	✓	✗[cd]	✓
doitgen	LA	10	30	10	10	10	E3	4204.52	✗[b]	✓	✓	✗[cd]	✓
mvt	LA	4	6	2	2	2	E5	307.31	✗[b]	✓	✓	✓	✓
symm	LA	10	33	11	11	11	E4	6338.77	✗[b]	✓	✓	✓	✗[e]
durbin	LA	10	55	27	13	15	E6	4205.76	✓	✓	✓	✗[cd]	✗[e]
gramschmidt	LA	14	34	17	8	9	E4	7476.37	✓	✓	✓	✓	✓
ludcmp	LA	22	181	66	56	59	E4	1839.32	✓	✓	✓	✗[cd]	✗[e]
trmm	LA	5	8	2	2	4	E4	4194.12	✗[b]	✓	✓	✓	✓
adi	Stencils	34	140	68	22	50	E4	43 285.76	✗[b]	✓	✓	✓	✓
fdtd-2d	Stencils	11	28	12	4	12	E4	14 255.44	✗[b]	✓	✓	✓	✓
heat-3d	Stencils	8	42	20	2	20	E4	27 887.57	✗[b]	✓	✓	✓	✓
jacobi-1d	Stencils	4	14	6	2	6	E6	7052.42	✗[b]	✓	✓	✓	✓
jacobi-2d	Stencils	6	22	10	2	10	E4	14 871.71	✗[b]	✓	✓	✓	✓
cholesky	LA	8	22	14	4	4	E4	571.44	✗[b]	✗[a]	✗[a]	✗[cd]	✓
lu	LA	8	16	10	3	3	E4	2603.65	✗[a]	✗[a]	✗[a]	✗[cd]	✓
trisolv	LA	4	13	7	4	2	E5	15.61	✗[ab]	✗[a]	✗[a]	✗[cd]	✓
floyd-warshall	Medley	3	21	10	1	10	E4	3484.39	✗[a]	✗[a]	✗[a]	✗[d]	✓
nussinov	Medley	11	52	19	9	24	E4	1157.45	✗[a]	✗[a]	✗[a]	✗[d]	✓
seidel-2d	Stencils	3	27	13	1	13	E4	13 019.95	✗[a]	✗[a]	✗[a]	✗[d]	✓
28	–	271	940	421	212	307	–	224 153.69	8	22	22	18	25

Loop-independent dependence problems
Loop-carried dependence problems
[a] Loop-carried dependence problems.
[b] If we apply inlining+parallelization, these benchmarks can be parallelized (lack of interprocedural optimization).
[c] Insufficient computational work.
[d] Existence of *parallel dependence*.
[e] Pluto fails in parallelizing loops comprising scalar variables.

TABLE 7 Performance summary (with 16, 32, and 64 threads)

No. of Threads	Suite	Cetus	Par4all	Rose	ICC	Pluto
16	PolyBench	1.0	5.5	5.0	12.5	5.9
	NPB	0.9	1.4	1.3	2.7	0.8
32	PolyBench	1.0	7.0	6.4	11.1	8.0
	NPB	0.9	1.2	1.1	2.0	0.7
64	PolyBench	1.0	5.7	5.0	10.2	5.2
	NPB	0.9	1.1	1.4	2.6	0.8

4.1.1 | Experimental results

We now discuss the quantitative effect of each framework on various PolyBench benchmarks. Table 6 shows static characteristics of the benchmarks, such as the application domain, the number of potentially parallelizable statements (Stmts), and the number of static dependences (Dep) of the types Read-After-Write (RAW), Write-After-Write (WAW), and Write-After-Read (WAR). We use the Candl tool[55] to obtain these characteristics. The eighth and ninth columns of the Table also list the problem size used for executing the benchmark and the sequential execution time in seconds for each benchmark. We have grouped the benchmarks based on loop-independent (dark background) and loop-carried dependences (light background). We see that 12 out of the 28 benchmarks have loop-independent computations and are likely to scale well with the number of threads. On the other hand, the remaining 16 benchmarks exhibit loop-carried dependence, which affects their scalability.

Table 7 summarizes the overall results. It lists the average speedup achieved by all the auto-parallelization frameworks under study on PolyBench and NPB benchmarks. We observe that it is difficult to find parallelization opportunities in NPB (speedups are relatively smaller compared to those in the case of PolyBench). A primary reason behind this is that the NPB codes are already quite optimized.[25]

We now discuss the parallelization results in more detail. Figure 4 shows the speedup obtained by the frameworks on each benchmark. We discuss performance by categorizing benchmarks based on the absence or presence of loop-carried dependence.

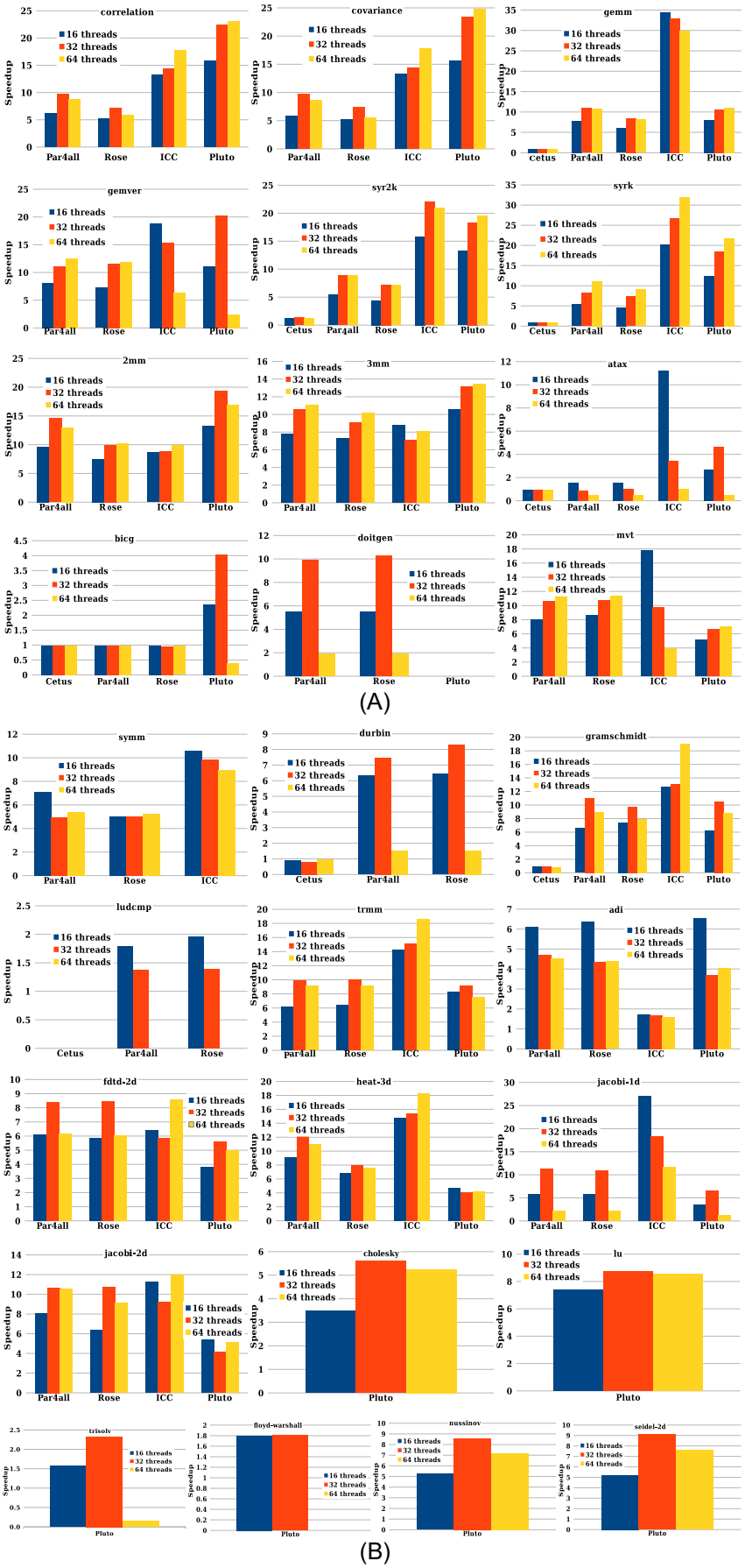


FIGURE 4 Quantitative analysis of the PolyBench benchmark. A, Benchmarks with loop-independent dependences; B, Benchmarks with loop-carried dependences due to scalar/vector

Benchmarks with loop-independent dependence

Table 6 shows the benchmarks grouped as loop-independent dependence problems. We observe from the support matrix that all the frameworks support benchmarks with loop-independent dependence with the aid of a privatization mechanism. At the same time, Cetus and ICC fail on several of these benchmarks. This happens because Cetus requires function inlining for improving the scope for intraprocedural optimizations (refer to Section 2.1), whereas ICC reports the existence of parallel dependence. ICC also reports insufficient computational work for two of the benchmarks (`bicg` and `doitgen`).

We now discuss the performance impact of frameworks in parallelizing the loop-independent dependence problems shown in Figure 4.[‡] We observe that, overall, Pluto and ICC exhibit relatively better speedups compared to others. ICC achieves the best overall speedup (with 32 threads: `gemm` 33.0×, `syr2k` 22.0×, and `syrk` 26.8×; with one thread[§]: `gemm` 5.1×, `syr2k` 2.3×, and `syrk` 2.8×). This is primarily due to vectorization and privatization. Pluto performs quite well (with 32 threads: `correlation` 22.4×, `covariance` 23.5×, `gemver` 20.2×, `2mm` 19.4×, `3mm` 13.1×, `atax` 4.6×, and `bicg` 2.4×). The best speedup is achieved due to the effect of polyhedral optimization and privatization. However, `atax` and `bicg` exhibit limited speedup as the parallel coverage is less compute intensive. Also, Pluto achieves lesser speedup for four benchmarks (`gemm`, `syr2k`, `syrk`, and `mvt`). Among Cetus, Par4all, and Rose, we observe that Par4all performs better than Rose (with 32 threads: `correlation` and `covariance` 9.7×, `gemm` 10.9×, `syr2k` 8.9×, `syrk` 8.2×, `2mm` 14.6×, and `3mm` 10.6×). Besides, Rose exhibits good results (with 32 threads: `gemver` 11.5×, `doitgen` 10.3×, and `mvt` 10.7×). Cetus fares overall inferior to other frameworks obtaining average performance equivalent to the sequential run. The effectiveness of the frameworks depends upon their techniques and methodologies. For instance, all the five frameworks apply array privatization. However, Cetus, Par4all, and Rose do not perform any additional optimization. On the other hand, ICC relies heavily on vectorization, whereas Pluto exploits polyhedral optimization. Although ICC and Pluto perform overall better than others, the performance improvements by Par4all and Rose suggest that array privatization provides considerable performance benefits.

Benchmarks with loop-carried dependences

The benchmarks shown in Table 6 exhibit three types of loop-carried dependence.

- Case 1: Loop-carried dependence due to scalar variable(s). Such benchmarks are potentially amenable to hierarchical reduction. This is exhibited by four benchmarks—`symm, durbin, gramschmidt, and ludcmp`.
- Case 2: Loop-carried dependence due to vector variable(s) in the inner/outer loop of a nested for. Such benchmarks are potentially amenable to simple loop transformations and array privatization. This is exhibited by nine benchmarks—`symm, durbin, ludcmp, trmm, adi, fdtd-2d, heat-3d, jacobi-1d, and jacobi-2d`. Note that a benchmark may have multiple kinds of loop-carried dependence, based on different variables. Out of these nine, only `adi` contains dependence in the inner loop; all others contain dependence in the outer loop.
- Case 3: Complex loop-carried dependence. Such benchmarks are potentially amenable to sophisticated loop analysis such as polyhedral loop transformations. Due to the complex dependence pattern, such benchmarks require considerable loop transformations or manual code changes for achieving parallelization. This is exhibited by six benchmarks—`cholesky, lu, trisolv, floyd-warshall, nussinov, and seidel-2d`.

Table 6 shows the support matrix of benchmarks comprising loop-carried dependence problems for the above three cases. Our investigation of the benchmarks in Case 1 reveals that all the frameworks except Pluto apply reduction variable recognition optimization[52] besides parallelization and array privatization. Since Pluto is a polyhedral tool and performs affine transformation based on array-index expressions,[11] it does not parallelize loops when the ℓ-value of a statement is a scalar.

For Case 2, the frameworks parallelize the dependence-free loop (inner/outer) of the nested for-loop as described in Section 3.2. Via our microbenchmarks, we make the following observations. Par4all and Rose parallelize all the benchmarks. However, ICC reports the existence of parallel dependence and insufficient computational work for two of the benchmarks (`durbin` and `ludcmp`). Cetus supported the least number of benchmarks in our setup (`durbin`, `gramschmidt`, and `ludcmp`) as it does not, by default, support interprocedural optimization.[¶]

For Case 3, ie, the benchmarks with complex loop-carried dependence are parallelized only by Pluto. These benchmarks do not have dependence-free loops; hence, the scope for parallelization is less. It requires additional loop analysis and transformation to remove the dependences. Pluto applies loop peeling in addition to parallelization, privatization, and polyhedral optimization to achieve this. ICC reports the existence of parallel dependence and insufficient computational work for all the Case 3 benchmarks and, therefore, does not parallelize these loops.

We now discuss the performance of the five frameworks in parallelizing the loop-carried dependence problems depicted in Figure 4.[‡] Overall, for Case 2 benchmarks, ICC performs better than all the other frameworks (with 32 threads: `symm` 9.8×, `gramschmidt` 13.1×, `trmm` 15.1×, `heat-3d` 15.4×, and `jacobi-1d` 18.4×; with one thread[§]: `symm` 2.0×, `gramschmidt` 1.8×, `trmm` 2.0×, `heat-3d` 5.5×, and `jacobi-1d` 3.9×). This performance stems from ICC applying additional transformations such as loop peeling along with array privatization, parallelization, and vectorization. Note that the single-threaded performance of ICC also achieves good speedup over the sequential baseline as ICC enables -O2 optimizations by

[‡] Benchmarks with 0 speedup indicate a timeout of 1 hour.
[§] Since ICC's baseline involves default optimization (-O2), we also provide single-thread results. This is to show that ICC exploits better overall parallelism.
[¶] Using function inlining, Cetus can improve its parallelization effect (refer to Section 2.1). However, to have uniformity in the configuration concerning other frameworks, we retained the default behavior of Cetus and did not explicitly enable function inlining.

default. Among Cetus, Par4all, Rose, and Pluto frameworks, Par4all and Rose provide better benefits (Par4all with 32 threads: `gramschmidt` 11.0×, `adi` 4.7×, `heat-3d` 12.3×, and `jacobi-1d` 11.3×; Rose with 32 threads: `symm` 5.0×, `durbin` 8.3×, `trmm` 10.0×, and `fdtd-2d` 8.5×; Par4all and Rose with 32 threads: `jacobi-2d` 10.7×). This indicates that, compared to polyhedral transformations, array privatization provides better performance improvement on PolyBench. Parallelization of `adi` shows that additional optimization (namely, loop fission) by ICC affects the scalability. For Case 3 benchmarks with complex dependences, only Pluto could parallelize the codes, but the performance is relatively lower (with 32 threads: `cholesky` 5.6×, `lu` 8.8×, `trisolv` 2.3×, `floyd-warshall` 1.8×, `nussinov` 8.6×, and `seidel-2d` 9.1×). The average speedup with 32 threads for each framework is listed in Table 7.

4.1.2 | Effect of static dependences

The fourth column of Table 6 (named Dep) shows the number of static dependences present in each benchmark. To assess how it affects performance, we bucketize the range of Dep values, namely, 0-10, 11-20, 21-30, 31-40, and >40, and study the performance of benchmarks falling into a range. The high-level expectation is that the more the dependence, the smaller the parallelization benefit (however, note that the overall speedup gets affected by other factors also). Columns Stmts and RAW in Table 6 are also indicative. However, the two statistics follow the same trend as Dep. Hence, the below discussion applies also to the two statistics.

Figure 5 depicts the variation in speedup between the individual Dep range of all the five frameworks. We observe that for a large number of dependences, the speedup is usually relatively smaller. However, as expected, the speedup does not always reduce with increasing number of static dependences. Therefore, the number of static dependences alone cannot be conclusively used to predict the parallel performance.

Second, the maximum speedup is achieved by ICC (up to 34.5× for Dep between 0 and 10). Besides, on an average, the best speedup is achieved by ICC (average of 11.1×). Pluto shows a considerable difference in performance across static dependence ranges. On the other hand, Cetus, Par4all, and Rose are relatively less sensitive to this value. This clearly indicates that the dependences are modeled more prominently in ICC and Pluto transformations. Among Cetus, Par4all, and Rose frameworks, Cetus neither transforms many benchmarks nor achieves good parallelism on the ones that it successfully transforms. The performance benefits of Par4all and Rose are overall comparable but are significantly lesser than those of ICC and Pluto.

4.2 | Auto-parallelization of NPB

We executed the NAS parallel benchmarks (NPB3.3-SER-C) through the five frameworks under study. NPB has been widely used in the literature for the evaluation of parallelization proposals. NPB is a suite of 10 programs, and the benchmarks are primarily derived from the domain of computational fluid dynamics (CFD). It includes applications dealing with unstructured adaptive mesh, parallel I/O, multizone applications, and computational grids. Drozdov et al[69] tested different optimization techniques in the LLVM compiler using NPB and compared the performance of different compilers. Griebler et al[68] provided an efficient C++ version of NAS benchmark kernels by replacing the FORTRAN code. Our study focuses on the parallelization of NPB-C benchmarks using auto-parallelizers. Unlike PolyBench, each NPB benchmark comprises different dependence structures, including loop-independent and loop-carried dependences due to a scalar/vector. Also, it has several functions involving computationally intensive loops with complex coding style (which matters for source-to-source translators). We used all 10 NPB benchmarks for our study.

Table 8 presents the effectiveness of the frameworks in auto-parallelizing the benchmarks. Successful parallelization here symbolizes two aspects: (i) code transformation happened, and (ii) the transformed code is equivalent to the serial version in terms of its output. The correctness check is programmatically performed by the test case of the individual benchmark. We observe that most of the benchmarks were successfully parallelized (✓). Semisuccessful parallelization indicates that either the code transformation or the semantic equivalence was not achieved. The former occurs due to pre-transformation issues such as timeout and syntactical errors during conversion. The latter is due to post-transformation issues that result in erroneous code transformation. We also tried rectifying the issues manually in the semisuccessful codes. The codes where we could address the pre-transformation and post-transformation issues are marked as ☑ in the table. We could successfully parallelize a few semisuccessful benchmarks using Par4all, Rose, and Pluto after this manual intervention (eg, `DC` by Par4all), denoted as α for pre-transformation and as β for post-transformation (discussed more later in this section). After applying all the changes, Par4all could parallelize all 10 NPB benchmarks. ICC and Pluto were a close second, parallelizing nine of them. Cetus could successfully parallelize eight, whereas Rose parallelized seven of them.

Despite the manual fixes, a few of the benchmarks under study were unsuccessful and are represented in the table using the symbol ☒. Pluto and Rose produced semantically incorrect output for `UA`, but the source of the error could not be found. Cetus and Rose failed to parallelize `FT` and `IS`, respectively, due to timeout.[#] Rose could not parallelize `FT` due to an error during code transformation. There are nonparallelized benchmarks exhibited by a few of the tools and are shown in the table using the symbol ✗. ICC did not parallelize `DC` and reported that some of the loops are not parallelization candidates because there exists either dependence across iterations or insufficient computational work. On the other hand, Cetus failed to parallelize `BT` due to the lack of support for interprocedural analysis (see Section 2.1). Interestingly, `CG` is the only benchmark successfully transformed by all the frameworks without any changes. This observation poses a serious question mark over the expressivity and generality of auto-parallelization frameworks, which is one of the takeaways of our study.

[#] Timeout indicates codes taking more than an hour for program transformation.

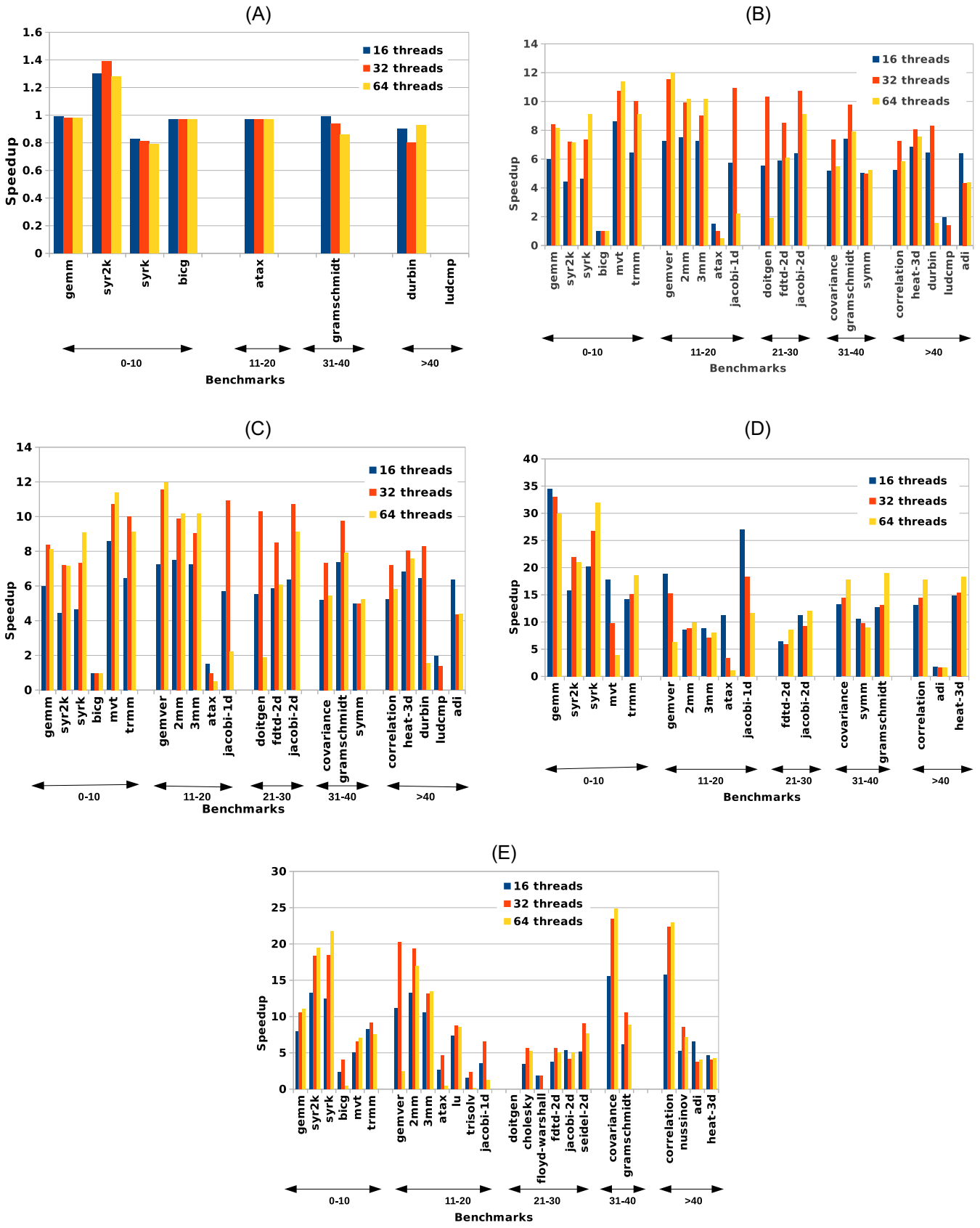


FIGURE 5 Speedup analysis of the PolyBench benchmark (note that the *y*-axes use different scales across plots). A, Cetus; B, Par4all; C, Rose; D, ICC; E, Pluto

TABLE 8 Complexity measurement of NPB using Candl and the parallelization results by various frameworks (Bm.—benchmark, ✗—nonparallelized, ☒—unsuccessful, ✓—successful, ☑—successful after subjecting to changes, α—changes for pre-transformation issues, β—changes for post-transformation issues; for BT and LU, Candl invoked by Pluto timed out, and hence, some statistics are not displayed)

Bm.	Description	Stmts	Dep	RAW	WAW	WAR	CLASS	Seq. Exec. Time T_S (S)	Cetus	Par4all	Rose	ICC	Pluto
BT	Block Tri-diagonal solver	419	2412	–	–	–	C	2591.08	✗[h]	✓	β^{d}☑	✓	$\alpha^{a}\beta^{g}$☑
CG	Conjugate Gradient, irregular memory access and communication	11	47	18	18	11	C	387.17	✓	✓	✓	✓	✓
DC	Data Cube	5	3	2	1	0	B	911.11	✓	α^{b} β^{c} ☑	$\alpha^{j}\beta^{c}$☑	✗[i]	β^{c}☑
EP	Embarrassingly Parallel	13	58	26	20	12	C	616.47	✓	✓	β^{f}☑	✓	β^{c}☑
FT	Discrete 3D fast Fourier Transform, all-to-all communication	9	5	2	2	1	C	668.01	☒[a]	✓	☒[j]	✓	β^{g}☑
IS	Integer Sort, random memory access	4	12	4	4	4	C	20.17	✓	✓	☒[a]	✓	✓
LU	Lower-Upper Gauss-Seidel solver	1282	15 708	–	–	–	C	2931.64	✓	✓	$\alpha^{j}\beta^{c}$☑	✓	$\alpha^{a}\beta^{g}$☑
MG	Multi-Grid on a sequence of meshes, long and short distance communication, memory intensive	11	47	18	18	11	C	256.44	✓	✓	✓	✓	β^{g}☑
SP	Scalar Penta-diagonal solver	230	884	305	263	316	C	2568.85	✓	✓	β^{d}☑	✓	β^{g}☑
UA	Unstructured Adaptive mesh, dynamic and irregular memory access	19	20	8	6	6	C	2081.15	✓	β^{c}☑	☒[eg]	✓	☒[eg]
10	–	2003	19 196	383	332	370	–	13 032.09	8	10	7	9	9

[a]Timeout after an hour.
[b]Due to recursion.
[c]Erroneous code transformation.
[d]Privatization error.
[e]Parallelized code output is not equivalent to serial code.
[f]Reduction error.
[g]Error after code transformation and the source of the error could not be deduced.
[h]Lack of support for interprocedural optimizations.
[i]Loops are not parallelization candidates, and there exist dependence across iterations and insufficient computational work.
[j]Error during code transformation and the source of the error could not be deduced.

4.2.1 | Details of the transformation errors: manual changes on pre- and post-transformation issues

First, we discuss about two pre-transformation issues (α), namely, (i) timeout and (ii) error during code transformation in Table 8, and our manual changes to fix those issues.

(i) **Timeout**: For `BT`, Pluto timed out[#] for the files *x_solve*, *y_solve*, and *z_solve*. Similarly, Pluto timed out in transforming the file *erhs* of `LU`. This is due to more number of statements within the parallelizable region.

(ii) **Error during code transformation**: For `DC`, auto-parallelization of files *jobcntl* and *adc* by Rose was unsuccessful, and the source of the error could not be deduced. Likewise, Rose produced an error during transformation of the file *ssor* of `LU`. Hence, for the timeout and the error cases, we use the original sequential code for the evaluation. For `DC`, Par4all fails to transform due to recursive functions *WriteViewToDisk()*, *WriteViewToDiskCS()*, *computeChecksum()*, and *WriteChunkToDisk()*. Hence, we eliminated the recursive functions during transformation. Figure 6A shows the snippet of the recursive code from `DC`.

Next, we illustrate the applied changes due to the post-transformation problems (β), namely, (i) illegal transformation, (ii) incorrect transformation, and (iii) unrecognized problems. These are listed in the last five columns of Table 8.

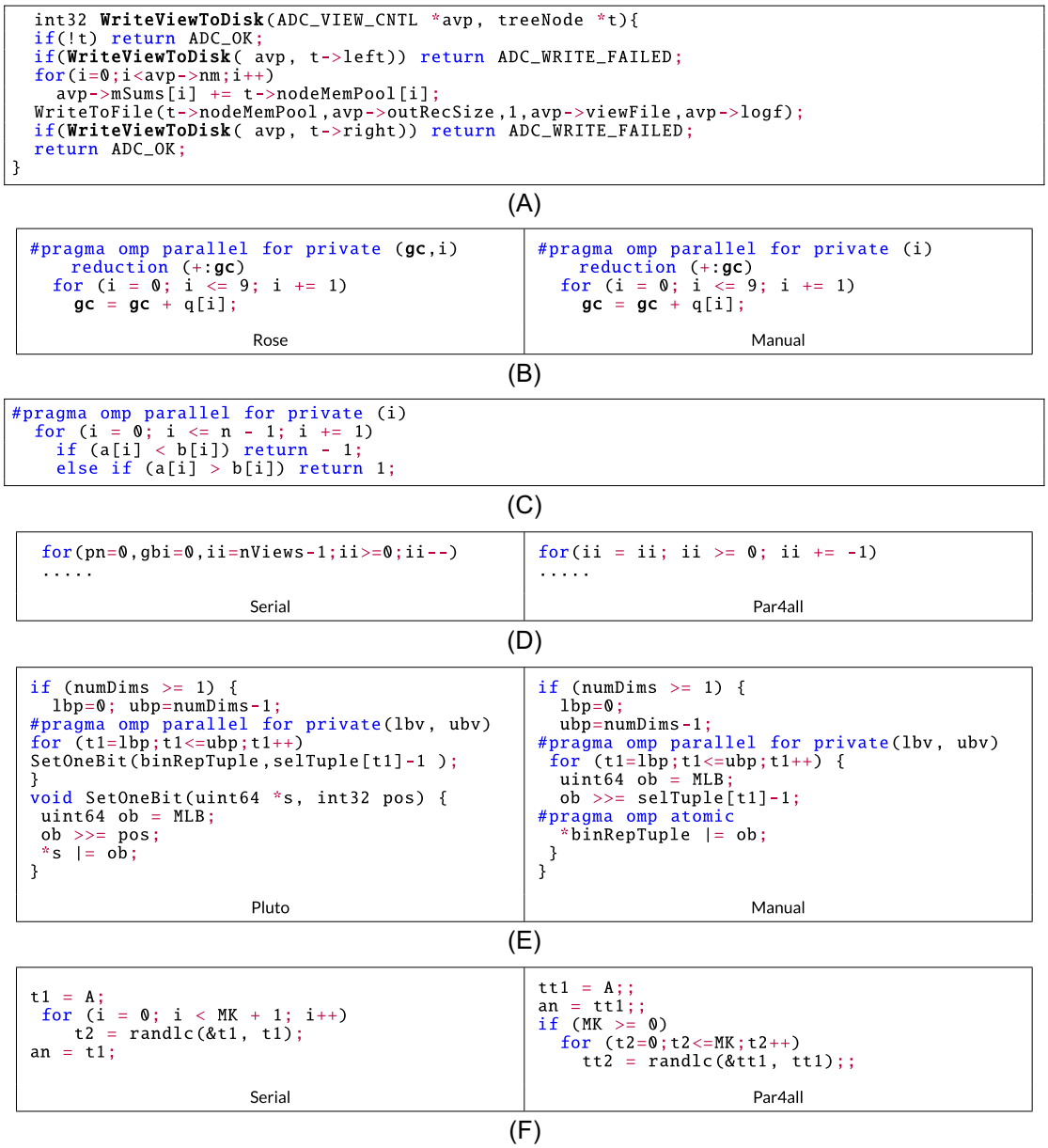


FIGURE 6 Pre- and post-transformation issues. A, Par4all fails to parallelize DC due to recursive *WriteViewToDisk()*; B, Invalid privatization + reduction transformation by Rose (left) and valid manual transformation (right); C, Rose produces illegal transformation (return within parallel pragma); D, Serial DC code and incorrect for-loop initialization by Par4all; E, Erroneous code transformation by Pluto (left) and correct manual transformation (right); F, Serial EP code (left) and incorrect code transformation by Pluto

(i) Illegal transformation: For BT and SP, Rose-transformed code resulted in a syntax error in function *compute_rhs()* as the tool applies privatization twice to the same index variable. For evaluation, we remove the duplicate privatization. Similarly, for EP, the converted code from Rose encountered a syntax error as the framework performs reduction and privatization to the same variable **gc**. Once again, for correct output, we removed the duplicate conversion. Figure 6B demonstrates the invalid and valid transformations, respectively. For DC, Rose produced illegal parallelization in function *KeyComp()* as depicted in Figure 6C. Having a return statement within parallel pragma is not allowed. Hence, OpenMP reports an invalid branch to/from an OpenMP structured block. For DC, Pluto applied incorrect parallelization by inserting a parallel directive to a for-loop comprising a function call with dependence.

(ii) Incorrect syntactical/logical transformation: For DC, Par4all exhibited two incorrect transformations in functions *PrefixedAggregate()* and *MultiFileProcJobs()*. In *PrefixedAggregate()*, Par4all by default converts switch-case into if-else statements. During this process, it creates a flag variable suffixed and prefixed as "–," which results in an incorrect identifier syntax. During our manual intervention, we removed the "–" symbols from the variable name. In *MultiFileProcJobs()*, Par4all modified the for-loop initialization incorrectly; thus, we retained the serial version as depicted in Figure 6D. Figure 6E illustrates Pluto's transformation in *CreateBinTuple()*, where the shared variable *s must be atomically updated. Pluto could not correctly identify this scalar dependence across function call and resulted in incorrect transformation. In the manually transformed code, we apply

inlining of function and then atomically update the variable *s. However, for evaluation purposes, we had to remove the OpenMP pragma to produce the correct output.

For `EP`, Pluto failed to preserve dependence while applying data locality optimization, resulting in incorrect output. We manually fixed the dependence. Figure 6F shows the original and incorrectly transformed codes. In the case of the `LU` benchmark, Rose performed incorrect transformation in *l2norm()* by replacing one of its arguments, `double v[][ldy/2*2+1][ldx/2*2+1][5]` as `double v[][][][5]`. This led to a syntax error due to the array having incomplete element type. Hence, we replaced it with the original array definition. For `UA`, the parallelized code by Par4all resulted in a segmentation fault. Our investigation revealed that closing braces were wrongly inserted at multiple places in file *mason*, which retained a valid C syntax, but changed the control flow. Hence, we reverted to the original file for evaluation.

(iii) Unrecognized problems: For few of the incorrect transformations by Pluto, although the source of the error is not deduced, we could identify the function that led to such a pitfall. For `BT`, the incorrect output is due to the code segment in function *initialize()*. Similarly, for benchmarks `FT` and `MG`, the problematic region is *compute_initial_conditions()* of `FT` and *zran3()* of `MG`. For `LU`, the incorrect transformation was due to *error()*, *setbv()* before parallelization. For `SP`, *rhs_norm()* results in incorrect output. We excluded the problematic functions from further parallelization and replaced them with the corresponding serial versions. While such a replacement would reduce performance, it ensures correct output.

4.2.2 | Experimental results: NPB result analysis

We now discuss the quantitative effect of various parallelization frameworks on NPB benchmarks. Table 8 shows the description and the static characteristics of the benchmarks, obtained using the `candl` tool. We used the problem size of Class C (Column 8) for all the benchmarks except for `DC`, for which we used Class B input.‖ The classes mainly differ in the sizes of the arrays and, in turn, the number of iterations.[70] Column 9 lists the sequential runtime in seconds for each benchmark.

Table 7 lists the average speedup achieved by various frameworks on NPB. We observe that, relative to PolyBench, NPB benchmarks are less amenable to parallelization with these frameworks, and ICC produces the best overall speedup.

Figure 7 shows the speedup achieved by the frameworks on each benchmark.‡ The results indicate that the performance delivered by the frameworks in parallelizing the NPB benchmarks was largely abysmal, except that by ICC. ICC could achieve small but better performance improvement in most of the benchmarks. To analyze the reasons behind this behavior of the frameworks, we compare the number of loops parallelized by each framework against the base version of NPB, which is manually parallelized; we call it Original. The reasons help us deduce the problems in the nonparallelized loops listed in Table 9.

Since the amount of parallelism achieved depends upon the number of compute-intensive portions of the program, we analyze in Table 10 the number of loops parallelized in the compute-intensive functions. We use the GNU profiler gprof to find the most time-consuming functions for each benchmark. The third column in the Table lists the percentage of total running time of the program used by the compute-intensive functions. We calculated the loop counts of the serial code parcount from the Original code and the frameworks' transformed codes. We categorize parcount into four cases to understand the complexity of loop parallelization: (i) single loop S, (ii) perfectly nested loop N, (iii) single loop within complex nested loop CS, and (iv) complex nested loop CN, which includes nested loops comprising one or multiple single/nested loops as well as imperfectly nested loops. We list these counts in Table 10 for Serial (nontransformed), for Original (manually parallelized NPB), and for the auto-parallelized codes output by the five frameworks. The speedups are computed relative to Serial, and we expect frameworks to approach the performance of Original (the speedup with 32 threads is listed below each benchmark in a separate row).

ICC shows the best performance improvement among all the frameworks for these benchmarks. During parallelization, ICC targets only a few loops for parallelization (see Table 10), yet achieves good speedup. To understand this, we carried out further analysis. A comparison was made with ICC's speedup results of P32 (speedup with 32 threads) and P1 (single-threaded speedup). The performance of P32 is the same as P1, ie, speedup does not improve with increasing threads. ICC does not exploit parallelism, and hence, it is similar to other frameworks. Due to the inherent optimization (ICC baseline uses optimized level -O2) apart from parallelization, nominal speedups were observed for benchmarks (with 32 threads: `BT` 3.4×, `EP` 2.0×, `FT` 2.6×, `LU` 4.6×, `MG` 5.5×, and `SP` 5.8×). Note that ICC works at the IR level, whereas the other frameworks are source-to-source translators.

Among the other four frameworks, Par4all shows performance improvement in two of the benchmarks (with 32 threads: `CG` 11.9× and `SP` 1.7×). We note that Par4all performs better for `CG` when compared to ICC. This indicates that array privatization can lead to significant performance improvement.

Although the value of parcount for most of the frameworks is equal to that of Original, the speedup is poor due to some of the nontrivial issues, as we discuss next in the subsections below.

Summary of issues identified in the nonparallelized loops

Table 9 illustrates the issues that forbid auto-parallelization of benchmarks. The issues are grouped into six different categories, namely, (i) small loop, (ii) nested parallelism problems, (iii) scalar and non-affine issues, (iv) trivial parallelization issues, (v) dependence problems, and (vi) error and

‖ The problem size for the `DC` benchmark was predefined up to Class B.

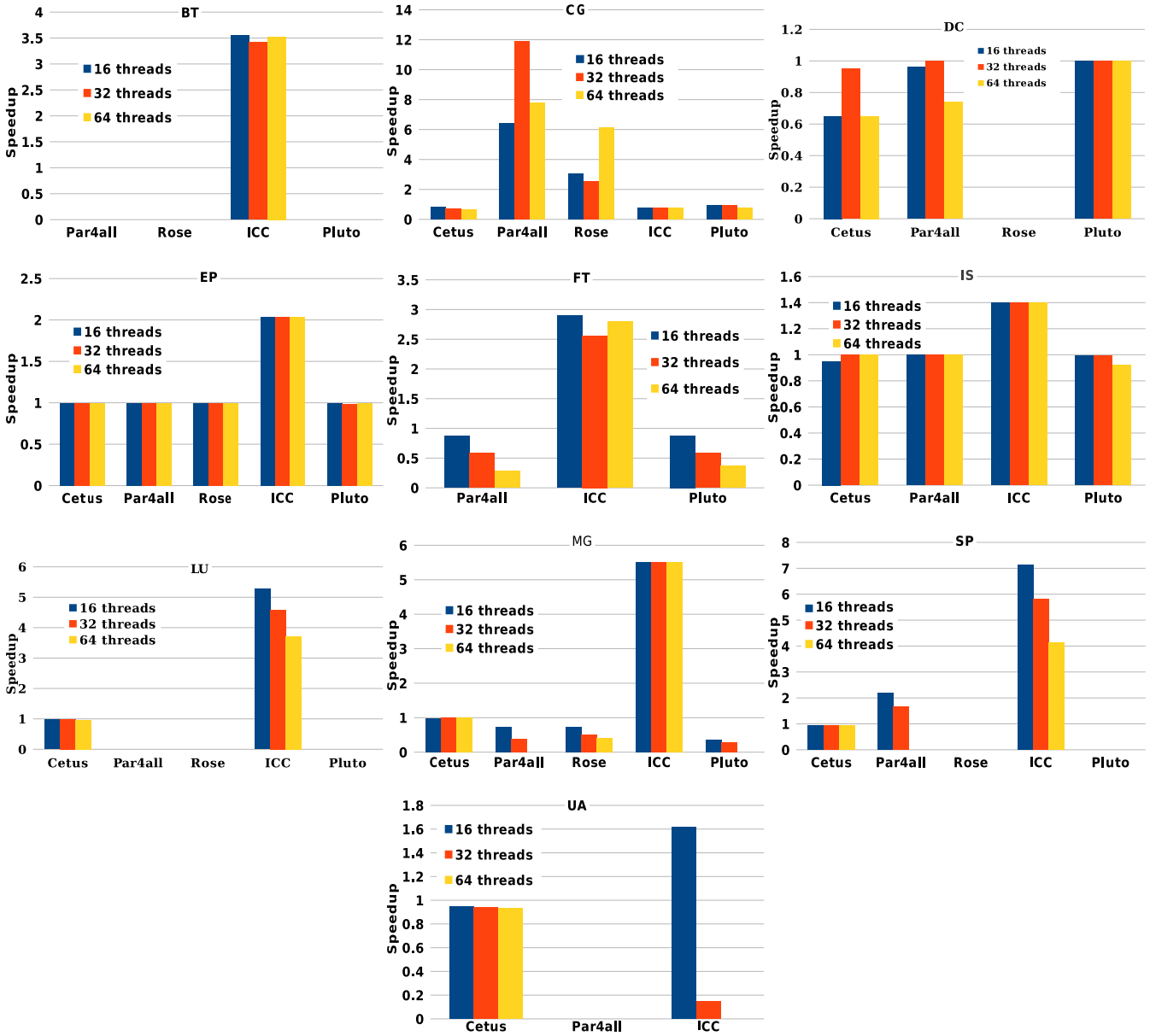


FIGURE 7 Quantitative analysis of NPB benchmarks (note the different scales for the *y*-axes)

timeout cases. Some of these (such as small loop) are relatively easy to handle, whereas some (such as dependence problems) are nontrivial. We observe that these lead to a significant drop in performance.

On the whole, Cetus and ICC do not entertain small loops (loops with few iterations). This leads to a positive effect, as the creation of threads incurs a considerable overhead. Thus, small loops do not compensate for this parallelism overhead.

Nested parallelism problems, primarily MI/MO (missed inner/outer loop parallelism), prevail in almost all parallelizers. Also, complex loop-carried dependence problems (LC) are not addressed by parallelizers. Parallelization of complex loops is a unique problem specific to Cetus. On the other hand, Pluto lacks the capability to handle scalar variables and non-affine issues, leading to not parallelizing loops.

As discussed earlier, ICC works at the intermediate representation and outputs only the parallelization report. ICC does not lead to parallelization due to (i) it not treating a loop a parallelization candidate (NPC), (ii) that it sometimes outputs no optimization reported (NOP) since the compiler is not able to introduce optimization, or (iii) the existence of parallel dependence (PDEP).

A few loops are not acquiescent to parallelization due to simple implementation issues. Parallelizers do not support certain constructs, namely, if constructs (IF), exits and returns (EXIT), function calls (FC), while statement (while), file I/O operations (FILE), declaration statements within for-loop (DECL), array updation (AU), left/right-shift operator (LS/RS), switch-case statements, and upper-to-lower bound (UL).

Apart from this, there occur error and timeout cases in the Rose compiler. This shows that Rose produces erroneous code transformation and does not parallelize loops with a large number of statements. We discuss further in the next section the reasons behind the tools' inability to parallelize and the poor performance of the resultant codes.

TABLE 9 Problems identified in nonparallelized loops

Categories	Problems	BT	CG	DC	EP	FT	IS	LU	MG	SP	UA
Small loops	ICW	I	I	I	CI	I	CI	CI	I	CI	CI
	Complex	C	✗	✗		C		C	C	C	C
	MO	PR	CPR	CPRL		✗		PRL	CPRL	CPRL	PR
Nested parallelism	MI	I	✗	✗	✗	✗	✗	I	I	I	I
	NOP	I	I	✗		✗		I	I	I	I
	NPC	I	I	I		I		I	✗	I	I
	NAL	L	L	L	✗	✗	✗	✗	✗	L	✗
Scalar and non-affine issues	NAS	✗	✗	L	✗	✗	L	✗	✗	✗	✗
	S	L	L	L	L	L	L	L	L	L	L
	SI	✗	✗	✗	✗	✗	L	✗	✗	✗	✗
	FC	CPR	✗	CPR	CPR	CP	CP	✗	CPR	✗	CPR
	IF	✗	CPRL	CPRL	CPRL	PL	CPL	L	CPRL	✗	CPR”
	EXIT	✗	CPRL	CPRL	CPRL	✗	✗	✗	✗	✗	✗
	while	✗	✗	CPRL	✗	✗	✗	✗	✗	✗	✗
	FILE	✗	✗	CPRL	✗	✗	✗	✗	✗	✗	✗
Trivial parallelization issues	DECL	✗	✗	CPRL	✗	✗	✗	✗	✗	✗	✗
	MS	PL	✗	✗	✗	C	✗	L	✗	✗	✗
	AU	✗	✗	✗	✗	✗	CP	✗	✗	✗	✗
	LS	✗	✗	✗	✗	✗	CPL	✗	✗	✗	✗
	RS	✗	✗	✗	✗	✗	CPL	✗	✗	✗	✗
	UL	✗	✗	✗	✗	✗	✗	CPRL	CPRL	CPRL	✗
	switch-case	✗	✗	✗	✗	✗	CL	✗	✗	✗	✗
Dependence problems	PDEP	I	I	I	I	I	I	I	I	I	I
	LC	✗	CPRL	✗	CPR	CP	CPL	✗	CPR	✗	✗
Error and timeout cases	ERR	✗	✗	✗	✗	R	R	✗	✗	✗	✗
	Timeout					R	R				

C, Cetus; **P**, Par4all; **R**, Rose; **I**, Intel C Compiler; **L**, Pluto;
Complex, complex loops; **FC**, function call; **MS**, more number of statements; **MO**, missed outer-loop parallelism; **MI**, missed inner-loop parallelism; **IF**, `if` constructs in the parallel region; **NPC**, not a parallelization candidate; **ICW**, insufficient computational work; **PDEP**, existence of parallel dependence; **NOP**, no optimization reported; **LC**, loop-carried dependence; **NAL**, non-affine loop bound; **NAS**, non-affine array subscript; **S**, operations on a scalar variable; **EXIT**, exits and intermediate returns; **DECL**, declarations within `for`-loop; **while**, while statement; **FILE**, file I/O operations; **AU**, array updation; **LS/RS**, left/right-shift operator; **ERR**, error during auto-parallelization; **Timeout**, timeout during auto-parallelization; **UL**, upper-to-lower bound; **SI**, scalar increment

Execution overhead problems

We observed three common scenarios that caused overhead in the execution time of the auto-parallelized codes.

1. *Shortcomings in the usage of implicit barriers.* In Table 10, parcount values of Par4all and Rose are approximately equal to that of Original for `BT`, `LU`, and `SP`. This indicates that the two frameworks were able to identify almost the same number of loops as those that are manually parallelized. Also, Pluto's parcount is minimum when compared to those of Par4all and Rose but is nonzero. However, Original (`BT` 9.9×, `LU` 12.3×, and `SP` 9.3×) outperforms other frameworks that caused timeout execution. These benchmarks contain multiple, independent, consecutive loops within the parallel region, and Original uses the synchronization qualifier nowait at the appropriate places, thereby removing barrier synchronization and improving the load balancing. Auto-parallelizers lack such a sophistication that requires high-level understanding about the use of implicit barriers or needs to identify dependences across codes beyond the loop body. A code fragment that depicts the usage of the **nowait** clause is shown in Figure 8A.
2. *Inefficient parallelization.* We found a significant improvement by Par4all in the parallelization of `CG`. Although the numbers of loops parallelized by Par4all and Rose are comparable to Original (refer to Table 10), Par4all (11.89×) performs significantly better than Rose (2.53×). Rose inserts `parallel pragma` naïvely for all the loops of the nested-for, which eventually leads to execution overhead, as observed in Figure 8B. This example outlines that since the tool targets the loops in the compute-intensive function (*conj_grad()* taking 93.4% of the execution time) for parallelization, a good speedup is achieved by Par4all. We observe several transformations done by Pluto for `LU` and `MG`, which lead to timeout and poor speedup.
3. *Parallelization of insignificant loops.* Parallelization of small loops may lead to a performance drop or execution overhead. One such scenario is depicted in Figure 8C. For `EP`, Par4all (1.00×), Rose (1.00×), and Pluto (0.99×) insert parallel pragma to shorter loops, which results in

TABLE 10 Performance measurement based on parallelism achieved in the compute-intensive functions

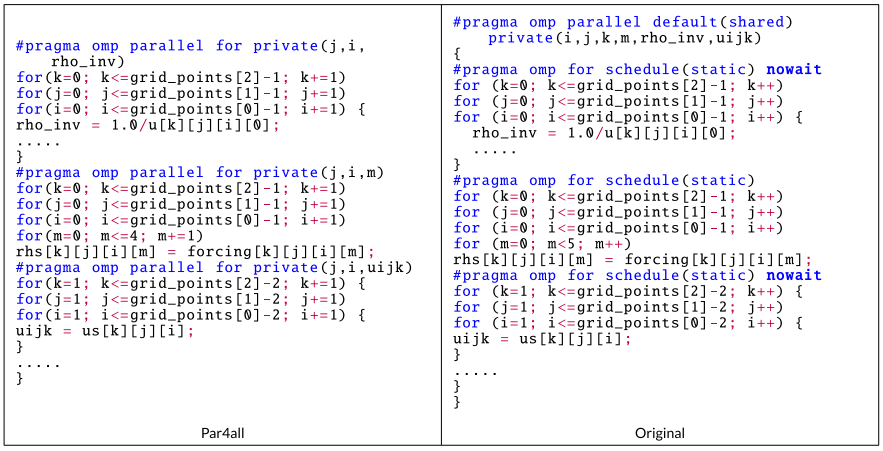
Benchmark	Functions	Time, %	Serial		Cetus		Par4all		Rose		ICC		Pluto		Original	
			S	N	S_p	N_p	S_p	N_p	S_p	N_p	S_p	N_p	S_p	N_p	S_p	N_p
BT	binvcrhs	20.34	No loops		No loops		No loops		No loops		No loops		No loops		No loops	
	compute_rhs	17.12	4	52	0	0	4	52	4	52	0	10	4	13	4	52
	z_solve	16.52	3	5	0	0	0	2	2	2	0	0	T		3	5
	y_solve	16.24	3	5	0	0	0	2	2	2	0	0	T		3	5
	x_solve	15.02	3	5	0	0	0	2	2	2	0	0	T		3	5
	matmul_sub	10.76	No loops		No loops		No loops		No loops		No loops		No loops		No loops	
	Speedup				–		0.00		0.00		3.41		0.00		9.94	
CG	conj_grad	93.36	7	5	2	3	7	4	7	4	0	0	2	0	7	4
	sparse	6.59	4	10	1	0	2	1	2	1	0	0	2	0	1	2
	Speedup				0.69		11.89		2.53		0.78		0.97		12.41	
DC	KeyComp	62.98	1	–	0	–	0	–	0	–	0	–	0	–	0	–
	TreeInsert	29.69	No loops		No loops		No loops		No loops		No loops		No loops		No loops	
	SelectToView	5.70	1	–	0	–	0	–	0	–	0	–	0	–	0	–
	Speedup				0.95		1.00		–		1.76		1.00		14.99	
EP	vranlc	75.44	1	–	0	–	0	–	0	–	0	–	0	–	0	–
	main	24.71	6	1	1	0	3	0	3	0	1	0	2	0	5	1
	Speedup				0.99		1.00		1.00		2.04		0.99		21.29	
FT	Swarztrauber	81.94	–	9	T		–	2	ERR		–	0	–	2	NA	
	fftXYZ	9.86	2	12	T		0	4	ERR		0	0	0	0	NA	
	evolve	4.92	–	3	–	0	3	–	ERR		–	0	–	3	NA	
	Speedup				0.98		0.58		–		2.56		0.58		13.89	
IS	rank	49.65	9	–	1	–	3	–	T		0	–	1	–	NA	
	randlc	41.90	2	–	0	–	2	–	T		0	–	0	–	NA	
	fullly_verify	4.92	3	–	1	–	1	–	T		1	–	1	–	NA	
	Speedup				1.00		1.00		–		1.40		0.99		11.23	
LU	rhs	22.39	14	25	0	0	14	21	14	21	0	4	opt	opt	14	25
	buts	22.00	2	5	0	0	0	5	0	5	0	0	0	5	0	0
	blts	18.68	2	5	0	0	0	5	0	5	0	0	0	5	0	0
	jacld	17.57	–	2	–	0	–	2	–	2	–	1	–	0	–	0
	jacu	17.37	–	2	–	0	–	2	–	2	–	1	–	0	–	0
	ssor	1.67	11	6	0	0	11	6	ERR		0	3	10	0	11	6
	Speedup				0.99		0.00		0.00		4.58		0.00		12.30	
MG	resid	50.34	2	2	0	0	2	0	2	0	0	0	2	0	2	2
	psinv	28.58	2	2	0	0	2	0	2	0	0	0	2	0	2	2
	interp	7.73	13	8	0	0	13	6	12	7	0	0	opt	opt	13	8
	rprj3	5.96	2	2	0	0	2	0	1	0	0	0	0	0	2	2
	vranlc	3.73	1	–	0	–	0	–	0	–	0	–	0	–	0	–
	norm2u3	1.67	–	3	–	0	–	0	–	0	–	1	–	0	0	3
	Speedup				0.99		0.39		0.50		5.52		0.28		6.38	
SP	compute_rhs	37.39	4	52	0	0	4	52	4	49	0	10	3	35	4	52
	z_solve	18.13	11	16	0	1	11	11	11	11	0	3	7	8	11	16
	y_solve	18.03	11	16	0	1	11	11	11	11	0	3	6	2	11	16
	x_solve	17.44	11	16	0	1	11	11	11	11	0	4	6	2	11	16
	Speedup				0.93		1.65		0.00		5.83		0.00		9.31	
UA	laplacian	51.30	–	15	–	0	–	14	–	14	–	5	–	15	0	–
	convect	14.05	18	16	0	0	0	0	18	0	0	2	2	3	17	16
	diffusion	9.93	4	15	0	0	0	0	4	14	0	3	2	4	4	14
	transf	6.83	4	29	0	0	0	0	0	5	0	1	ERR		4	29
	transfb	6.28	4	29	0	0	0	0	1	10	0	0	ERR		4	29
	Speedup				0.94		0.00		–		0.15		–		11.3	

Function, name of the compute-intensive function; **Time, %**, percentage of the total running time of the program used by the function; **Original**, original parallel version of the NPB code; **S**, single loop count in a simple and complex nested loop; **N**, nested loop count in a perfect and complex nested loop; **ERR** or **T**, error or timeout during auto-parallelization; **NA**, not applicable since the original code is different from serial; **opt**, more optimized code

Nonparallelized code

Parallelized code is not equivalent to the serial code

Not applicable due to error and timeout cases



```
#pragma omp parallel for private(j,i,
    rho_inv)
for(k=0; k<=grid_points[2]-1; k+=1)
for(j=0; j<=grid_points[1]-1; j+=1)
for(i=0; i<=grid_points[0]-1; i+=1) {
rho_inv = 1.0/u[k][j][i][0];
.....
}
#pragma omp parallel for private(j,i,m)
for(k=0; k<=grid_points[2]-1; k+=1)
for(j=0; j<=grid_points[1]-1; j+=1)
for(i=0; i<=grid_points[0]-1; i+=1)
for(m=0; m<=4; m+=1)
rhs[k][j][i][m] = forcing[k][j][i][m];
#pragma omp parallel for private(j,i,uijk)
for(k=1; k<=grid_points[2]-2; k+=1) {
for(j=1; j<=grid_points[1]-2; j+=1)
for(i=1; i<=grid_points[0]-2; i+=1) {
uijk = us[k][j][i];
}
.....
}
```

Par4all

```
#pragma omp parallel default(shared)
    private(i,j,k,m,rho_inv,uijk)
{
#pragma omp for schedule(static) nowait
for (k=0; k<=grid_points[2]-1; k++)
for (j=0; j<=grid_points[1]-1; j++)
for (i=0; i<=grid_points[0]-1; i++) {
  rho_inv = 1.0/u[k][j][i][0];
  .....
}
#pragma omp for schedule(static)
for (k=0; k<=grid_points[2]-1; k++)
for (j=0; j<=grid_points[1]-1; j++)
for (i=0; i<=grid_points[0]-1; i++)
for (m=0; m<5; m++)
rhs[k][j][i][m] = forcing[k][j][i][m];
#pragma omp for schedule(static) nowait
for (k=1; k<=grid_points[2]-2; k++) {
for (j=1; j<=grid_points[1]-2; j++)
for (i=1; i<=grid_points[0]-2; i++) {
uijk = us[k][j][i];
}
.....
}
}
```

Original

(A)

```
.....
#pragma omp parallel for
    private(sum, k)
for(j = 0; j <= lastrow-
    firstrow+1-1; j += 1) {
sum = 0.0;
for(k = rowstr[j]; k <=
    rowstr[j+1]-1; k += 1)
sum = sum+a[k]*p[colidx[k]];
q[j] = sum;
}
.....
```

Par4all

```
.....
#pragma omp parallel for
    private (sum,j,k)
for (j=0; j<=lastrow -
    firstrow+1-1; j+=1) {
sum = 0.0;
#pragma omp parallel for
    private(k) reduction(+:
    sum)
for (k = rowstr[j]; k <=
    rowstr[j+1]-1; k+=1) {
sum = sum+a[k]*p[colidx[k]];
}
q[j] = sum;
}
.....
```

Rose

```
#pragma omp parallel default
    (shared) private(j,k,suml
    )
{
.....
#pragma omp for
for (j = 0; j < lastrow -
    firstrow + 1; j++) {
suml = 0.0;
for (k = rowstr[j]; k <
    rowstr[j+1]; k++) {
suml=suml+a[k]*p[colidx[k]];
}
q[j] = suml;
}
.....
}
```

Original

(B)

```
%Disabled due to low
    profitability:#pragma omp
    parallel for
for (i=0; i<10; i ++ )
{
 q[i]=0.0;
}
```

Cetus

```
if (NQ >= 1) {
  lbp=0; ubp=NQ-1;
#pragma omp parallel for
    private(lbv,ubv,t3)
for(t2=lbp;t2<=ubp;t2++){
    q[t2] = 0.0;;
  }
```

Pluto

```
for (i=0; i<NQ; i++)
{
q[i] = 0.0;
}
```

Original

(C)

```
for (i3=1; i3<=n3-1-1; i3+=1) {
for (i2=1; i2<=n2-1-1; i2+=1) {
#pragma omp parallel for private (i1)
for (i1 = 0; i1 <= n1 - 1; i1 += 1) {
u1[i1] = u[i3][i2 - 1][i1] + .....;
u2[i1] = u[i3 - 1][i2 - 1][i1] + .....;
}
#pragma omp parallel for private (i1)
for (i1=1; i1<=n1-1-1; i1+=1) {
r[i3][i2][i1] = v[i3][i2][i1] - .....);
}}}
```

Rose

```
#pragma omp parallel for default(shared)
    private(i1,i2,i3,u1,u2)
for (i3 = 1; i3 < n3-1; i3++) {
for (i2 = 1; i2 < n2-1; i2++) {
for (i1 = 0; i1 < n1; i1++) {
u1[i1] = u[i3][i2-1][i1] + .....;
u2[i1] = u[i3-1][i2-1][i1] + .....;
}
for (i1 = 1; i1 < n1-1; i1++) {
r[i3][i2][i1] = v[i3][i2][i1] - .....;
}}}
```

Original

(D)

```
for(k = 1; k <= np; k += 1) {
kk = k_offset+k;
.....
if (!(i<=100))
goto _break_5;
ik = kk/2;
if (2*ik!=kk)
t3 = randlc(&t1, t2);
if (ik==0)
goto _break_5;
.....
for(i = 0; i <= (1<<16)-1; i += 1) {
.....
sx = sx+t3;
sy = sy+t4;
.....
}}
```

Par4all

```
#pragma omp parallel default(shared)
    private(k,kk,t1,t2,t3,i,ik)  {
#pragma omp for reduction(+:sx,sy) nowait
for (k=1; k<=np; k++) {
kk = k_offset + k;
.....
for (i=1; i<=100; i++) {
ik = kk / 2;
if ((2 * ik) != kk)
t3 = randlc(&t1, t2);
if (ik == 0)
break;
.....
}
.....
for (i=0; i<NK; i++) {
.....
sx = sx + t3;
sy = sy + t4;
.....
}}}
```

Original

(E)

FIGURE 8 NPB analysis. A, Parallelization of BT: code snippet from *compute_rhs()*. Original uses the nowait clause, which improves concurrent processing; B, Parallelization of CG: code snippet from *conj_grad()*. Rose naïvely targets all the loops, whereas Par4all judiciously chooses loops for parallelization; C, Parallelization of EP: code snippet from *main()*—parallelization of insignificant loop by Pluto; D, Parallelization of MG: code snippet from *resid()*—parallelization of CN loop (missed outer-loop parallelism); E, Parallelization of EP: code snippet from *main()*—parallelization of nonsupported programming construct in CN loop

performance overhead (for instance, we have shown the result of Pluto that causes overhead). Cetus applies conditional parallelism (model-based profitability test as mentioned earlier in Section 2.1), which eliminates the parallel execution of shorter loops. However, Cetus (0.99×) proves inferior compared to all the other frameworks. It is obvious from the study that its parcount is minimum across all the benchmarks, reducing opportunities for parallelism. ICC inherently avoids parallelization of computationally less intensive loops and reports insufficient computational work.

Nested parallelism problems

When we look at the counts of Serial in Table 10, there are several perfectly nested loops (N), as well as several imperfectly or complex nested loops (CN). With the exception of Cetus, all the frameworks support parallelization of both N and CN types. However, the way in which each framework applies the parallelism is distinct. Cetus does not support nested parallelism in many instances due to its complexity (refer to Section 2.1). In the case of nested parallelism (N and CN), it is noticed that ICC parallelizes only the outer loop, and Pluto parallelizes only the inner loop. Par4all and Rose miss outer-loop parallelism while handling CN. Figure 8D illustrates how Original handles the CN in *resid()* of `MG` when compared to Rose. The performance of Original (6.38×) is better than that achieved by Rose (0.50×).

Scalar and non-affine issues in Pluto

As previously illustrated in Section 4.1, Pluto does not parallelize the code when the ℓ-value of the statement is a scalar variable. Most of the compute-intensive loops in the NPB encompass scalar reduction, which is not handled by Pluto. Also, non-affine expressions are not parallelized by this polyhedral framework. In NPB, there occur more challenges such as non-affine loop bounds, non-affine array subscripts, and scalar variables, as shown in Table 9. In particular, Pluto could not parallelize the compute-intensive function *conj_grad()*, which takes 93.4% of `CG`'s execution time. This is due to the presence of scalar variable *sum* and non-affine loop bound *k = rowstr[j]; k<=rowstr[j+1]-1* in the inner loop, as shown in Figure 8B.

Another significant problem that prevents frameworks from parallelization is loop-carried dependence (LC), which is studied in detail in Sections 3.2 and 4.1. Table 9 shows that NPB benchmarks comprise LC problems. This could be handled only by Pluto by applying loop peeling and loop distribution optimization. However, due to some of the non-affine issues, scalar problems, and other nontrivial issues, Pluto performs inferior to Original.

Apart from the nontrivial parallelization problems above, there are several other issues that prevent the frameworks from successful and correct transformation. We discuss those below.

Other parallelization issues

Table 9 depicts the problems that terminate the frameworks in parallelizing some of the loops. Most common problems prevailing in the frameworks are if construct (IF), abnormal exit, and intermediate return (ABN), which hinder the tool from performing loop parallelization. Figure 8E illustrates how the nonsupported programming constructs if and break in `EP` affect the parallelization by Par4all (1.00×). Original (21.29×) performs better than all the other frameworks. It removes the implicit barrier **nowait** and applies scalar reduction besides parallelization. Cetus, Par4all, and Rose do not support interprocedural transformation via a function call (FC). Although ICC discloses details about its unsuccessful behavior, these are generic reasons. To probe further, when we examined the auto-parallelized loops, we found that ICC could support IF and function call (FC). Furthermore, the usage of a few of the programming constructs (highlighted in Table 9 as trivial parallelization issues) inhibits the parallelization of several compute-intensive parts of the code segment, which, in turn, deteriorates performance. These challenges hint at support for more high-level constructs, sophisticated analyses, and user-driven parallelization.

4.2.3 | Effect of static dependences

In Table 8, the fourth column (named Dep) shows the number of static dependences in each NPB benchmark. Since dependences affect parallel performance, we analyze it for each framework. In particular, we bucketize the range of Dep values, namely, 0-30, 31-60, and >60, and observe the speedup of benchmarks within each range. As described in the PolyBench study, we expect the speedup to be smaller when the dependence is more.

Figure 9 summarizes the impact of the Dep range affecting the speedup for all the five frameworks. The frameworks show a varied behavior for the same set of benchmarks. Cetus performs almost the same across various Dep values and, in several cases, results in increased time compared to the single-threaded version (speedup less than unity). Par4all has a similar behavior except for `CG`, where the speedup is the largest of all the frameworks. This occurs as most of the loops are dependence free, and Par4all parallelizes the maximum number of loops in `CG` compared to other frameworks. Rose could not parallelize all the benchmarks; thus, we cannot make a conclusive remark. However, we do observe that its parallelization improvement of NPB is limited to `CG` alone. Similar to other frameworks, Pluto also is not effective on NPB and results in performance degradation on various benchmarks. On the other hand, ICC shows consistent improvement on most of the benchmarks. Surprisingly, the only benchmark for which ICC results in less-than-one speedup is `CG`. This occurs due to ICC's cost model, which marks most of the loops to be *not parallelization candidates* due to insufficient computational work predicted. However, more surprisingly, the speedup ICC achieves is more for larger values of Dep. This is a clear indicator that in the case of NPB, the number of static dependences or the number of RAW dependences cannot faithfully capture the parallelization effect, and other parameters (such as the dynamic dependences, opportunities for loop transformations, etc) dominate performance.

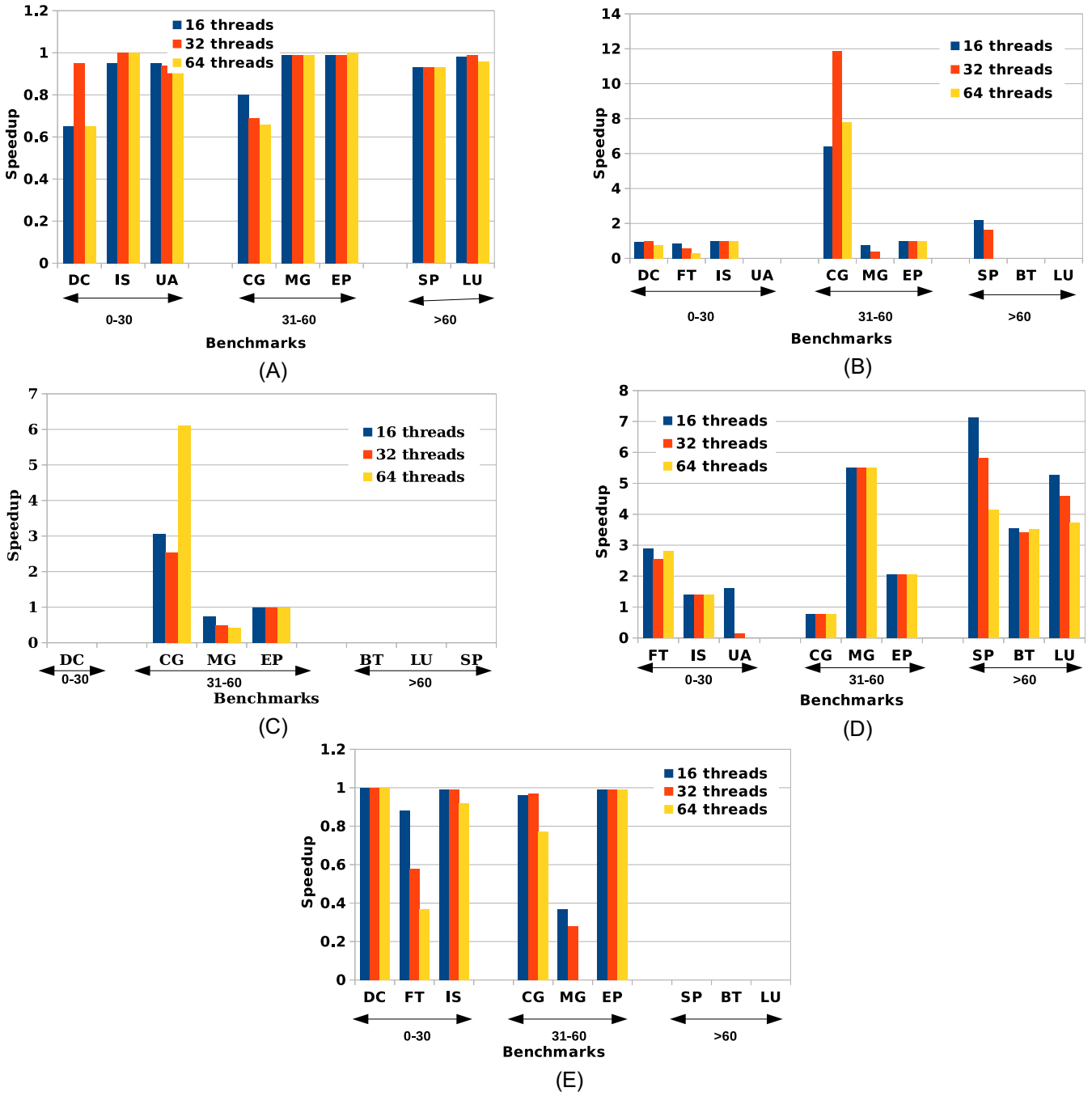


FIGURE 9 Speedup analysis of NPB benchmarks. A, Cetus; B, Par4all; C, Rose; D, ICC; E, Pluto

4.3 | Summary

We studied the performance behavior of various benchmarks parallelized by different frameworks. These frameworks differ in their parallelization capabilities and mechanisms. All the five frameworks are able to identify loop-independent dependence and effectively parallelize such loops. We found that parallelizers are successful in parallelizing the regular and affine loops such as the PolyBench benchmark suite. Among the five frameworks, ICC clearly stands out. For PolyBench benchmarks, Pluto and Par4all resulted in significant performance improvement. Pluto successfully parallelizes applications with complex loop-carried dependence. ICC supports interprocedural analysis and optimization. Insignificant loops are not parallelized by Cetus and ICC. However, when we consider NPB benchmarks that have irregular and non-affine patterns with complex code structures, all the frameworks face issues during parallelization. Some of the significant issues are listed below.

- Excluding ICC and Cetus, all the other frameworks produce erroneous code transformation that leads to incorrect output or syntactical error.
- Rose is the only framework that causes error during transformation.
- Inefficient parallelization is performed by Par4all, Rose, and Pluto. This causes overhead in the execution time.

- Imperfectly or complex nested loops are not handled well by the frameworks.
- Scalar and non-affine issues lead to ineffective parallelization by Pluto.

We think that these issues are quite serious and forbid widespread acceptance of tools. Several of the compute-intensive functions in the NPB are dependence free, yet the complex code structure prohibits the frameworks from achieving parallelization. While several of these issues can be identified with more sophisticated static analysis, others require either profile-driven approaches or user annotations to achieve more parallel performance. Another way to take care of complex syntactic constructs is to work at the IR level (similar to ICC) or even at the assembly code level.

5 | RELATED WORK

Several studies have been conducted on the performance evaluation of auto-parallelizer frameworks. Nobayashi and Eoyang[30] performed a comparative analysis on automatic vectorizing compilers. Their work illustrates that automatic vectorization along with restructuring loops results in improved speedup. Shen et al[31] presented an evaluation of parallelizing compilers, which focuses on data dependence analysis and a few parallel execution techniques. Several studies[26,27,71,72] assessed the effectiveness of parallelizing compilers and their techniques using Perfect benchmarks. The studies emphasized that scalar expansion and reduction replacement are critical in achieving significant performance gains in a large portion of the benchmark suite. Our work has similar goals for the modern set of parallelizing frameworks, and it assesses them by focusing primarily on the various kinds of loop dependences.

Hisley et al[29] evaluated the effectiveness of the SGI compiler in parallelizing CFD codes. The study reported that when additional user-inserted compiler directives were combined with automatic parallelization capabilities of the compiler, impressive speedups were obtained. The goals of our study complement their work.

Eigenmann et al[1] studied the Kap and Vast parallelizing compiler using Perfect benchmarks. They assessed that parallelization of reduction operations and the substitution of generalized induction variables were the primary contributors to performance.

Scientific applications for high-performance computing can be incredibly complex containing, in general, irregular control structures with complicated dependence structures. This complexity makes it difficult for compilers to analyze and to perform optimizations.[25,73] This is especially applicable for source-to-source transformers. Several works[74-77] deal with improving the data dependence analysis techniques to handle such complex structures for program parallelization. Banerjee et al[36] presented an overview of automatic program parallelization techniques, which covers dependence analysis techniques. The work surveys several experimental studies on the effectiveness of parallelizing compilers. Automatic parallelization techniques that focus on irregular and general-purpose programs were developed and studied earlier.[78-81] In our work also, we examined the framework using NPB that comprises several problems, including irregular patterns, imperfectly nested loop, dependences, and other code complexity issues. We investigated the inability of the tools due to these code complexity issues and provided possible solutions. We also highlight the situations where user intervention was needed.

The significance of loop transformations is described in earlier works.[32,34,35] Loop transformation enhances coarse-grain parallelism. The authors reported that tuned application programs had shown better results. Rauchwerger[82] discussed the two important and effective transformations, namely, privatization and reduction parallelization, which can be applied to the loop. Former research studies[39,40,83] reveal that array privatization showed good performance improvement. We made similar observations in our evaluation. Kim et al[84] examined the performance analysis of the Polaris compiler using Perfect benchmarks. The authors found that reduction operation can make a significant improvement in speedup. Our evaluation also finds that privatization and reduction variable recognition result in substantial performance improvement during auto-parallelization.

Bacon et al[56] evaluated a large number of compiler transformations and high-level program restructuring techniques. The authors demonstrated that these transformations could yield high performance when applied appropriately. Studies on loop fusion[85-87] and loop fission[88] have shown improved parallelism as well as data locality. Also, loop fusion reduced the use of temporary arrays. Loop unrolling is a well-known loop transformation that has been used in optimizing compilers; it improves instruction-level parallelism. Davidson and Jinturkar[89] and Sarkar[90] evaluated the importance of loop unrolling. A set of former works[11,91] shows the performance impact of loop tiling, which provides load balance, locality, and parallelism. Another important loop transformation technique is loop peeling.[92,93] The technique relies on moving computations in early iterations out of the loop body such that the remaining iterations can be executed in parallel. In our setup, Pluto supports loop peeling. The frameworks based on the polyhedral model are more powerful[55,94] for automatic optimization, parallelization, and dependence analysis.[95] Experimental evidence shows that the use of the model in parallelizers has shown a significant gain in speedup.

The closest to our work is a study by Mustafa and Eigenmann,[25] who evaluated the performance of five parallelizing compilers (Cetus, OpenUH, Rose, PGI, and ICC). The authors studied the individual techniques of the parallelizers. Along with the individual techniques of the parallelizers, we also examined the common loop transformation techniques, namely, loop tiling, loop fusion, loop unrolling, and loop fission. We included polyhedral-based parallelizer, Pluto, and PIPS-based parallelizer, Par4all, in our study.

A few performance studies[25,33] reveal that Cetus-parallelized code along with tuning proves beneficial in achieving considerable performance. We compare various frameworks with Cetus in our research. However, throughout the study, we retained the default behavior of Cetus without additional tuning optimizations to have uniformity with respect to other frameworks.

Overall, our work contributes in (i) the performance evaluation of modern parallelizing compilers, (ii) investigating the ability of the tools in addressing the dependence structures, (iii) illustrating the role of loop transformation techniques in gaining speedup along with parallelization, and (iv) studying the auto-parallelizer behavior in parallelizing regular as well as irregular programs.

6 | CONCLUSIONS AND FUTURE WORK

This paper provides an overview of the characteristic features of five auto-parallelizers: Cetus, Par4all, Rose, ICC, and Pluto. Using PolyBench and NPB benchmark suites, we have studied the capabilities and limitations of these auto-parallelizers. We investigated the effect of various kinds of loop dependences and loop transformations toward achieved speedup and found that ICC fares as an overall good parallelizer. Pluto works quite well on certain types of programs (such as PolyBench) but needs to be strengthened to become more general-purpose. Cetus, Par4all, and Rose sometimes achieve considerable benefits over other frameworks, depending upon the program constructs and opportunities for parallelization.

The performance analysis on PolyBench shows that among the five frameworks, ICC stands out as it does vectorization besides parallelization and additional optimization. Pluto and Par4all resulted in significant performance improvement. It showed that polyhedral optimization exhibited by Pluto and array privatization evinced by Par4all showed considerable benefits. Cetus supported the least number of benchmarks due to dependence issues and the lack of support for intraprocedural optimization. It performs inferior to all other frameworks.

The performance behavior of NPB codes parallelized by different frameworks was studied. For affine codes such as the PolyBench suite, the parallelizers, mainly ICC and Pluto, exhibited better speedup. However, in the case of the NPB suite, all the frameworks face issues during parallelization due its irregular coding structures. Following are few primary issues that should be taken care of.

1. All the frameworks produce erroneous code transformation that leads to incorrect output or syntactical error except ICC and Cetus.
2. ICC does not exploit parallelism for NAS parallel benchmarks (NPB) codes, and the performance improvement was observed due to its inherent baseline optimization (-O2).
3. Rose is the only framework that causes an error *during* transformation.
4. There occurred overhead in execution time due to inefficient parallelization by Par4all, Rose, and Pluto.
5. Imperfectly or complex nested loops are not handled well by the frameworks.
6. Pluto does ineffective parallelization due to scalar and non-affine issues.

The abovementioned issues are quite serious and forbid the widespread acceptance of tools. Several of the compute-intensive functions in the NPB are dependence free, yet the complex code structure prohibits the frameworks from achieving parallelization. Although Pluto and ICC perform optimization besides parallelization for regular codes with an analyzable pattern, these tools do not fetch benefit and become ineffective for the codes comprising irregular coding style. It would be interesting to build a meta-auto-parallelizer that provides combined benefits of various frameworks.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the constant support and motivation provided by Dr AK Bhaduri, Director, IGCAR. S Prema would like to thank the DAE fellowship for a perspective research grant for PhD. We would like to acknowledge Jyothi Vedurada, Somesh Singh, and Joe Augustine of the Programming Languages, Architecture and Compiler Education Laboratory (PACE), IIT Madras, for their valuable suggestions and reviewing the manuscript.

ORCID

S. Prema https://orcid.org/0000-0002-2530-7426

REFERENCES

1. Eigenmann R, Hoeflinger J, Padua D. On the automatic parallelization of the perfect benchmarks(R). *IEEE Trans Parallel Distributed Syst*. 1998;9(1):5-23.
2. Schulte W, Tillmann N. Automatic parallelization of programming languages: past, present and future. In: *Proceedings of the 3rd International Workshop on Multicore Software Engineering (IWMSE)*. New York, NY: ACM; 2010:1-1. http://doi.acm.org/10.1145/1808954.1808956
3. Midkiff SP. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. San Rafael, CA: Morgan & Claypool Publishers; 2012. *Synthesis Lectures in Computer Architecture*; vol. 19. http://doi.org/10.2200/S00340ED1V01Y201201CAC019
4. Bae H, Mustafa D, Lee J-W, et al. The Cetus source-to-source compiler infrastructure: overview and evaluation. *Int J Parallel Prog*. 2013;41(6):753-767. http://doi.org/10.1007/s10766-012-0211-z
5. Dave C, Bae H, Min SJ, Lee S, Eigenmann R, Midkiff S. Cetus: a source-to-source compiler infrastructure for multicores. *Computer*. 2009;42(12):36-42.
6. Lee S-I, Johnson TA, Eigenmann R. Cetus—an extensible compiler infrastructure for source-to-source transformation. In: *Languages and Compilers for Parallel Computing: 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003. Revised Papers*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2003:539-553.

7. Johnson TA, Lee S-I, Fei L, et al. Experiences in using cetus for source-to-source transformations. In: *Languages and Compilers for High Performance Computing 17th International Workshop, LCPC 2004, West Lafayette, IN, USA, September 22-24, 2004, Revised Selected Papers*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2005:1-14. http://doi.org/10.1007/11532378_1
8. Amini M, Creusillet B, Even S, et al. Par4All: from convex array regions to heterogeneous computing. Paper presented at: IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC; 2012; Paris, France. https://hal-mines-paristech.archives-ouvertes.fr/hal-00744733
9. Ventroux N, Sassolas T, Guerre A, Creusillet B, Keryell R. SESAM/Par4ALL: a tool for joint exploration of MPSoC architectures and dynamic dataflow code generation. In: *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '12)*. New York, NY: ACM; 2012:9-16. http://doi.acm.org/10.1145/2162131.2162133
10. Bondhugula U, Baskaran M, Krishnamoorthy S, Ramanujam J, Rountev A, Sadayappan P. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In: *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction (CC '08/ETAPS '08)*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2008:132-146. http://dl.acm.org/citation.cfm?id=1788374.1788386
11. Bondhugula U, Hartono A, Ramanujam J, Sadayappan P. A practical automatic polyhedral parallelizer and locality optimizer. *ACM SIGPLAN Not*. 2008;43(6):101-113. http://doi.acm.org/10.1145/1379022.1375595
12. Gómez-Sousa H, Arenaz M, Rubiños-López Ó, Martínez-Lorenzo JÁ. Novel source-to-source compiler approach for the automatic parallelization of codes based on the method of moments. Paper presented at: 2015 9th European Conference on Antennas and Propagation (EuCAP); 2015; Lisbon, Portugal.
13. Lobeiras J, Arenaz M, Hernández O. Experiences in extending parallware to support OpenACC. In: *Proceedings of the Second Workshop on Accelerator Programming Using Directives (WACCPD '15)*. New York, NY: ACM; 2015:4:1-4:12. http://doi.acm.org/10.1145/2832105.2832112
14. Quinlan D, Liao C. The rose source-to-source compiler infrastructure. Paper presented at: Cetus Users and Compiler Infrastructure Workshop in conjunction with PACT 2011, Vol 2011; 2011; Galveston Island, TX.
15. Quinlan D. Rose: compiler support for object-oriented frameworks. *Parallel Process Lett*. 2000;10(02n03):215-226. http://www.worldscientific.com/doi/abs/10.1142/S0129626400000214
16. Tian X, Bik A, Girkar M, Grey P, Saito H, Su E. Intel® OpenMP C++/FORTRAN compiler for hyper-threading technology: implementation and performance. *Intel Technol J*. 2002;6(1).
17. Grosser T, Zheng H, Aloor R, Simbürger A, Größlinger A, Pouchet L-N. Polly - polyhedral optimization in LLVM. In: Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT), Vol 2011; 2011; Chamonix, France.
18. Ierotheou CS, Jin H, Matthews G, Johnson SP, Hood R. Generating OpenMP code using an interactive parallelization environment. *Parallel Comput*. 2005;31(10-12):999-1012. Part of special issue: OpenMP. http://www.sciencedirect.com/science/article/pii/S0167819105001080
19. Bluemke I, Fugas J. A tool supporting C code parallelization. In: *Innovations in Computing Sciences and Software Engineering*. Dordrecht, The Netherlands: Springer Science+Business Media BV; 2010;259-264. https://doi.org/10.1007/978-90-481-9112-3_44
20. Kwon D, Han S, Kim H. MPI backend for an automatic parallelizing compiler. In: Proceedings Fourth International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN); 1999; Fremantle, Australia.
21. Hall MW, Anderson JM, Amarasinghe SP, et al. Maximizing multiprocessor performance with the SUIF compiler. *Computer*. 1996;29(12):84-89.
22. Wilson R, French R, Wilson C, et al. *The SUIF Compiler System: A Parallelizing and Optimizing Research Compiler*. Stanford, CA: Stanford University; 1994. Technical Report.
23. Blume B, Eigenmann R, Faigin K, et al. Polaris: the next generation in parallelizing compilers. In: *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 1994:141-154.
24. Blume W, Doallo R, Eigenmann R, Grout J, Hoeflinger J, Lawrence T. Parallel programming with polaris. *Computer*. 1996;29(12):78-82. http://doi.org/10.1109/2.546612
25. Mustafa D, Eigenmann R. PETRA: performance evaluation tool for modern parallelizing compilers. *Int J Parallel Prog*. 2015;43(4):549-571. http://doi.org/10.1007/s10766-014-0307-8
26. Blume W, Eigenmann R. Performance analysis of parallelizing compilers on the perfect benchmarks programs. *IEEE Trans Parallel Distributed Syst*. 1992;3(6):643-656.
27. Eigenmann R, Blume W. An effectiveness study of parallelizing compiler. In: *Proceedings 20th International Conference Parallel Processing 1991, Vol II*. Boca Raton, FL: CRC Press; 1991:II-17-II-25.
28. Chapman B, Jost G, van der Pas R. *Using OpenMP: Portable Shared Memory Parallel Programming*. Cambridge, MA: The MIT Press; 2007. *Scientific and Engineering Computation*.
29. Hisley D, Agrawal G, Pollock L. Evaluating the effectiveness of a parallelizing compiler. In: *Languages, Compilers, and Run-Time Systems for Scalable Computers: 4th International Workshop, LCR '98 Pittsburgh, PA, USA, May 28-30, 1998 Selected Papers*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 1998:195-204. https://doi.org/10.1007/3-540-49530-4_14
30. Nobayashi H, Eoyang C. A comparison study of automatically vectorizing FORTRAN compilers. In: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing (Supercomputing); 1989; Reno, NV.
31. Shen Z, Li Z, Yew PC. An empirical study of FORTRAN programs for parallelizing compilers. *IEEE Trans Parallel Distributed Syst*. 1990;1(3):356-364.
32. Imai T. Detecting more independent loops across hierarchical structures. In: Proceedings of Fourth International Conference on Computing and Information (ICCI); 1992; Toronto, Canada.
33. Dave C, Eigenmann R. Automatically tuning parallel and parallelized programs. In: *Languages and Compilers for Parallel Computing: 22nd International Workshop, LCPC 2009, Newark, DE, USA, October 8-10, 2009, Revised Selected Papers*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2010:126-139. *Lecture Notes in Computer Science*; vol. 5898. http://doi.org/10.1007/978-3-642-13374-9_9
34. Mustafa D, Aurangzeb A, Eigenmann R. Performance analysis and tuning of automatically parallelized OpenMP applications. In: *Proceedings of the 7th International Conference on OpenMP in the Petascale Era (IWOMP '11)*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2011:151-164. http://dl.acm.org/citation.cfm?id=2023025.2023041

35. Tiwari A, Chen C, Chame J, Hall M, Hollingsworth JK. A scalable auto-tuning framework for compiler optimization. In: Proceedings of the 2009 IEEE International Symposium on Parallel Distributed Processing (IPDPS); 2009; Rome, Italy.
36. Banerjee U, Eigenmann R, Nicolau A, Padua DA. Automatic program parallelization. *Proc IEEE*. 1993;81(2):211-243.
37. Kennedy K, Allen JR. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. San Francisco, CA: Morgan Kaufmann Publishers Inc; 2002.
38. Wolfe MJ. *Optimizing Supercompilers for Supercomputers*. Cambridge, MA: MIT Press; 1990.
39. Liu Z, Chapman B, Weng T-H, Hernandez O. Improving the performance of OpenMP by array privatization. In: *Proceedings of the OpenMP Applications and Tools 2003 International Conference on OpenMP Shared Memory Parallel Programming (WOMPAT '03)*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2003:244-259. http://dl.acm.org/citation.cfm?id=1761900.1761925
40. Li Z. Array privatization for parallel execution of loops. In: *Proceedings of the 6th International Conference on Supercomputing (ICS '92)*. New York, NY: ACM; 1992:313-322. http://doi.acm.org/10.1145/143369.143426
41. Blume W, Eigenmann R, Hoeflinger J, et al. Automatic detection of parallelism: a grand challenge for high-performance computing. *IEEE Parallel Distrib Technol Syst Appl*. 1994;2(3):37-47. http://doi.org/10.1109/M-PDT.1994.329796
42. Cook S. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. 1st ed. Waltham, MA: Morgan Kaufmann Publishers Inc; 2013.
43. Stone JE, Gohara D, Shi G. OpenCL: a parallel programming standard for heterogeneous computing systems. *Comput Sci Eng*. 2010;12(3):66-73.
44. Keryell R, Ancourt C, Creusillet B, Coelho F, Jouvelot P, Irigoin F. *PIPS: A Workbench for Building Interprocedural Parallelizers, Compilers and Optimizers*. Paris, France: École Nationale Supérieure des Mines de Paris; 1996. Technical Paper. PIPS4U.
45. Amini M, Ancourt C, Coelho F, et al. PIPS is not (just) polyhedral software adding GPU code generation in PIPS. Paper presented at: First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011) in Conjunction With CGO; 2011; Chamonix, France. https://hal-mines-paristech.archives-ouvertes.fr/hal-00744312
46. Irigoin F, Jouvelot P, Triolet R. Semantical interprocedural parallelization: an overview of the PIPS project. In: *Proceedings of the 5th International Conference on Supercomputing (ICS '91)*. New York, NY: ACM; 1991:244-251. http://doi.acm.org/10.1145/109025.109086
47. Creusillet B, Irigoin F. Interprocedural array region analyses. *Int J Parallel Program*. 1996;24(6):513-546. http://doi.org/10.1007/BF03356758
48. Pouchet L-N. PoCC: the polyhedral compiler collection. 2013.
49. Bastoul C, Cohen A, Girbal S, Sharma S, Temam O. Putting polyhedral loop transformations to work. In: *Languages and Compilers for Parallel Computing: 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003. Revised Papers*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2004:209-225. https://doi.org/10.1007/978-3-540-24644-2_14
50. Bastoul C. *Openscop: A Specification and A Library for Data Exchange in Polyhedral Compilation Tools*. Orsay, France: University of Paris-Sud; 2011. Technical Report.
51. Bastoul C. A polyhedral representation extractor for high level programs. Clan; 2008.
52. Bastoul C. Code generation in the polyhedral model is easier than you think. In: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. Washington, DC: IEEE Computer Society; 2004:7-16. https://doi.org/10.1109/PACT.2004.1342537
53. Ahmed N, Mateev N, Pingali K. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In: *Proceedings of the 14th International Conference on Supercomputing (ICS '00)*. New York, NY: ACM; 2000:141-152. http://doi.acm.org/10.1145/335231.335245
54. Verdoolaege S. isl: an integer set library for the polyhedral model. In: *Proceedings of the Third International Congress Conference on Mathematical Software (ICMS '10)*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2010:299-302. http://dl.acm.org/citation.cfm?id=1888390.1888455
55. Benabderrahmane M-W, Pouchet L-N, Cohen A, Bastoul C. The polyhedral model is more widely applicable than you think. In: *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*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2010:283-303. http://doi.org/10.1007/978-3-642-11970-5_16
56. Bacon DF, Graham SL, Sharp OJ. Compiler transformations for high-performance computing. *ACM Comput Surv*. 1994;26(4):345-420. http://doi.acm.org/10.1145/197405.197406
57. Banerjee U. *Dependence Analysis*. New York, NY: Springer Publishing Company, Incorporated; 2013.
58. Solihin Y. *Fundamentals of Parallel MULTICORE Architecture*. 1st ed. Boca Raton, FL: Chapman & Hall/CRC; 2015.
59. Padua DA, Wolfe MJ. Advanced compiler optimizations for supercomputers. *Commun ACM*. 1986;29(12):1184-1201. http://doi.acm.org/10.1145/7902.7904
60. Xue J. *Loop Tiling for Parallelism*. New York, NY: Kluwer Academic Publishers; 2000.
61. Hill MD, Marty MR. Amdahl's law in the multicore era. *Computer*. 2008;41(7):33-38. http://doi.org/10.1109/MC.2008.209
62. Hennessy JL, Patterson DA. *Computer Architecture: A Quantitative Approach*. 5th ed. Waltham, MA: Morgan Kaufmann Publishers Inc; 2011.
63. Pouchet LN. PolyBench: the polyhedral benchmark suite. 2015.
64. Pouchet L-N, Yuki T. PolyBench/C 3.2. 2012.
65. Bailey DH, Barszcz E, Barton JT, et al. The NAS parallel benchmarks. *Int J High Perform Comput Appl*. 1991;5(3):63-73. https://doi.org/10.1177/109434209100500306
66. Bailey DH, Barszcz E, Barton JT, et al. The NAS parallel benchmarks—summary and preliminary results. In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. New York, NY: ACM; 1991:158-165. http://doi.acm.org/10.1145/125826.125925
67. Seo S, Kim J, Jo G, Lee J, Nah J, Lee J. SNU NPB suite. 2016.
68. Griebler D, Loff J, Fernandes LG, Mencagli G, Danelutto M. Efficient NAS benchmark kernels with C++ parallel programming. In: Proceedings of the 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP); 2018; Cambridge, UK.
69. Drozdov AY, Novikov SV, Vladislavlev VE, Kochetkov EL, Il'in PV. Program auto parallelizer and vectorizer implemented on the basis of the universal translation library and LLVM technology. *Program Comput Softw*. 2014;40(3):128-138. http://doi.org/10.1134/S0361768814030037
70. Wong P, van der Wijngaart R. *NAS Parallel Benchmarks I/O Version 2.4*. Moffett Field, CA: NASA Ames Research Center; 2003. NAS Technical Report NAS-03-00.

71. Eigenmann R, Hoeflinger J, Li Z, Padua DA. Experience in the automatic parallelization of four perfect-benchmark programs. In: *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 1992:65-83. http://dl.acm.org/citation.cfm?id=645669.665205
72. Eigenmann R. Toward a methodology of optimizing programs for high-performance computers. In: *Proceedings of the 7th International Conference on Supercomputing (ICS '93)*. New York, NY: ACM; 1993:27-36. http://doi.acm.org/10.1145/165939.165948
73. Göhringer D, Tepelmann J. An interactive tool based on polly for detection and parallelization of loops. In: *Proceedings of Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM '14)*. New York, NY: ACM; 2014:1:1-1:6. http://doi.acm.org/10.1145/2556863.2556869
74. Blume W, Eigenmann R. Nonlinear and symbolic data dependence testing. *IEEE Trans Parallel Distributed Syst*. 1998;9(12):1180-1194.
75. Blume W, Eigenmann R. The range test: a dependence test for symbolic, non-linear expressions. In: *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing (Supercomputing '94)*. Los Alamitos, CA: IEEE Computer Society Press; 1994:528-537. http://dl.acm.org/citation.cfm?id=602770.602858
76. Psarris K, Kyriakopoulos K. Nonlinear symbolic analysis for advanced program parallelization. *IEEE Trans Parallel Distributed Syst*. 2008;20(5):623-640.
77. Kyriakopoulos K, Psarris K. Efficient techniques for advanced data dependence analysis. Paper presented at: 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05); 2005; St. Louis, MO.
78. Kim H, Johnson NP, Lee JW, Mahlke SA, August DI. Automatic speculative DOALL for clusters. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. New York, NY: ACM; 2012:94-103. http://doi.acm.org/10.1145/2259016.2259029
79. Raman A, Kim H, Mason TR, Jablin TB, August DI. Speculative parallelization using software multi-threaded transactions. *SIGARCH Comput Archit News*. 2010;38(1):65-76. http://doi.acm.org/10.1145/1735970.1736030
80. Campanoni S, Jones T, Holloway G, Reddi VJ, Wei G-Y, Brooks D. HELIX: automatic parallelization of irregular programs for chip multiprocessing. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. New York, NY: ACM; 2012:84-93. http://doi.acm.org/10.1145/2259016.2259028
81. Vandierendonck H, Rul S, De Bosschere K. The paralax infrastructure: automatic parallelization with a helping hand. In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT '10)*. New York, NY. Association for Computing Machinery (ACM); 2010:389-400.
82. Rauchwerger L. Run-time parallelization: its time has come. *Parallel Comput*. 1998;24(3-4):527-556. http://doi.org/10.1016/S0167-8191(98)00024-6
83. Tu P, Padua D. Automatic array privatization. In: Pande S, Agrawal DP, eds. *Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques, and Run Time Systems*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2001:247-281. http://dl.acm.org/citation.cfm?id=380466.380474
84. Kim SW, Voss M, Eigenmann R. Performance analysis of compiler-parallelized programs on shared-memory multiprocessors. In: Proceedings of Compilers for Parallel Computers (CPC); 2000; Aussois, France.
85. Blainey B, Barton C, Amaral JN. Removing impediments to loop fusion through code transformations. In: *Languages and Compilers for Parallel Computing: 15th Workshop, LCPC 2002, College Park, MD, USA, July 25-27, 2002. Revised Papers*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2005:309-328. http://doi.org/10.1007/11596110_21
86. Fraboulet A, Kodary K, Mignotte A. Loop fusion for memory space optimization. In: *Proceedings of the 14th International Symposium on Systems Synthesis (ISSS '01)*. New York, NY: ACM; 2001:95-100. http://doi.acm.org/10.1145/500001.500025
87. Manjikian N, Abdelrahman TS. Fusion of loops for parallelism and locality. *IEEE Trans Parallel Distrib Syst*. 1997;8(2):193-209. http://doi.org/10.1109/71.577265
88. Kennedy K, McKinley KS. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In: *Languages and Compilers for Parallel Computing: 6th International Workshop Portland, Oregon, USA, August 12-14, 1993 Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 1994;301-320. https://doi.org/10.1007/3-540-57659-2_18
89. Davidson JW, Jinturkar S. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. In: *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO 28)*. Los Alamitos, CA: IEEE Computer Society Press; 1995:125-132. http://dl.acm.org/citation.cfm?id=225160.225184
90. Sarkar V. Optimized unrolling of nested loops. *Int J Parallel Prog*. 2001;29(5):545-581. http://doi.org/10.1023/A:1012246031671
91. Bandishti V, Pananilath I, Bondhugula U. Tiling stencil computations to maximize parallelism. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. Los Alamitos, CA: IEEE Computer Society Press; 2012:40:1-40:11. http://dl.acm.org/citation.cfm?id=2388996.2389051
92. Song L, Glück R, Futamura Y. Loop peeling based on quasi-invariance/induction variables. *Wuhan Univ J Nat Sci*. 2001;6(1-2):362-367. https://doi.org/10.1007/BF03160270
93. Song L, Kavi KM. A technique for variable dependence driven loop peeling. Paper presented at: Fifth International Conference on Algorithms and Architectures for Parallel Processing, 2002. Proceedings; 2002; Beijing, China.
94. Simbürger A, Apel S, Größlinger A, Lengauer C. The potential of polyhedral optimization: an empirical study. In: *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE '13)*. Piscataway, NJ: IEEE Press; 2013:508-518. https://dl.acm.org/citation.cfm?id=3107720
95. Bhattacharyya A, Amaral J. Automatic speculative parallelization of loops using polyhedral dependence analysis. In: *Proceedings of the First International Workshop on Code Optimisation for Multi and Many Cores (COSMIC '13)*. New York, NY: ACM; 2013:1:1-1:9. http://doi.acm.org/10.1145/2446920.2446921

How to cite this article: Prema S, Nasre R, Jehadeesan R, Panigrahi BK. A study on popular auto-parallelization frameworks. *Concurrency Computat Pract Exper*. 2019;e5168. https://doi.org/10.1002/cpe.5168