Automatic Construct Selection and Variable Classification in OpenMP

Mohammad Norouzi
Technische Universitaet Darmstadt
Darmstadt, Germany
norouzi@cs.tu-darmstadt.de

Felix Wolf
Technische Universitaet Darmstadt
Darmstadt, Germany
wolf@cs.tu-darmstadt.de

Ali Jannesari
Iowa State University
Ames, Iowa
jannesari@iastate.edu

ABSTRACT

A major task of parallelization with OpenMP is to decide where in a program to insert which OpenMP construct such that speedup is maximized and correctness is preserved. Another challenge is the classification of variables that appear in a construct according to their data-sharing semantics. Manual classification is tedious and error prone. Moreover, the choice of the data-sharing attribute can significantly affect performance. Grounded on the notion of parallel design patterns, we propose a method that identifies code regions to parallelize and selects appropriate OpenMP constructs for them. Also, we classify variables in the chosen constructs by analyzing data dependences that have been dynamically extracted from the program. Using our approach, we created OpenMP versions of 49 sequential benchmarks and compared them with the code produced by three state-of-the-art parallelization tools: Our codes are faster in most cases with average speedups relative to any of the three ranging from 1.8 to 2.7. Additionally, we automatically reclassified variables of OpenMP programs parallelized manually or with the help of these tools, improving their execution time by up to 29%.

KEYWORDS

parallelization, OpenMP, design patterns, data-sharing semantics

ACM Reference Format:
Mohammad Norouzi, Felix Wolf, and Ali Jannesari. 2019. Automatic Construct Selection and Variable Classification in OpenMP. In *2019 International Conference on Supercomputing (ICS '19), June 26–28, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3330345.3330375

1 INTRODUCTION

Multi-core systems have become the mainstream architecture in the computer industry, and OpenMP is a major programming paradigm for this architecture. It provides a set of compiler directives, library routines, and environment variables that allow sequential source code to be incrementally parallelized with little effort.

An important step during the parallelization with OpenMP is selecting the right constructs (e.g., worksharing loop or task) and inserting them at the right position into the program. The goal is to achieve maximum speedup without violating correctness. An OpenMP construct instructs the compiler and the runtime system how to enable parallel execution of the enclosed code block. Usually, the decision where to insert which construct is made by the programmer without tool support [23, 25], which is time consuming and error prone. Compiler-based methods exist [2, 4, 16] but are too conservative because they usually assume parallelism-preventing dependences where they do not occur in practice. Further methods [26] are confined to the worksharing-loop construct. Another step during OpenMP parallelization is the classification of variables that appear inside a construct according to their data-sharing semantics. OpenMP defines a default attribute for each variable, but this might not always be correct or the most efficient choice. Therefore, programmers should reconsider the default. This process is tedious and error-prone, considering the large number of variables that can potentially appear in a construct. To ease the burden, automatic methods have been proposed. A majority of them are based on static data-dependence analysis [2–4, 16, 23, 25], but fail due to variable aliasing or when dependences stretch across multiple compilation units. On the other hand, existing methods based on dynamic data dependences [26] do not classify variables that require program-scope analysis. Also, they classify variables only in worksharing-loop constructs. In this paper, we propose a novel approach that automatically selects appropriate OpenMP constructs, fits them into the code, and classifies variables used in their dynamic extent. It leverages an existing technique that suggests possible design patterns that are suitable to parallelize a sequential program [8]. We extend this work in that we now identify components of a pattern in the code and map them onto an OpenMP construct. We determine the memory access pattern of variables belonging to each construct from dynamically acquired data dependences and derive suitable data-sharing attributes. In this way, we move from a mere suggestion of patterns towards their semi-automatic implementation. We assessed the performance of our approach in comparison to previously parallelized versions of the benchmarks and three parallelization tools: PLUTO [4], autoPar [16], and Mercurium [3]. We make the following specific contributions beyond the state of the art:

- In comparison to the three competitors, the parallel version produced by our tool chain achieves superior performance in most cases with average speedups relative to any of the three ranging from 1.8 to 2.7.
- Unlike the compilers, which parallelize programs using either worksharing-loop or task constructs but not both in combination, our tool chain chooses between loop and task construct. This allows more parallelization opportunities to be exploited in a broader spectrum of programs.

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330375

- Our variable classification method is not limited to specific types of variables, clauses, or constructs, further reducing the classification effort.
- Our classification method makes more efficient choices. Reclassifying variables in programs parallelized by any of the three tools or in their pre-existing parallel versions, made the programs up to 29% faster.

The remainder of the paper is organized as follows. We discuss related work in Section 2. In Section 3, we explain the methods our approach is built on top of. Section 4 presents our approach, which is followed by an evaluation in Section 5. Finally, we review our achievements in Section 6.

2 RELATED WORK

In the field of parallel computing, design patterns have been introduced to identify and express parallelism on different levels, ranging from the decomposition of an abstract computational problem down to the selection of specific parallel-programing constructs. Mattson et al. [18] and McCool et al. [19] collected numerous design patterns, including DOALL, reduction, pipeline, and task parallelism, and discussed their implementation using OpenMP. In this paper, we refer to a more recent feature set of OpenMP, including tasking and multi-dimensional array reduction, which helps avoid some of the code restructuring described in their work.

There are several methods that identify parallelizable code sections and choose suitable OpenMP constructs. Some of them require substantial programmer support—either to identify the regions to be parallelized [25] or to address correctness problems pinpointed without providing precise resolution instructions [23]. Both consume considerable time and limit programmer productivity. Our approach also involves the programmer, but rather at the end to insert OpenMP constructs according to precise specifications including data-sharing attributes. Auto-parallelizing tools [4, 13, 16, 21] accomplish the task automatically, but may miss parallelization opportunities because the values of pointers and array indices are often not visible at compile time, making parallelization more conservative than it needs to be. The polyhedral model, the main vehicle of popular auto-parallelizing compilers such as PLUTO [4], is limited to loop nests with affine loop bounds and array accesses [4]. Surprisingly, methods based on dynamic data-dependence analysis do not reach beyond loops either [26]. Our approach also leverages dynamic data dependences, but can choose between OpenMP worksharing-loop and task constructs.

Furthermore, there are methods for automatic variable classification. We divide them into two groups. The first group uses static data-dependence analysis. For example, autoPar [16], a tool based on the ROSE compiler framework [20], inserts worksharing-loop and task constructs into sequential programs. It applies use-def chain and liveness analysis to classify variables in loop constructs. These techniques, however, are not sufficient to determine the data-sharing attribute of all types of variables, for example, they struggle with pointers that appear under different names or global variables affected by inter-procedural dependences. Wang et al. [25] apply the same set of techniques to classify variables in task constructs, while Royuela et al. [22, 23] extended the Mercurium compiler to classify variables in task constructs. Both groups of authors classify variables in task constructs that have been previously inserted into the program, without identifying task dependences. PLUTO also classifies variables in loop constructs [4], but it identifies neither firstprivate, lastprivate, nor reduction variables. In general, static classification methods are likely to fail when the program contains pointer and global variables [24].

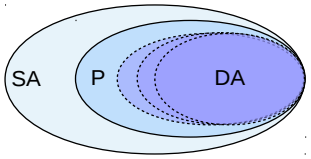


Figure 1: The relation between static and dynamic data dependences. SA contains data dependences that static analysis cannot rule out. P includes data dependences that occur in practice with the set of all possible inputs of a program. DA contains data dependences identified via dynamic analysis with a limited set of inputs.

To overcome these limitations, the second group of classification methods resorts to dynamic data-dependence analysis. Wang et al. [26] profile memory accesses inside loops to classify variables. Aldea et al. [1] identify the data-sharing attribute of variables by speculatively monitoring memory accesses inside loops. These methods, however, are again limited to loops. Also, they cannot tell whether a variable should be declared lastprivate or firstprivate. Using dynamic data-dependence analysis, we are able to classify pointers and global variables. In addition, we cover variables in both worksharing-loop and task constructs and distinguish among private, shared, reduction, lastprivate, and firstprivate variables.

3 BACKGROUND AND OBJECTIVES

Our approach builds on top of DiscoPoP [14], a parallelism discovery tool. DiscoPoP abandons the idea of fully automatic parallelization and instead points programmers to likely parallelization opportunities that it identifies by analyzing dynamic dependences. In this way, it considers only data dependences that actually occur. From these dynamic dependences, DiscoPoP derives possible parallel design patterns that programmers should consider for parallelization. In this sense, one can think of DiscoPoP as a recommender system for parallelization.

Figure 1 shows the relationship between the date dependences that static and dynamic analyses obtain. To better understand the relation shown in the figure, consider the example in Listing 1:

```
1 for(i=0;i<n;i++){
2     w = a[f(i)];
3     a[g(i)]=v;
4 }
```

Listing 1: Code example that illustrates the limitations of static data-dependence analysis.

Depending on the return values of functions f and g, there may be a data dependence between lines 2 and 3 arising from array a. Static analysis cannot not rule out data dependences in such cases. In fact, to be on the safe side, it conservatively assumes that there

is a data dependence. Dynamic approaches, on the other hand, take the opportunity to investigate if a data dependence actually appears in practice with given inputs. As we increase the diversity of inputs, the set of data dependences identified by dynamic analysis may become larger and larger, approximating and eventually becoming equal to the set of possible data dependences, as depicted in Figure 1.

However, as Kim et al. [12] demonstrated, data dependences in frequently executed code regions do not change noticeably with respect to different inputs, a result we confirm in the evaluation of this paper in Section 5. Nevertheless, to minimize the risk of missing dependences, state-of-the-art parallelism discovery tools [8, 11] including DiscoPoP address input sensitivity of dynamic dependences by running the program with a range of representative inputs, considering all dependences that occur. Numerous earlier case studies support that implementing the design patterns that DiscoPoP suggests can reproduce manual parallelization strategies [8, 9]. This is consistent with the results of related tools [11, 12] that recreated manual parallelizations as well. We conclude that parallelism discovery based on dynamic dependences presents a viable alternative to static methods in those cases where the latter are too rigid or too conservative. Of course, the risk of missing dependences cannot be fully eliminated, but we argue later in Section 5 that the validation effort is not higher than with manual parallelization, while saving substantial time through automatic selection of parallelization opportunities and their guided implementation.

The approach we present here takes the output of DiscoPoP, essentially a set of code pieces annotated with patterns, as input and maps them onto suitable OpenMP constructs. Then, it classifies all variables that would be accessed in the dynamic extent of the constructs according to their data-sharing semantics. The overall workflow and the precise relationship between our approach and DiscoPoP is sketched in Figure 2, where our contributions appear highlighted.

As the first step of the workflow, a low-overhead profiler (with an average slowdown of around 90x) [15] determines data dependences with representative program input vectors. This includes dependences that stretch over multiple files. Afterwards, the source code is translated into LLVM IR and then statically decomposed into so-called *computational units* (CUs). CUs are short pieces of code without appreciable parallelization at the thread level. Finally, a matching procedure [8, 9] is used to identify possible design patterns in a graph which is formed of the CUs as vertices and the dependences between them as edges.

As the basis for the method presented in this paper, we extract an internal data structure of DiscoPoP, called *program execution tree* (PET), henceforth briefly referred to as *execution tree* or simply *tree* if the context allows. Figure 3 shows an example. It is called a tree because it reflects the execution flow of the program like a call tree. Its nodes can be of the following types: function, loop, conditional block, or CU. Function, loop, and conditional-block nodes have CUs that lexically appear inside them as children. The tree is annotated with dependences that exist among the CUs, including both data and control dependences. Because of these dependences, which connect its nodes beyond mere parent-child relationships, the tree exhibits properties of both a tree and a more general graph.

To implement a parallel design pattern in OpenMP, a programmer must select the right construct (e.g., loop or task) and insert it

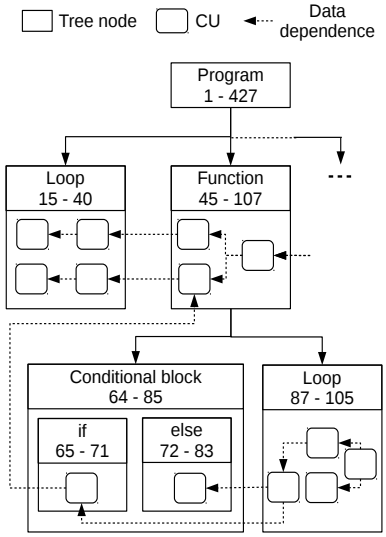


Figure 3: An example execution tree.

at the right position into the program, that is, fit it into the code. Moreover, the programmer must classify variables that are accessed during construct execution according to their data-sharing semantics. These two steps are where we make our main contributions. We automate both (i) construct selection plus fitting and (ii) variable classification. To support the fitting process, we also had to (iii) further elaborate the pattern matching procedure for several patterns. Moreover, to reduce the risk of missing critical but input-sensitive dependences, we now also (iv) perform code-coverage analysis that verifies which CUs have actually been executed with a given input deck and which not. This helps choose a representative set of inputs. All four new contributions are highlighted with dark background in Figure 2. The recommendations produced as the output of our tool chain can be directly implemented in OpenMP. Thus, we move substantially closer towards the final parallel program.

4 APPROACH

Below, we focus on our two main contributions (i) construct selection and fitting and (ii) variable classification. In Section 4.1, we explain how we map the patterns onto OpenMP worksharing-loop and task constructs, after which we describe in Section 4.2 how we classify variables occurring in these constructs according to their data-sharing semantics. The input to these two steps is the execution tree of the program produced by DiscoPoP, annotated with possible (abstract) design patterns that DiscoPoP recommends as foundation of the parallelization. As a prerequisite for the construct selection, we extended the pattern matching procedure in contribution (iii) to identify more complex cases of reduction and determine the reduction operator and the names of reduction variables, a task which was left to the programmer in the previous version of DiscoPoP. In addition, we now detect dependences between recursive tasks and further improve the efficiency of pipelines by merging their stages. The refinements of the pattern matching process are presented in more detail alongside the construct selection in the first subsection, whereas the discussion of the code-coverage analysis

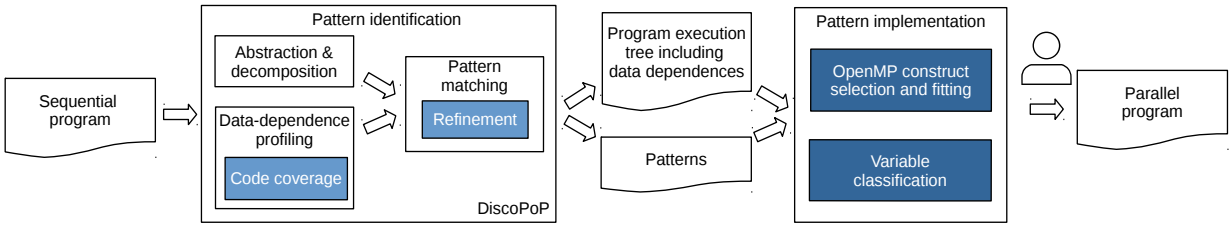


Figure 2: The relation between DiscoPoP and our approach. The dark boxes show our contribution.

we apply to counter input sensitivity (contribution (iv)) is deferred to the evaluation in Section 5.

4.1 Construct Selection and Fitting

The main task of this step is to translate the relatively coarse pattern information into precise OpenMP constructs. This involves identifying the components of each pattern in the source code and deciding where exactly to insert directives. Table 1 shows our suggested OpenMP constructs for each pattern. We explain each pattern in more details as follows.

Table 1: OpenMP constructs suggested for various parallel design patterns.

Parallel pattern	Components	OpenMP construct
DOALL	Loop	Worksharing loop
Reduction	Loop + reduction operation	Worksharing loop + reduction clause
Task parallelism	Workers	Task + depend clause
	Barriers	Taskwait directive
Pipeline	Pipeline stages	Task + depend clause
		Worksharing loop

DOALL. The DOALL pattern can be applied to loops of a program. A loop node in the execution tree is reported as DOALL if there is no inter-iteration dependence in the loop. We merge all child CUs that belong to a DOALL loop into a single loop node. We then enclose the loop with a worksharing-loop construct. We obtain the start and end lines of the construct from the tree.

The programmer can use the schedule clause to specify how the loop iterations are assigned to threads, taking into account both data locality and load balance as the most fundamental scheduling criteria [5]. Possible schedules can be broadly divided into two categories: static and dynamic. A static schedule assigns the iterations before the loop starts. A dynamic schedule assigns the iterations to unoccupied threads while the loop is running. Dynamic schedules can balance the load even if the iterations take an unpredictable amount of time, but they incur periodic synchronization overhead, which is why static schedules are preferable if the amount of work is always the same or follows a regular pattern that can be exploited. To determine the scheduling strategy, we measure the load balance across the iterations of a loop. Unless the loop contains a conditional statement or the boundaries of inner loops depend on the outer loop index, the load is balanced because all iterations execute the same number of instructions. In this case, we suggest static scheduling and dynamic scheduling otherwise. We will consider data locality and uneven but regular load distributions in our future work.

Reduction. Reduction patterns are suitable for a loop with a specific type of inter-iteration dependence, that is, the loop uses an associative binary operator to reduce all elements of a container to a single scalar value. This happens, for example, when a loop adds all the elements of an array. The original version of DiscoPoP already identifies the simplest version of a reduction (i.e., an array reduced to a scalar variable). Here, we cover more complex cases of reduction, including the reduction of a multi-dimensional array into a one-dimensional one. For this purpose, we instrument all LLVM-IR instructions that create inter-iteration dependences in a loop. We record the source-line numbers for each read and write operation on every variable. The variables can be scalar or arrays with any number of dimensions. If a memory address is written only in a single source line and read only at the same source line, we mark the loop as a possible candidate for a reduction. We also remember the reduction variable and operation (e.g., +).

The OpenMP worksharing-loop construct provides a reduction clause. To implement a reduction pattern, we merge the child CUs in a reduction loop into a single loop node. We then enclose the loop with a parallel for construct and add a reduction clause containing the reduction operator plus the variable(s).

Task parallelism. The task-parallelism pattern can be applied to a collection of CUs within a hotspot function that are arranged in a roughly diamond-shaped dependence structure. DiscoPoP classifies the CUs that belong to such a diamond as worker, barrier, or fork. A fork represents the entry point of the pattern (i.e., the top of the diamond). CUs that depend on the fork CU are classified as workers. They form the body of the diamond. CUs that depend on at least two workers are classified as barriers. The bottommost barrier closes the pattern. Note that workers and barriers need not be disjoint. This role assignment occurs before the execution tree is passed on to the construct-selection and fitting unit, the first of the two main contributions of this paper.

Unfortunately, recursive functions, where task parallelism exists abundantly, previously lacked the dependence information needed to recognize such a diamond because DiscoPoP reserved only a single node to represent the execution of such a function, collapsing all related data dependences. Thus, it was impossible to determine

whether recursive function calls depended on each other. Static analysis is insufficient to provide such details either. For this reason, we developed an LLVM pass that identifies data dependences between recursive function calls such that we can determine which call of a function depends on which other calls. The pass receives as input the list of recursive functions. It first assigns a unique identifier *call ID* to each function call and then instruments read and write instructions that appear inside these functions. To determine dependences, we maintain a map that stores a call ID for every memory address. Whenever a memory address x is written or read, we update the map with the call ID of the writer/reader. If a read happens after a write on x, we report a RAW dependence between the reader and the writer.

Once we have determined the dependences between workers and confirmed the diamond structure, we suggest enclosing each worker with a task construct. We refrain from mutual-exclusion synchronization (e.g., critical construct) to resolve data dependences between two workers because this would not preserve the order in which workers access memory. Instead, we propose depend clauses with a specific dependence type, including `in`, `out` and `inout` to maintain dependences between workers. The dependence type indicates how a task is related to a sibling task, that is, whether it prepares data for or receives data from another task. We explain how we identify dependence types in Section 4.2. Moreover, our approach recommends inserting a taskwait directive before barriers. This directive lets the current task region wait on the completion of child tasks generated since the beginning of the current task. Furthermore, we enclose the fork CU with only an OpenMP single directive nested inside a parallel construct. The parallel construct creates the team of threads that will execute the tasks. The single directive ensures that only one thread in the team creates the tasks.

Pipeline. In a loop with inter-iteration dependences, it may still be possible to partly overlap the execution of consecutive iterations with each other, resulting in pipeline parallelism. A pipeline is a sequence of stages, where each stage consumes data from the previous stage and supplies data to the next. If the stages of a pipeline are executed many times, parallelism can be exploited by overlapping the processing of different inputs as they travel through the pipeline.

The stages identified by DiscoPoP, which identifies patterns at the level of single CUs, are often too small to make the pipeline efficient and may still contain forward dependences. A forward dependence is a data dependence that exists between two stages S_j and S_k with $j < k$ if in loop iteration i, S_j needs a result of S_k produced in the previous iteration $i - 1$. A forward dependence adversely affects the execution of the pipeline because an earlier stage of an iteration has to wait for the results of a later stage of the previous iteration.

We resolve forward dependences by merging pipeline stages. We try to merge different combinations of pipeline stages. To verify whether merging has reduced the number of the forward dependences, we compare their number before and after we merge. We keep trying different combination until there is no forward dependence left, which can amount to the elimination of the pipeline in the worst case.

After resolving forward dependences and reaching the appropriate granularity, we suggest how to parallelize the pipeline in OpenMP. We enclose each stage with a task construct. Most important, we propose depend clauses including correctness-preserving dependence types to maintain data dependences between stages. Finally, we enclose the loop in which the pipeline pattern has been identified with a single directive nested inside a parallel construct for the same reasons we mentioned above: to create the team of concurrent threads executing the pipeline only once.

4.2 Variable Classification

Unlike existing methods, our variable classification goes beyond liveness and use-def analyses because they are insufficient to determine the data-sharing semantics in all situations. For example, global variables accessed from different functions or pointers that appear under different names can cause trouble. Instead, we rely on the execution tree of a program, which, in addition to the results of the above analyses, contains inter-procedural and alias information. Using the tree, we identify the data-sharing attribute of variables—regardless of the number of files in which they appear and their number of aliases.

To classify variables, we consider RAW dependences available in the tree. We preserve other dependences by keeping the sequential order of their memory accesses intact. We first explain our approach for worksharing-loop, then for task constructs.

Worksharing-loop construct. Algorithm 1 shows how we classify variables in loop constructs. Figure 4a serves as an illustrating example. According to our algorithm, we first receive the list of identified DOALL and reduction loops from the execution tree. Then, we collect all the child CUs in the dynamic extent of an identified loop node. We then traverse the dynamic extent of the loop to identify variables that should be classified. In the example, `a`, `b`, `c`, `d`, `iter`, `i`, and `init` should be classified. `iters`, `p`, and `q` do not belong to the loop and, therefore, do not require classification.

To prepare the classification, we traverse the execution tree for each loop node using depth-first search (DFS). We put CUs visited before the loop node into its left subtree. Also, we put CUs visited after the loop into its right subtree. In Figure 4a, the left and right subtrees of the loop node contain the CUs in functions `foo` and `bar`, respectively. We create the left and right subtrees because data dependences may span over multiple files in the program and this is why we cannot use source-code line numbers for classification.

For each variable, we then check the source and destination of every RAW dependence. If a variable is written in the left subtree of a loop node and is only read inside the loop node, then its value is passed into the loop. We mark the variable as firstprivate or shared, depending on whether it is local or global. In cc-NUMA architectures, it is preferred to classify local read-only variables as firstprivate [5]. This is because their cache controllers communicate to keep memory coherent when more than one cache holds a copy of the same memory location. For this reason, ccNUMA systems may perform poorly when multiple processors attempt to access the same memory location in rapid succession. To improve performance, we declare a read-only local variable firstprivate. The variable can be an array or a scalar variable. In this way, each processor will access its private memory, which is much faster. In

Algorithm 1: Variable classification for the worksharing-loop construct

```
loops = PET.getDOALLAndReductionLoops()
for each loop ∈ loops do
    variables = loop.collectVariables()
    rightSubTree = PET.visitedCUsInDFSAfter(loop)
    leftSubTree = PET.visitedCUsInDFSBefor(loop)
    for each var ∈ variables do
        if var.isWrittenIn(leftSubTree)
        & var.isReadOnlyIn(loop) then
            if var.isGlobal() then
                var.mark(shared)
            else
                var.mark(firstprivate)
        else if var.isWrittenIn(loop)
        & var.isReadIn(rightSubTree) then
            var.mark(lastprivate)
        else if var.isFirstWrittenIn(loop) then
            if var.isScalar() then
                var.mark(private)
            else
                var.mark(shared)
    if var.isLoopIndexIn(loop) then
        var.mark(private)
    else if var.isReductionIn(loop) then
        var.mark(reduction)
```

Figure 4a, we mark `a` as firstprivate because it is written in the left subtree and only read inside the loop. `iter` is also written in the left subtree and only read inside the loop. Nevertheless, because it is a global variable, we mark it as shared.

To identify lastprivate variables, we check if a variable is written inside the loop and read in the right subtree of the loop, that is, a value is produced inside the loop and consumed after the loop. In Figure 4a, `b` is lastprivate because it is written inside the loop and read after the loop in function `bar`.

If a variable is first written inside the loop but never read after it, then it is useful only for the loop. We mark it as private if it is a scalar. If the variable is an array and every loop iteration accesses only the elements that correspond to the loop index, we mark it as shared because parallel threads will process disjoint chunks of the array. In the figure, array `init` is shared. `c` is private because it is scalar and first written in the loop.

Finally, if a variable is the loop index or a reduction variable, we mark it as private or reduction, respectively. In the example, `i` is private because it is the loop index. `d` is reduction because it creates an inter-iteration dependence that can be resolved using the reduction clause. The actual identification of reduction variables occurs earlier during the refined pattern matching procedure for reduction, as explained before.

Task construct. Algorithm 2 shows how we classify variables in task constructs, illustrated using the example in Figure 4b. The code in the example contains a loop in which a three-stage pipeline has been identified. Our construct selection suggests creating a task for each stage. Our algorithm first obtains the list of tasks from the execution tree. For each task, we collect the child CUs in its dynamic extent. We then traverse the dynamic extent of the task to identify variables that should be classified (`a`, `b`, `c`, `d`, `e`, and `i` in the example). For each task, we traverse the tree again using DFS. We place CUs that are visited before the task node into its left subtree. Because the task construct of OpenMP does not support lastprivate variables, we do not have to maintain a right subtree. We also determine the sibling tasks, which make up the stages in the example. In Figure 4b, Tasks 1-3 make up the pipeline. The left subtrees of Tasks 1 and 3 contain CU_1 and CU_0, respectively.

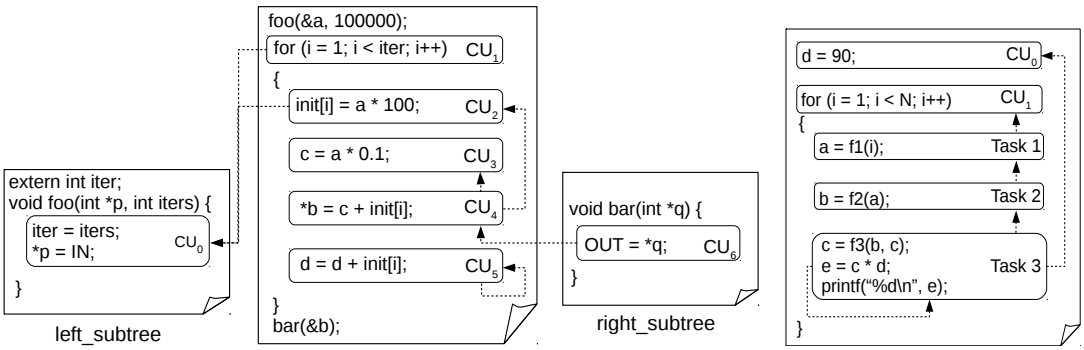
We identify the dependence type of variables in a task by looking for the source and destination of RAW dependences. If the task reads the value of a variable which is written in a sibling task, the task should wait for the sibling task to produce the value first. In this case, we mark the dependence type of the variable as `in`. On the other hand, if a variable is written in the task and read by a sibling, the task should produce a value for the sibling. In this case, we mark the dependence type of the variable as `out`. In the example, there is a RAW dependence between Tasks 1 and 2 concerning variable `a`; the result of the call to function `f1` is stored in `a`, which is later read in the call to function `f2`. We mark the dependence type of `a` as `out` for Task 1 and `in` for Task 2. We find variable `b` in a similar situation, which implies a dependence between Tasks 2 and 3.

If a variable is first read and then written in a task, then every instance of the task should wait for its previously generated instance to produce the value of the variable. In this case, we mark the dependence type of the variable as `inout`. In the figure, we mark the dependence type of `c` as `inout` for Task 3 because the value of `c`, which is read as function parameter in Task 3, should be received from a previously generated instance of the same task.

Also, we classify variables in a task by looking for the source and destination of RAW dependences. We mark a variable as private if it is first written inside a task. In the example, `e` is private in task 3. Finally, if a variable is only read inside a task and written in a CU in its left subtree, then a value is passed into the task by the thread that created the task. We mark the variable as firstprivate or shared, depending on whether it is local or global. In Figure 4b, `i` and `d` are only read in Tasks 1 and 3, respectively. We mark `i` as firstprivate because it is a local variable and `d` as shared because it is global.

5 EVALUATION

Below, we summarize our experimental results and provide details of the test environment and the benchmarks that we used. Our evaluation criteria are the performance and correctness of the benchmarks that we parallelize or whose variables we classify. We ran our tests with benchmarks from three suites, including Polybench 3.2, NPB 3.3, and BOTS 1.1.2. We compare our approach with three state-of-the-art tools that parallelize sequential programs using OpenMP or classify variables in OpenMP constructs: PLUTO 0.11.4, autoPar, which is part of ROSE 0.9.9.13, and the variable classifier of Mercurium 2.0.0. In addition, we compare our approach with



(a) A loop in which a reduction pattern is identified. To parallelize the loop, we suggest the worksharing-loop construct with reduction clause.

(b) A loop in which a three-stage pipeline is identified. We suggest the task construct to exploit parallelism among the the stages.

Figure 4: Variable classification.

Algorithm 2: Variable classification for the task construct

```
tasks = getTasks(PET)
for each task ∈ tasks do
    variables = task.getVariables()
    leftSubTree = PET.visitedCUsInDFSBefor(task)
    siblings = task.getSiblings()
    for each var ∈ variables do
        if var.isWrittenIn(siblings)
        & var.isReadIn(task) then
            var.markDepType(in)
        else if var.isWrittenIn(task)
        & var.isReadIn(siblings) then
            var.markDepType(out)
        else if var.isFirstReadIn(task)
        & var.isWrittenIn(task) then
            var.markDepType(inout)
        else if var.isFirstWritten(task) then
            var.mark(private)
        else if var.isWrittenIn(leftSubTree)
        & var.isReadOnlyIn(task) then
            if var.isGlobal() then
                var.mark(shared)
            else
                var.mark(firstprivate)
```

manual parallelization, that is, pre-existing OpenMP versions of the benchmarks. The NBP and BOTS suites already contain OpenMP versions. Moreover, we considered the OpenMP version of Polybench developed by Grauer-Gray et al. [6]. Since Polybench has been designed as a test suite for polyhedral compilers, it is well suited for comparison with PLUTO. NBP, which offers parallelization potential in nested loops, is a good match for autoPar. BOTS, a set of programs designed for tasking, is the yardstick of our comparison with the classifier of Mercurium, which targets tasking. We compiled the benchmarks using gcc 4.9.3. Experiments were run on an Intel(R) Xeon(R) Gold 6126 2.60 GHz double socket with 64 GB memory, running Ubuntu 14.04 (64-bit edition). Reported execution times are the median of five isolated executions with 48 threads for Polybench and NPB and 24 threads for BOTS, respectively. To compare the different parallelization approaches quantitatively, we calculate average speedups across entire benchmark suites using the geometric mean [7].

5.1 Full Parallelization

The first set of experiments has been designed to evaluate our full parallelization tool chain, including all steps shown in Figure 2. However, before we compare in Section 5.1.2 the speedups achieved by DiscoPoP with those of its competitors, including the manually parallelized versions, we first validate the correctness of the code we produce in Section 5.1.1.

5.1.1 Correctness. First, we establish the output equivalence in comparison to the serial versions. Then, we investigate how sensitively the data dependences respond to changes of the input, before we apply a common data race detector to check whether the parallelized codes are free of races. Finally, we inspect the source code of the parallelized programs manually to see whether their behavior would differ from the hand-crafted versions.

Output equivalence. We ran each program with five different inputs, a number an overwhelming majority of benchmarks (45 out of 49) readily provided. Most of the benchmarks work on arrays and their inputs determine the size and the dimension of those arrays. However, they fill the arrays themselves, either randomly or based on a specific pattern. In the few cases without a sufficient number of predefined inputs, we generated the missing inputs randomly or we constructed them ourselves (e.g., for sort in BOTS). The output of the parallel code produced with the help DiscoPoP consistently matched the output of the serial version.

Table 2: Data races reported by Intel Inspector for DiscoPoP and manual parallelization.

Benchmark suites	DiscoPoP			Manual parallelization		
	Min	Max	Avg	Min	Max	Avg
Polybench	0	4	1.36	0	4	1.23
NPB	1	13	7.50	1	13	7.50
BOTS	0	11	2.36	0	12	2.27

Input sensitivity. An expected source of input sensitivity is code coverage, which we analyze at the level of CUs, the sources and sinks of the data dependences we extract. For this purpose, we traversed the sequential code of each benchmark statically to obtain the set of all CUs in the program. Again, we ran each program with five different inputs and, during each run, marked all the CUs that were visited. It turned out that every benchmark touched all the CUs of the program—regardless of the input. In a next step, we compared the data dependences collected with different inputs one by one. Although we noticed some differences, varying the input did not change dependences for the code regions subject to parallelization.

Data races. Data races constitute one of the most important and harmful classes of parallelization errors. To uncover potential data races, we ran both the parallel versions produced with DiscoPoP and the hand-written parallel benchmarks under the control of Intel Inspector [10], a widely used data-race detector, which, however, is known to report false positives. The results are summarized in Table 2. We scrutinized each single data race reported for the parallel code we produced and could not verify any of them to be true. Given that Intel Inspector reported a comparable number of races for the manually parallelized programs, we argue that checking the code produced with our approach for the occurrence of data races is not more laborious than checking hand-crafted parallel code. In fact, it might even be less if the hand-written versions already underwent significant debugging cycles earlier.

Behavioral differences. Lastly, we visually examined the source code of the parallel versions created with the help of DiscoPoP and found no behavioral differences in comparison with the source code of the hand-crafted parallel versions.

5.1.2 Performance. Some auto-parallelizing tools failed to parallelize some benchmarks. We confirmed the failure of tools by contacting their developers. PLUTO, which is designed for polyhedral parallelization, failed to parallelize NPB and BOTS because the loops in these program suites usually do not satisfy the constraints of the polyhedral model (i.e., affine loop bounds and array accesses). autoPar failed to parallelize BOTS and Polybench. In addition, it failed to parallelize LU, SP, BT, and FT from NPB. The classifier component in Mercurium only classifies variables in OpenMP task constructs. Nevertheless, it failed to classify variables in most benchmarks in BOTS. Grauer-Gray et al. missed to parallelize two benchmarks (i.e., deriche and heat-3d) in Polybench, which they confirmed upon request. Now, we evaluate how the parallel code produced with DiscoPoP performs in comparison to the other three tools and the manually parallelized versions for each benchmark suite. To measure performance, we chose a medium-sized input from the five inputs.

Polybench. Figure 5 shows how our approach performs on Polybench in comparison to PLUTO and parallel versions created by Grauer-Gray et al.. The base line is the serial version of the benchmarks with the same input which we used for their parallel version. We parallelized the benchmarks by PLUTO with `--parallel --tile` (with default settings for tile sizes).

Compared with PLUTO, our codes are faster in most cases. PLUTO performs optimizations including tiling and vectorization as part of its parallelization process. However, the optimizations allowed PLUTO to achieve better speedup for several benchmarks (e.g., trmm and syr2k) in which hotspot loops are DOALL. For these benchmarks, however, our parallelized version is still better than the manually parallelized version. Our parallelization of the benchmark correlation outperforms the manually parallelized version by a factor of 12.5.

For all benchmarks (e.g., seidel and adi) in which the outer hotspot loops are not DOALL while the loops nested inside are, our approach suggested the insertion of OpenMP constructs at slightly different code locations than the manual parallelization did, leading to better performance. In these cases, our approach outperformed PLUTO by a factor of 2.1 on average.

PLUTO leaves the loop-scheduling strategy unspecified, in which case the implementation-defined default applies. In this situation, most implementations schedule statically. Our approach, in contrast, specifies the scheduling strategy individually for each parallel loop. It suggested static scheduling for the loops of jacobi, seidel-2d, heat-3d, floyd-warshall, adi, gemver, and gesummv because their iterations have identical workloads. For the remaining benchmarks in which the DOALL pattern was identified, our approach proposed dynamic scheduling because the workload varies across iterations. We ran the benchmarks with both scheduling strategies (i.e., static and dynamic) and confirmed ours to be superior.

In some benchmarks, we identified complex cases of reduction (e.g., a two dimensional array being reduced to one dimension) stretching across an entire loop nest. However, Grauer-Gray et al. did not parallelize the reduction hidden in nested loops. PLUTO resolved inter-iteration dependences related to possible reductions in the course of other optimizations, forgoing the chance to apply a reduction clause. Using the reduction clause for benchmarks where we identified a reduction pattern, our approach outperformed PLUTO by a factor of 1.9 on average.

We parallelized some benchmarks using the task construct. These include fdtd-2d, ludcmp, and reg_detect, where we identified pipeline patterns. Moreover, we applied the task construct to fdtd-2d, where four DOALL loops are nested inside a non-DOALL loop. The manually parallelized version saw only the DOALL loops parallelized. On the other hand, DiscoPoP found a four-stage pipeline with two forward dependences in the non-DOALL loop. After resolving the forward dependences, DiscoPoP recommended a two-stage pipeline in addition to parallelizing the loops nested inside. This enabled superior speedup in comparison to the manually parallelized version. Similar situations were found in ludcmp and reg_detect. Finally, we discovered task parallelism in deriche and

Table 3: Number and type of OpenMP constructs each approach chose for parallelization. For Polybench, NPB, and BOTS, the competitors are always PLUTO, autoPar, and Mercurium, respectively.

Benchmark suites	# worksharing-loop constructs (w/ reduction clause)			# worksharing-loop constructs (w/o reduction clause)			# task constructs		
	DiscoPoP	Competitor	Manual	DiscoPoP	Competitor	Manual	DiscoPoP	Competitor	Manual
Polybench	7	0	0	61	96	60	27	0	0
NPB	15	20	16	140	74	149	2	0	0
BOTS	1	0	0	1	0	0	62	62	62

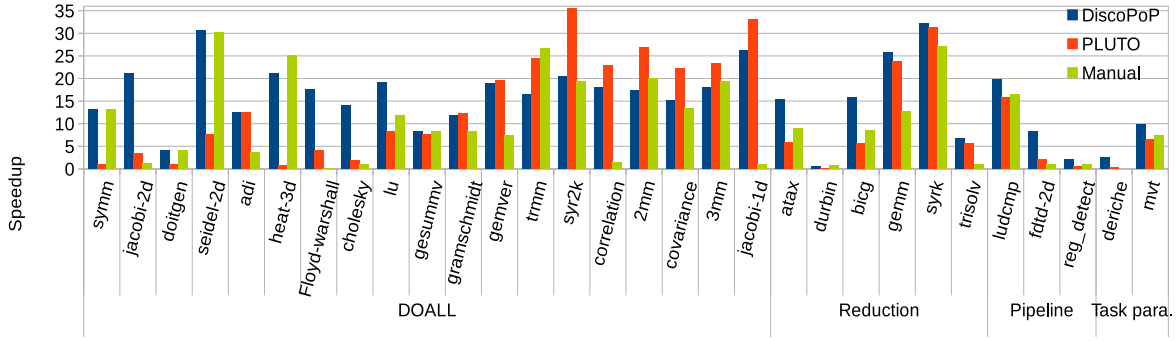


Figure 5: Parallelization of Polybench using our approach vs. PLUTO vs. manual parallelization by Grauer-Gray et al. [6] with 48 threads.

mvt. The latter contains two independent DOALL loops. In the manually parallelized version, each loop was parallelized separately. Our approach suggested running them in parallel to each other using the task construct – in addition to their individual parallelization. Again, this made it faster than the handcrafted version. A similar situation was found in deriche. In all benchmarks where we identified pipeline or task-parallelism patterns, PLUTO and the manually created parallel versions preferred the worksharing-loop construct. Relative to PLUTO, our code was 2.9 times faster on average.

NPB. Figure 6 compares our approach with autoPar and manual parallelization of NPB programs. Our approach came close to the performance of manual parallelization for most of the programs. We realized that the improvements in LU, BT, and SP can be attributed to our variable classification.

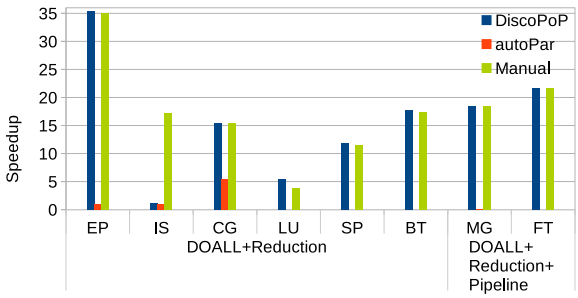


Figure 6: Parallelization of NBP using our approach vs. autoPar vs. parallel versions shipped with NBP with 48 threads.

autoPar missed one reduction in EP and two in IS because it tried to identify them statically. The developers of NPB restructured IS to exploit the available parallelism and thus, it executed faster than our parallelized version. In CG, the DOALL loops had reduction loops nested inside. Although it was sufficient to parallelize only the outer DOALL loops, autoPar chose to parallelize these inner reduction loops as well, causing significant runtime overhead. Moreover, our approach outperformed autoPar with MG. This was because we parallelized a pipeline in addition to reduction and DOALL loops. Using an extra array, the NPB developers parallelized the pipeline using the worksharing-loop construct. Compared with their version, our approach did not produce an improvement for small inputs. For large inputs ($\geq 10^6$), however, their version of MG aborted with a segmentation fault because the memory allocation for the extra array exceeded the available memory. Our version did not need the extra array to run in parallel and therefore finished without segmentation fault also with large inputs. FT behaved like MG except that autoPar failed to parallelize it. We recommended dynamic scheduling only for two loops in CG and static scheduling for the remaining loops and all other benchmarks in the suite, whereas autoPar relied on the default scheduling strategy.

BOTS. Figure 7 compares our approach with the classifier of Mercurium and the manually parallelized version of BOTS. The results for Mercurium were created by taking a manually parallelized version, stripping off all data-sharing attribute clauses, and reclassifying the variables using Mercurium. Essentially, this amounts to manual OpenMP parallelization, followed by an automatic classification of variables. Because the parallelism available in BOTS was less than in the other benchmark suites, we used 24 instead of 48 threads in these experiments.

Our approach identified the task parallelism pattern in all benchmarks except sparselu. It came close to or exceeded the performance of all other parallel versions. sort, sparselu, and floorplan owed their improvements to our variable classification algorithm, results we elaborate in Section 5.2. The performance of alignment was better because we parallelized a reduction loop that was not parallelized elsewhere.

```
1    for (i = 0; i < n; i++) {
2        a[j] = (char) i;
3        if (ok(j + 1, a)) {
4           nqueens(n, j + 1, a,&res);
5           *solutions += res;}
6    }
```

Listing 2: Hotspot loop in nqueens from BOTS.

Listing 2 shows the hotspot loop in nqueens. Task parallelism was identified in this loop; each call to function `nqueens` was marked as a worker task and the statement in line 5 as the barrier for the workers. Our approach suggested inserting a task directive before the call to function `nqueens` and a taskwait directive before the statement in line 5. In the OpenMP version of nqueens from BOTS, the task construct was inserted before the `if`, creating many empty tasks due to a false condition and thus causing significant runtime overhead. Compared with the manual parallelization, our approach achieved a speedup of 10.0.

Table 3 summarizes the number and type of the OpenMP constructs selected by each parallelization approach. In general, the DiscoPoP-based tool chain seems to choose more freely among worksharing-loop and task constructs, whereas the other approaches appear to favor either one or the other. Overall, the code we produced is on average 1.8 times faster than that of PLUTO, 2.7 times faster than that of autoPar, 1.8 times faster than that of Mercurium, and on average very close to the hand-crafted version (i.e., a speedup of 1.3), but with a substantial degree of automation.

5.2 Variable Classification

We compared our variable classification approach with the classification approach of the tools and expert programmers who parallelized the benchmarks. We considered two scenarios. In the first scenario, we executed each benchmark as it was parallelized by the three tools or created by a programmer. In the second, we changed the data-sharing clause of the variables in the parallelized benchmark according to our suggestions. We validated the correctness of our classifications following the same method as in Section 5.1.1 and could not find any error. Figure 8 shows how much our classifier improved the performance beyond the classification selected by either the tool we chose for a benchmark or manual parallelization.

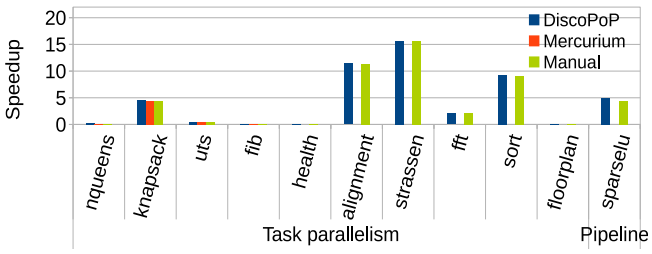


Figure 7: Parallelization of BOTS using our approach vs. Mercurium vs. parallel versions shipped with BOTS with 24 threads.

The Mercurium classifier failed to classify global variables in most of the benchmarks in BOTS. The benchmarks either contained a call to a function that was not defined in the same file where the call occurred or Mercurium failed to identify the synchronization point preceding the creation of tasks in the benchmarks. Our measurement results for the classifier are consistent with the results of Royuela et al. [22], who developed the Mercurium classifier. Our classifier, in contrast, was able to determine the data-sharing semantics of all variables. Compared with the tools and preexisting OpenMP versions of the benchmarks, our classifier improved the performance of benchmarks under three conditions, which we explain below.

Depend clause vs. taskwait, atomic, and critical. Manually parallelized versions of BOTS benchmarks did not use the depend clause. Instead, they used taskwait directive, atomic, and critical constructs. One possible reason could be that the benchmarks were parallelized before the introduction of the depend clause in OpenMP. In general, our approach can be used to re-classify variables in legacy programs, potentially increasing their performance. For example, compared with the preexisting OpenMP version, we improved the execution time of sparselu and floorplan by 17% and 25%, respectively.

Private vs. threadprivate. Martorell et al. [17] showed that threadprivate directives can slow down execution. We observed that the slowdown was negligible if the number of accesses to threadprivate variables was less than a million for the given input and architecture. In LU, however, our classifier suggested using a private clause instead of the threadprivate directive and achieved a performance improvement of 29% in comparison to the preexisting OpenMP version. It made the same recommendation for the variables declared as threadprivate in BT and SP.

Firstprivate vs. shared. PLUTO could not identify firstprivate variables because it identified dependences only within loops. It used shared instead of firstprivate clauses for all variables that could be declared either way without compromising correctness. Similar to PLUTO, Grauer-Gray et al. had classified all such variables as shared in their OpenMP version of Polybench, probably because analyzing dependences at program scope is too laborious. Classifying read-only local variables as firstprivate in Polybench, our classifier improved the execution time of the benchmarks in comparison to PLUTO and manual parallelization by up to 25%. Read-only variables in NPB and BOTS were global. Our classifier suggested shared semantics for them like the three tools and the programmers.

6 CONCLUSION

Our approach strikes a viable compromise between conservative but limited auto-parallelization on the one hand and unguided manual parallelization on the other. Compared with state-of-the-art parallelizing tools, our method is more likely to produce efficient

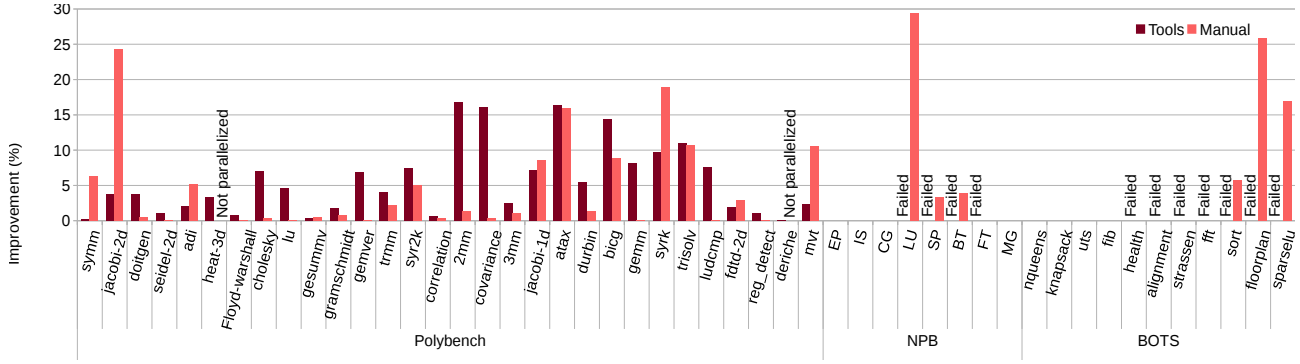


Figure 8: Execution time improvements after applying our classifier to the code produced by either PLUTO, autoPar, and Mercurium on the one hand (Tools) and manual classification on the other. "Not parallelized" indicates that the benchmarks were not parallelized manually. "Failed" indicates that the tool failed to classify variables in the benchmarks or parallelize them.

parallel programs. In addition, it parallelized a broader range of programs from different benchmark suites. Also, we presented an approach for classifying variables in OpenMP worksharing-loop and task constructs. Our approach classified all types of variables in both the loop and task constructs according to different types of data-sharing semantics. We also showed that our approach increases the performance of programs which are already parallelized by reclassifying their variables. In the future, we plan to evaluate our method with more complex programs. Moreover, we aim to include further OpenMP constructs such as taskloop, and support additional parallel patterns, including geometric decomposition. Also, we seek to define a metric that helps prioritize the patterns before they are implemented. Finally, we intend to exploit more parallelism by providing guidance for code refactoring beyond the insertion of pragmas.

ACKNOWLEDGEMENT

This work has been funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project. Additional support has been provided by the German Research Foundation (DFG) through the Program Performance Engineering for Scientific Software and the US Department of Energy under Grant No. DE-SC0015524.

REFERENCES

[1] Sergio Aldea, Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. 2016. An OpenMP Extension that Supports Thread-Level Speculation. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (January 2016), 78–91.

[2] Eduard Ayguade, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. 2009. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems* 20, 3 (March 2009), 404–418.

[3] Jairo Balart, Alejandro Duran, Eduard Gonzalez, Xavier Martorell, Eduard Ayguade, and Jesus Labarta. 2004. Nanos Mercurium: a Research Compiler for OpenMP. In *European Workshop on OpenMP (EWOMP)*. Stockholm, Sweden, 103–109.

[4] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*. Tucson, AZ, USA, 101–113.

[5] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. 2007. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press.

[6] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Proc. of Innovative Parallel Computing (InPar)*. San Jose, CA, USA, 1–10.

[7] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems: Twelve ways to tell the masses when reporting performance results. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[8] Zia Ul Huda, Rohit Atre, Ali Jannesari, and Felix Wolf. 2016. Automatic Parallel Pattern Detection in the Algorithm Structure Design Space. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS)*. Chicago, IL, USA, 43–52.

[9] Zia Ul Huda, Ali Jannesari, and Felix Wolf. 2015. Using Template Matching to Infer Parallel Design Patterns. *ACM Transactions on Architecture and Code Optimization* 11, 4, Article 64 (January 2015), 21 pages.

[10] Intel. 2019. Intel Inspector, Memory and Thread Debugger. https://software.intel.com/en-us/intel-inspector. Accessed: 2019-05-16.

[11] Alain Ketterlin and Philippe Clauss. 2012. Profiling Data-Dependence to Assist Parallelization: Framework, Scope, and Optimization. In *Proc. of the International Symposium on Microarchitecture (MICRO)*. Vancouver, B.C., Canada, 437–448.

[12] Minjang Kim, Nagesh B. Lakshminarayana, Hyesoon Kim, and Chi-Keung Luk. 2013. SD3: An Efficient Dynamic Data-Dependence Profiling Mechanism. *IEEE Trans. Comput.* 62, 12 (December 2013), 2516–2530.

[13] Feng Li, Antoniu Pop, and Albert Cohen. 2012. Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs. *IEEE Micro* 32, 4 (June 2012), 19–31.

[14] Zhen Li, Rohit Atre, Zia Ul Huda, Ali Jannesari, and Felix Wolf. 2016. Unveiling Parallelization Opportunities in Sequential Programs. *Journal of Systems and Software* 117, C (July 2016), 282–295.

[15] Zhen Li, Ali Jannesari, and Felix Wolf. 2015. An Efficient Data-Dependence Profiler for Sequential and Parallel Programs. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS)*. Hyderabad, India, 484–493.

[16] Chunhua Liao, Daniel J. Quinlan, Jeremiah J. Willcock, and Thomas Panas. 2010. Semantic-Aware Automatic Parallelization of Modern Applications Using High-Level Abstractions. *International Journal of Parallel Programming* 38, 5 (October 2010), 361–378.

[17] Xavier Martorell, Marc Gonzlez, Alex Duran, Jairo Balart, Roger Ferrer, Eduard Ayguade, and Jesus Labarta. 2006. Techniques Supporting Threadprivate in OpenMP. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS)*. Rhodes Island, Greece, 227–234.

[18] Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming*. Addison-Wesley Professional.

[19] Michael McCool, James Reinders, and Arch D. Robison. 2012. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann.

[20] Daniel J. Quinlan. 2000. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters* 10, 2/3 (October 2000), 215–226.

[21] Pedro Ramos, Gleison Mendonca, Divino Soares, Guido Araujo, and Fernando Magno Quintao Pereira. 2018. Automatic Annotation of Tasks in Structured Code. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Limassol, Cyprus, 20–33.

[22] Sara Royuela, Alejandro Duran, Chunhua Liao, and Daniel J. Quinlan. 2012. Auto-scoping for OpenMP Tasks. In *International Workshop on OpenMP (IWOMP)*.

Rome, Italy, 29–43.

[23] Sara Royuela, Roger Ferrer, Diego Caballero, and Xavier Martorell. 2015. Compiler Analysis for OpenMP Tasks Correctness. In *Proc. of the Conference on Computing Frontiers (CF)*. Ischia, Italy, 12–20.

[24] Michael Voss, Eric Chiu, Patrick Man Yan Chow, Catherine Wong, and Kevin Yuen. 2005. An Evaluation of Auto-scoping in OpenMP. In *International Workshop on OpenMP Applications and Tools: Shared Memory Parallel Programming with OpenMP (WOMPAT)*. Houston, TX, USA, 98–109.

[25] Chun-Kun Wang and Peng-Sheng Chen. 2015. Automatic Scoping of Task Clauses for OpenMP Tasking Model. *Journal of Supercomputing* 71, 3 (March 2015), 808–823.

[26] Zheng Wang, Georgios Tournavitis, Björn Franke, and Michael F. P. O'boyle. 2014. Integrating Profile-driven Parallelism Detection and Machine-learning-based Mapping. *ACM Transactions on Architecture and Code Optimization* 11, 1, Article 2 (February 2014), 26 pages.