Received: 15 March 2018 | Accepted: 27 May 2018
DOI: 10.1002/cpe.4770

WILEY

SPECIAL ISSUE PAPER

Dissecting sequential programs for parallelization—An approach based on computational units

Rohit Atre[1] | Zia Ul-Huda[1] | Felix Wolf[1] | Ali Jannesari[2]

[1]Technische Universität Darmstadt, Darmstadt, Germany
[2]Iowa State University, Ames, Iowa, USA

Correspondence
Ali Jannesari, Iowa State University, Ames, IA 50011 USA.
Email: jannesari@iastate.edu

Summary

When trying to parallelize a sequential program, programmers routinely struggle during the first step: finding out which code sections can be made to run in parallel. While identifying such code sections, most of the current parallelism discovery techniques focus on specific language constructs. In contrast, we propose to concentrate on the computations performed by a program. In our approach, a program is treated as a collection of computations communicating with one another using a number of variables. Each computation is represented as a computational unit (CU). A CU contains the inputs and outputs of a computation, and the three phases of a computation are read, compute, and write. Based on the notion of CU, which ensures that the read phase executes before the write phase, we present a unified framework to identify both loop parallelism and task parallelism in sequential programs. We conducted a range of experiments on 23 applications from four different benchmark suites. Our approach accurately identified the parallelization opportunities in benchmark applications based on comparison with their parallel versions. We have also parallelized the opportunities identified by our approach that were not implemented in the parallel versions of the benchmarks and reported the speedup.

KEYWORDS

multicore architectures, parallelism discovery, profiling, static analysis, task parallelism

1 | INTRODUCTION

Millions of legacy programs are awaiting their parallelization. The difficulties in parallelizing sequential programs exist in nearly every step of the parallelization process. So far, numerous parallel programming models[1,2] and frameworks have been proposed to ease the development of parallel code. But "which code sections to run in parallel?" is still one of the most difficult questions that developers are required to answer. Answering this question requires a deep understanding of the source code, and it can be extremely difficult when the source code has been written by someone else. Unfortunately, it is a common occurrence in large organizations. Thus, many efforts have been made to automate the parallelism discovery methods and provide tool for parallelization.

Existing parallelism discovery approaches have been built on top of data-dependence analysis, either statically[3,4] or dynamically.[5,6] The idea of using data dependences to discover parallelism is based on Bernstein's conditions.[7] Let P_i and P_j be two program sections. I_i and O_i are the sets of input and output variables of P_i. Similarly, I_j and O_j are the sets of input and output variables of P_j. P_i and P_j can be executed in parallel if

$$I_j \cap O_i = \varnothing,$$
$$I_i \cap O_j = \varnothing,$$
$$O_i \cap O_j = \varnothing.$$

Suppose P_i is executed before P_j in sequential order. Violating the first condition introduces a read-after-write (RAW) dependence, meaning P_i produces a result used by P_j. Similarly, the second condition represents a write-after-read (WAR) dependence. The last condition represents a write-after-write (WAW) dependence: P_i and P_j write to the same location, the final result comes from the logically executed last code section.

Existing methods check these three kinds of dependences to identify parallelism. However, they do not strictly follow Bernstein's conditions since not all the dependences correspond to the input and output variables. These methods do not necessarily distinguish between input and output variables of a code section. As a result, they apply data-dependence analysis on all of the variables. It not only makes parallelism discovery unnecessarily complex but also leads to both false positives and false negatives in identified parallelism of a sequential program.

In this paper, we present an approach to identify both loop parallelism and task parallelism in sequential programs following Bernstein's conditions. Our method treats a program as a set of computations with communications among them. Data dependences among computations are obtained using dynamic dependence profiling. Our approach targets both loop parallelism and task parallelism. A loop is easily parallelizable if there is no inter-iteration dependence in the loop. Such loops are called DOALL loops.[8] Loops that contain inter-iteration dependences and can still be parallelized are called DOACROSS[8] loops. In case of task parallelism, there are two kinds of tasks: tasks that are instances of the same code section but process different data (SPMD)[9] and tasks that execute completely different code sections performing different computations (MPMD).[10] Our approach helps us identify all the aforementioned kinds of parallelism following Bernstein's conditions (BC) and produces parallelization opportunities in sequential programs. Computations identified by our approach provide us with the versatility to create a framework where they can be used as a task, an iteration of a loop, a stage in a pipeline, or other parallel constructs.

We compared the parallelization opportunities found by our approach with the existing parallel versions. In addition, we also parallelized the opportunities identified as parallelizable but not parallelized in the parallel version of the benchmark applications. Our experiments on Barcelona OpenMP Tasks Suite (BOTS),[11] NAS parallel benchmarks,[12] PARSEC[13] benchmark suite, and Starbench parallel benchmark suite[14] showed that all of the code sections identified as parallelizable by our approach are parallelized in existing parallel versions.

The main contribution of this work provides a technique to identify loop and task parallelism in sequential programs. This is accomplished by

1. defining and identifying CUs that perform a unit of work in program;
2. expressing the sequential program as a graph of CUs using the dependences and the call graph;
3. analyzing the CU graph to identify the loops and the tasks that can be parallelized;
4. evaluating the tasks and the loops identified as parallelizable.

In our previous works,[15-17] we also discussed the concept of CU, but these works focus on the dynamic analysis and discussed our profiler specifically in greater detail. In our previous work,[16] an overview of our toolset DiscoPoP is provided, which includes the evaluation of the dynamic profiler with respect to its performance and memory consumption. In addition, our previous work[15] also discusses the use of CUs to identify parallel patterns in existing sequential code that is discussed in greater detail in our other works.[18,19] In our previous works,[20,21] we focus on task parallelism within larger computations and functions, and identification of CUs using a different method with Use-Definition Chain (UD Chain).

However, the contribution of this paper focuses on demonstrating further improvements made to the CU, the process of CU identification statically, and its applications in various parallelization scenarios. It does not discuss the profiler but explains how the dynamic information generated by the profiler is used with statically identified CUs to discover parallelism in a hybrid approach. Our approach for identification and use of CUs to discover specific parallelization scenarios with respect to loops and tasks is discussed in details in this paper. This work demonstrates how an improved CU clearly distinguishes the inputs and outputs of a computation using sets of variables, allowing a direct application of Bernstein's conditions. In addition, our method presented in this paper discovers both task parallelism and loop parallelism using the same framework. A CU in our current approach now acts as a task, a stage in a pipeline, or an iteration of a loop or a subset of either these based on the given context.

The rest of this paper is organized as follows. Before introducing our approach, we summarize the current state of the art and the related works in Section 2. Then, we describe our approach in Section 3, including the algorithms for building computations and discovering parallelism. Experiments, analysis, and evaluation results are presented in Section 4. Finally, Section 5 concludes this paper and discusses potential future work.

2 | RELATED WORK

Parallelism discovery has always been an interesting topic in the field of parallel programming. Early approaches analyze source code statically and predict parallelism based on theoretical models.[3,4] Bobbie[22] presented a method to partition a program for parallelization. The method adopts syntax-driven data-dependence analysis and detects parallelism based on Bernstein's conditions.[7] It uses bipartite graph matching to partition the code.

Without taking care of the runtime behavior of the target program, key parallelism usually remains undetected in static approaches. To overcome the disadvantage, methods adopting profiling techniques to gather runtime data have emerged.[5,6] Such methods are usually referred to as dynamic parallelism discovery approaches. Additionally, most of these approaches have a cost model or ranking system to produce accurate results. Kremlin[23] calculates the length of critical path in a given code region. It calculates a metric called self-parallelism using this knowledge to quantify and express the parallelism of a code section. Alchemist[24] identifies predefined code constructs that can serve as potential candidates for asynchronous execution in sequential programs. It estimates the effectiveness of parallelizing a certain construct by profiling the dependence distance using Valgrind.[25] Kremlin and Alchemist mainly focus on loops, which are easier to profile and quantify. At the same time, other dynamic parallelism discovery approaches deal with task parallelism. Parwiz[26] profiles sequential programs and represents them using execution trees. It further attaches data dependences to the nodes of the execution tree and discovers task parallelism where two or more nodes are independent of one another. MAPS[27] concentrates on parallelism discovery for applications on multiprocessor System-on-Chip (MPSoC) systems. It identifies code sections called *coupled blocks*. Each coupled block is considered as a task, and two tasks can run in parallel if there is no data dependence between them. Ottoni et al[28] propose a *Decoupled Software Pipelining*. They analyze the dependence graph and merge the strongly connected components to generate a directed acyclic graph (DAG) out of a loop. Raman et al propose a technique called *parallel stage decoupled software pipelining*,[29] which converts programs with pointer-based loops into pipelines. These two approaches specifically target loops and exploit pipeline parallelism in sequential applications. Implicitly parallel languages such as Swift/T[30] also focus on generating and executing data-driven tasks but they propose a programming model of their own.

The main issue with the aforementioned approaches is that they treat every variable and every data dependence as equally important in parallelism discovery, although it is not the case. They neglect the key information provided by Bernstein's conditions: only the dependences on input and output variables prevent parallelism. Considering all of the variables and dependences in programs makes these approaches unnecessarily complex and produces numerous false positives and false negatives. Like all the dynamic parallelism discovery approaches, our method adopts profiling techniques to gather runtime data. Unlike those approaches, our method discovers parallelism based on computational units (CUs). We identify CUs in sequential programs and build a CU graph as the representation of a program.

3 | APPROACH

We firstly introduce computational unit, which is the most important concept in our method. Then, CU graph, ie, the graph we use to represent a sequential program, is introduced. In the end, we show the algorithms for discovering task parallelism and loop parallelism.

3.1 | Computational unit

A *computational unit* is a collection of instructions that follow the *read-compute-write* pattern: firstly, a collection of instructions read a set of variables, this set is used to perform a computation, and then the result is written back to another set of variables. These two sets of variables are called *read set* and *write set*, respectively. These two sets do not necessarily have to be disjoint. The *read phase* of the CU contains load instructions reading the variables in the read set, and the *write phase* of the CU contains store instructions writing variables in the write set.

Tasks communicate with one another by reading and writing variables that are global to them and perform the computations necessary for this communication locally. Hence, a CU is defined by read-compute-write pattern where the variables in a CU's read set and the write set are required to be global to the CU. The variables local to the CU will not be used to communicate with other tasks and do not affect parallelization. Hence, they are part of the compute phase of the CU. Variable scope analysis is used to distinguish variables that are global to a code section. It is to be noted that the variables considered global in the read set and the write set do not have to be global in scope to the entire program. These variables can be local to the code section encapsulating the target code but global in scope to the target code section.

CUs are built using static analysis for every *region*. A region is a single entry, single exit code block. It is a connected subgraph of a control flow graph that has exactly two connections to the remaining graph. A region could be a group of basic blocks with branches inside or it can be a function, a loop, an if-else structure, or a basic block. In practice, a basic block rarely contains noteworthy parallelism because it usually contains a small number of instructions. Codes in different branches of an if-else structure are semantically exclusive, thus rarely run in parallel. Hence, in our approach, we mainly focus on regions like functions and loops, which contain important computations that can potentially run in parallel.

As mentioned earlier, our earlier implementations of CU focused on computations identified through def-use chains for task parallelism[20] within functions and larger computations. CUs produced by such method were non-contiguous lines of code. It was also observed that such parallelism would require code modification and transformation, and it would not be easily scalable or load balanced. In contrast, this work focuses on identifying and improving CUs as contiguous code units that can serve as building blocks for different kinds of parallelism within a common framework.

3.1.1 | Cautious property

A code section is only considered to be a CU if it is cautious. Cautious property[31] was previously defined for operators in unordered algorithms: an operator is said to be cautious if it reads all the elements of its neighborhood before it modifies any of them. By adapting the cautious property to the CUs, we consider a code section to be cautious if each variable in its read set is read before that variable is written in the write phase of the CU. Cautious property provides a clear way of separating the read phase and the write phase. Hence, read-compute-write pattern of the CU is guaranteed by the cautious property. After parallelism discovery, it also allows multiple CUs to be run speculatively without buffering updates or making backup copies of modified data, since all conflicts are detected during the read phase and before the write phase. It also means that the tasks extracted based on CUs do not need to have any special requirement on runtime frameworks.

Algorithm 1 shows the algorithm of constructing CUs statically. For each region, all the variables global to that region are identified. These variables are classified into read set (inputs) and the write set (outputs) of the CU. The read set and the write set are used to build the read phase and the write phase. Then, we check if the cautious property is satisfied by read phase and write phase. If so, the target region is recognized as a computational unit.

Algorithm 1 Algorithm of building CUs.

```
for each region R in the program do
globalVars = variables that are global to R
isCautious = true
  for each variable v in globalVars do
    if v is read then readSet += v
      for each instruction Irv reading v do readPhase += Irv
      if v is written then writeSet += v
        for each instruction Iwv writing v do writePhase += Iwv
    for each variable v in readSet do
      for each instruction Ir reading v do
        for each instruction Iw writing v do
          if Ir happens after Iw then isCautious = false
      if isCautious then cu = new CU(R,readSet,writeSet,readPhase,writePhase)
      else
        for each read instruction Iv violating cautious property do build CU for instructions do not belong to any CU before Iv
```

A variable can sometimes be written and never read. Such a case does not violate the cautious property of the CU. Cautious property is violated only when a global variable is firstly written and then read. The first read instruction that happens after a write in that case is called a cautiousness violating point. A region can be not cautious. It means that it can contain more than one computational unit. When a region is not cautious, we find the cautiousness violating points in the region by identifying the first read after a write and then break the region into multiple code sections based on the cautiousness violating points. We then construct CU for each of these code sections within the region. The end result is such that the region that is not cautious contains multiple CUs. This process is shown in the last else branch in Algorithm 1.

Figure 1 shows an example of building CU for a loop. This example also demonstrates that our improvements on CU also keep its definition consistent with our previous work. In this example, *readSet* and *writeSet* both contain the variable {x}. Each loop iteration firstly reads the old value of x, then computes a new value via local variables a and b, and then calculates a new value of x. Finally, the new value is written back to x. This follows the read-compute-write pattern. For a single iteration of the loop, the loop region is cautious. All the reads of variable x happen before the write to x. Lines 3-5 follow the read-compute-write pattern and hence are in one CU. At source-line level, the compute phase of the CU is in lines 3-5. This overlaps with its read phase that is in lines 3-4 and the write phase that is on line 5. At instruction level, the three phases are obviously separate from one another. If variables a and b were declared outside the loop, then they would be considered global to the loop as well, according to the definition of CU. This would also mean that the loop would have cautiousness violating point and it would be made up of two CUs with lines 3-4 being one CU and the line 5 being the second CU.

Note that CUs never cross region boundaries. This is necessary; otherwise, a CU could grow too large. Such a CU could possibly swallow iterations of a loop and other code sections and hide important parallelism that we actually want to discover. Fine-grained CUs, however, can be grouped together to form coarse tasks if necessary, which we discuss in our previous work.[20]

3.2 CU graph

A computation may depend on data produced from other computations. To represent such dependences, we use a dynamic dependence profiler DiscoPoP.[15] DiscoPoP profiles detailed data dependences, gathers control-flow information, and identifies hotspots across the target program. To overcome the input sensitivity of the dynamic dependence analysis, we run the profiler multiple times using representative inputs and merge the dependence results obtained. Then, we build a CU graph, in which vertices are statically generated CUs and edges are dynamic data dependences. Hence, the CU graph combines static and dynamic information to help us discover parallelism. Data dependences in a CU graph are always among instructions in read phases and write phases of the CUs. Dependences that are local to a CU (within a CU) are hidden in the CU graph because they do not prevent parallelism among CUs, according to Bernstein's conditions. Moreover, since the number of global variables to a code section is usually far less than the number of local variables, a CU graph is much simpler than the traditional instruction-based dependence graph. This simplified the parallelism discovery process. The CU graph is then expanded using runtime information to represent instances of tasks or loops.

```
int x = 3;
for (int i = 0; i < MAX_ITER; ++i) {
    int a = x + rand() / x;
    int b = x - rand() / x;
    x = a + b;
}
```

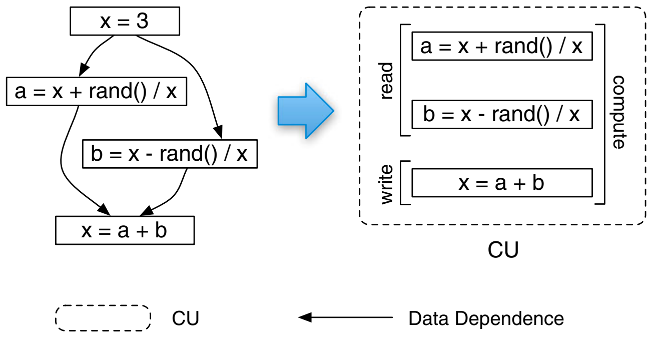

FIGURE 1 Building a CU

3.3 | Parallelism discovery

There are two kinds of parallelism that we can identify: parallelism among different computations and parallelism among different instances of the same computation. Parallelism among different computations can be easily identified using CU graphs without instantiating these computations. On the other hand, identifying parallelism among different instances of the same computation requires some additional effort. To discover such parallelism, a CU must be instantiated using real inputs passed into the computation and real outputs it produces. We call a CU graph with its nodes instantiated as an *expanded CU graph*.

For example, consider the following function:

`int foo(int x, const Widget &w).`

When building a CU for `foo`, read set contains `x` and `w` and write set contains the virtual return variable `ret`. Given the following calls to `foo`:

`a = foo(width1, window1);`

`b = foo(width2, window2).`

The CU of foo is instantiated into two instances: the first instance with inputs `width1`, `window1`, and output `a`, and the second instance with inputs `width2`, `window2`, and output `b`. They replace the formal parameters `x`, `w`, and the virtual return value `ret`, respectively. We call a CU graph with its nodes instantiated as an *expanded CU graph*. In the end, data dependences must be checked among instances of a CU based on the expanded CU graph in order to detect parallelism among instances. In case of functions, multiple function calls of the same function are instantiated with inputs instead of formal parameters. In case of loops, every iteration implicitly instantiates the CUs that belong to the loop and the separate instances of CUs can be represented in the expanded CU graph.

3.3.1 | Task parallelism

Figure 2 shows the flowchart of our task parallelism discovery process. The algorithm only considers CUs that are hotspots in terms of execution time and discovers task parallelism based on the following rules.

1. A CU is instantiated into different instances using their real inputs and outputs with respect to the control flow. Two instances of the same computation can run in parallel if they are independent in the expanded CU graph (SPMD task parallelism).
2. Two different computations can run in parallel if their corresponding CUs are independent in the CU graph (MPMD task parallelism).

SPMD task parallelism: To better illustrate the second rule, we show a parallelization opportunity found in *fib*, an application from Barcelona OpenMP Task Suite, which produces the *n*th number in the Fibonacci series. Figure 3 shows the CU graph of *fib*. CUs 67-1 and 67-2 belong to the function `fib0()`, which is not cautious in itself and hence is divided in two CUs and connected by a RAW dependence. CU 67-0 represents the function `fib()` and is data dependent on CU 67-2 and itself. As the CU graph alone does not reveal any parallelism here, we produce the expanded CU graph (shown in Figure 4). In Figure 4, the CU of function `fib()` is instantiated into multiple instances. The self-dependence of CU 67-0 in the CU graph is due to the dependence between current `fib()` call and its subsequent recursive calls, as shown in the expanded CU graph. The instances of `fib()` that are on the same recursion level satisfy Bernstein's conditions. These calls are identified to take up approximately 100% of the execution time. This makes `fib()` a definite candidate for task parallelism.

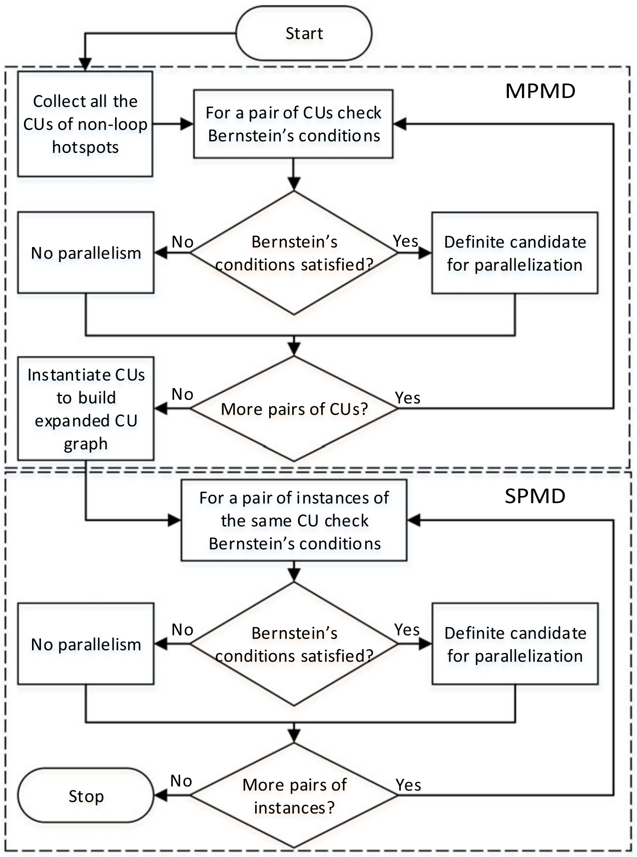
Start
MPMD
Collect all the CUs of non-loop hotspots
For a pair of CUs check Bernstein's conditions
No parallelism
No
Bernstein's conditions satisfied?
Yes
Definite candidate for parallelization
Instantiate CUs to build expanded CU graph
No
More pairs of CUs?
Yes
SPMD
For a pair of instances of the same CU check Bernstein's conditions
No parallelism
No
Bernstein's conditions satisfied?
Yes
Definite candidate for parallelization
Stop
No
More pairs of instances?
Yes

FIGURE 2 Flowchart Showing the Criteria for a Finding Task Parallelism Candidates

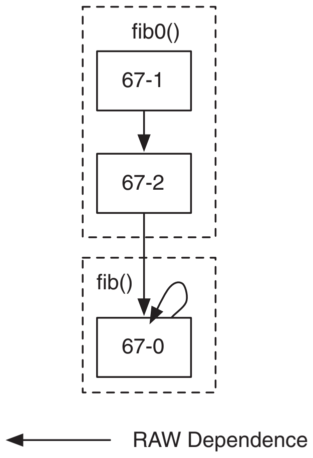

FIGURE 3 CU Graph of *fib* Benchmark From BOTS for Task Parallelism

MPMD task parallelism: In *Fluidanimate* application of PARSEC, we found an opportunity for MPMD task parallelism in the function `RebuildGrid`, which has two different computations identified as two independent CUs. The Courant-Friedrichs-Lewy (CFL) condition check is performed by one of the CUs and the other CU represents the rest of the function. These two CUs can run in parallel as they do not have any dependences between them and they satisfy Bernstein's conditions. This type of parallelization can be easily implemented using constructs like OpenMP sections.[1] Parallelism within a computation is not covered in this paper. However, such parallelism can be detected by applying techniques that tracks def-use chains[20] on compute phases of CUs.

3.3.2 | Loop parallelism

For loop parallelism discovery, every iteration of the loop instantiates every CU within the loop. The expanded CU graph is used to check if the iterations of a loop are independent or if the loop has inter-iteration dependences. An inter-iteration dependence is either a dependence between

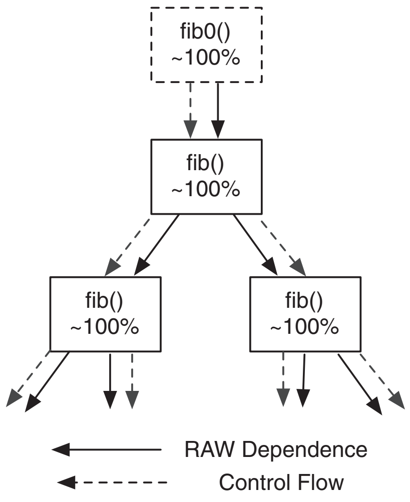

FIGURE 4 Expanded CU Graph of *fib*

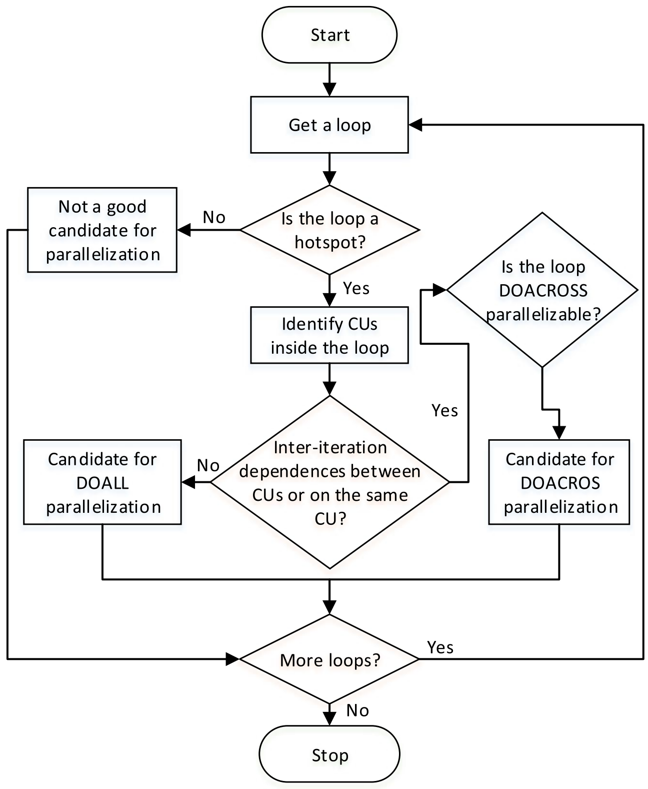
Start
Get a loop
Is the loop a hotspot?
No
Not a good candidate for parallelization
Yes
Identify CUs inside the loop
Is the loop DOACROSS parallelizable?
Yes
Inter-iteration dependences between CUs or on the same CU?
No
Candidate for DOALL parallelization
Candidate for DOACROS parallelization
More loops?
Yes
No
Stop

FIGURE 5 Flowchart Showing the Criteria for Finding Loop Parallelism Candidates

two separate instances of CUs or a self-dependence on a CU. Figure 5 shows the flowchart of our loop parallelism discovery process. Similar to task parallelism discovery, we only consider loops that are hotspots in terms of execution time and discover loop parallelism based on the following rules.

- Iterations of a loop can run in parallel if for all CUs built for the loop, there are no inter-iteration RAW dependences among the CUs or on a single CU (DOALL parallelism).
- If there are inter-iteration dependences in the loop, the loop may still be analyzed to check if it can be parallelized by using techniques, eg, reduction, privatization, and pipeline (DOACROSS parallelism).

DOALL parallelism: In the case of DOALL loops, the iterations of loop are independent of each other. This means that the instances of CUs are also independent and as a result they satisfy Bernstein's conditions. However, it is not necessary to analyze all the instances of the CUs of a loop because loops have no parameters to be replaced. To better illustrate the rule for discovering DOALL parallelism, we consider a loop identified as parallelizable in *IS* from NPB. Figure 6 shows the CU graph of this loop. Loop iteration variables are considered local unless they are modified in the

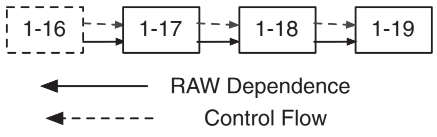

FIGURE 6 CU Graph of *IS* From NPB (DOALL)

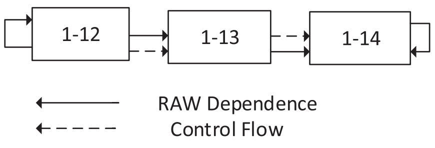

FIGURE 7 CU Graph of Function `processImage()` of *rgbyuv* from Starbench Showing a Pipeline (DOACROSS)

loop body. Multiple CUs are identified inside the loop body. The graph shows that there are three RAW dependences between the remaining CUs but none of them are inter-iteration dependences. As a result, the iterations of the loop can run in parallel.

DOACROSS parallelism: Dependence distances of the inter-iteration RAW dependences can be further analyzed to discover DOACROSS[8] loops. A DOACROSS loop has inter-iteration dependences, but the dependence distance should not be as large as the distance between the first line of an iteration and the last line of the previous iteration. In other words, the first CU of the loop should not depend on the last CU of the loop. That means iterations of a DOACROSS loop can overlap, thus containing parallelism. Based on our CU graph, if the length of the longest dependence is smaller than the distance from the last CU to the first CU, we could classify such loops as candidates for DOACROSS parallelism. As there are inter-iteration dependences in DOACROSS loops, all the instances of CUs are not independent of each other like in DOALL. But a subset of CUs in an iteration is independent of another subset of CUs in the next iteration, and hence, they satisfy Bernstein's conditions and can be run in parallel. Utilizing such parallelism is usually achieved by applying the pipeline pattern. For example, consider the function `processImage()` in *rgbyuv* application of STARBENCH benchmark suite. Figure 7 shows the simplified CU graph generated for the inner loop in the function `processImage()`. The CUs 1-12 and 1-14 have self-dependences indicating that there are inter-iteration dependences in the loop. As a result, this loop cannot be parallelized using DOALL parallelism directly, and as the first CU in the loop with respect to sequential execution order is not dependent on the last CU, this loop can be considered a valid candidate for DOACROSS parallelism. The three CUs in Figure 7 represent the three stages of a pipeline: input stage; computation of *Y*, *U*, *V* using *R*, *G*, *B*; and the output stage. This example demonstrates that a CU can also act as a stage in a pipeline.

4 | EVALUATION

We conducted a range of experiments to evaluate the effectiveness of our approach. We applied our method on benchmarks in Barcelona OpenMP Task Suite (BOTS),[11] PARSEC benchmark,[13] NAS Parallel Benchmarks (NPB),[12] and Starbench benchmark[14] to evaluate our task and loop parallelism discovery. All four benchmarks contain sequential benchmark applications and their equivalent parallel versions. After applying our method on the sequential benchmark applications, we compare the identified parallelization opportunities to the existing parallel versions in order to evaluate our approach. For the opportunities that do not have corresponding parallel version, we implemented our own parallel version for these applications. Benchmarks evaluated belong to various domains and confirm that our approach can identify parallelism in different applications. The evaluated benchmarks include small and middle-sized applications. The generated CU graphs for the hotspots are in a reasonable size to be analyzed and traced by the developers. Obviously our approach needs further improvements when dealing with larger applications. A direction of our future works will be focused on the scalability of our approach concerning parallelism targeting larger applications on larger size machines.

Currently, we focused on major parallelism exploitation opportunities (hotspots) by defining a threshold for identifying hotspots based on execution time. The user could also analyzes cases for minor (non-hotspot) parallelism exploitation opportunities by lowering the preset threshold values.

Our approach is implemented in LLVM 3.6.1,[32] and all benchmarks are compiled using Clang 3.6.1.[33] Experiments were run on a server with 2×8-core Intel Xeon E5-2650, 2-GHz processors with 32-GB memory, running Ubuntu (64-bit server edition). The performance results reported are an average five independent executions.

4.1 | Task parallelism

SPMD: Table 1 shows the results of discovering task parallelism on benchmarks of BOTS. Twenty hotspots were analyzed in all benchmarks of BOTS, and 12 of them contain SPMD parallelism as shown in Table 1. Comparing the hotspots that are identified as parallelizable (shown in column *Parallelizable*) with their parallel versions (in column *Implemented*), it was observed that all of them are parallelized using master-worker (MW) pattern.

TABLE 1 SPMD task parallelism in BOTS

App.	Function	Parallelizable	Implemented	Exec. Time (%)	# Tasks	# Dep.
sort	cilkmerge	✓	MW	34.4	1	47
	cilksort	✓	MW	74.8	2	56
	seqquick	✗	✗	22.6	-	-
	seqmerge	✗	✗	52.0	-	-
	sort	✗	✗	74.9	-	-
fib	fib	✓	MW	~100	1	0
fft	fft	✗	✗	~100	-	-
	fft_aux	✓	MW	97.2	1	0
	fft_twiddle_16	✓	MW	83.0	1	0
	fft_unshuffle_16	✓	MW	12.7	1	0
floorplan	add_cell	✓	MW	~100	1	0
health	sim_village	✓	MW	~100	1	0
sparselu	sparselu	✓	MW	34.4	1	8
	bmod	✗	✗	89.6	-	-
strassen	strassen_main	✗	✗	95.2	-	-
	OptimizedStrassenMultiply	✓	MW	95.2	1	105
	MultiplyByDivideAndConquer	✓	MW	82.0	1	0
	FastNaiveMatrixMultiply	✗	✗	21.4	-	-
	FastAdditiveNaiveMatrixMultiply	✗	✗	61.9	-	-
uts	serTreeSearch	✓	MW	99.6	1	0

TABLE 2 MPMD task parallelism in PARSEC

App.	Function	CR	# CUs	LS	# Dep.	# Dep. Resolved
Fluidanimate	RebuildGrid	Yes	2	1.60	300	16
Fluidanimate	ProcessCollisions	No	6	1.81	121	0
Canneal	routing_cost_given_loc	Yes	2	1.32	19	2
Blackscholes	CNDF	No	2	0.98	38	0
Fluidanimate	ComputeForces	Yes	3	1.52	32	6

It can also be observed that eight hotspots were not identified as parallelizable in the benchmark suite. The CUs for these functions do not fulfill Bernstein's conditions in either CU graphs or expanded CU graphs. We verified our results by observing the parallel versions of the respective benchmarks. The last column in Table 1 lists percentage of the execution time, number of tasks identified, and number of dependencies for each hotspot.

MPMD: MPMD opportunities in PARSEC were similar to `RebuildGrid`, which was discussed in Section 3.3.1. All the cases were parallelized manually by us with fork-join model using OpenMP *section* or *task* directives. Speedups reported in Table 2 refer to the ratio of execution time of the tasks run in parallel to the execution time of the sequential version of the same code region (function). As a result, it is called *local speedup* (LS in Table 2). Refactoring the code mainly involved privatization of variables (eg, adding OpenMP private clauses), adding necessary synchronization (eg, using critical sections in OpenMP), or replicating some part of the code across multiple threads (eg, copying the same data between threads/sections or having shared data with synchronization operations between threads).

Table 2 also provides characteristics obtained from the CU graphs of sequential versions. We also report the number of dependences resolved to parallelize the identified opportunity. These dependences were resolved by refactoring the code (CR for Code Refactoring in Table 2), privatizing or sharing the variables, etc to expose more parallelism. It can be seen that the number of resolved dependences is very small compared to the total number of dependences. This clearly shows that the number of dependences does not represent the difficulty encountered during parallelization.

4.2 | Loop parallelism

DOALL: Applying loop parallelism discovery algorithm to the applications of NPB revealed the information in Table 3. It contains a list of applications and the loops that were analyzed for finding parallelism. A total of 25 hotspot loops were analyzed in all the benchmarks of NPB and are identified as parallelizable (shown in column *Parallelizable*)using DOALL. All of these identified loops are already parallelized in their respective parallel versions using DOALL (shown in column *Parallelized*).

TABLE 3 DOALL parallelism in NPB

App.	Loop (File-Line)	Parallelizable	Parallelized	Exec. Time (%)	# Dep.
IS	is.c-372	✓	DOALL	61.1	4
EP	ep.c-167	✓	DOALL	∼100	7
BT	x_solve.c-70	✓	DOALL	29.9	0
	y_solve.c-69	✓	DOALL	30	0
	z_solve.c-69	✓	DOALL	30	0
SP	x_solve.c-48	✓	DOALL	29.9	0
	y_solve.c-48	✓	DOALL	30	0
	z_solve.c-48	✓	DOALL	30	0
FT	auxfnct.c-163	✓	DOALL	27.2	2
	fft3d.c-81	✓	DOALL	25.7	23
	fft3d.c-114	✓	DOALL	99.1	9
	fft3d.c-156	✓	DOALL	23.2	9
	fft3d.c-181	✓	DOALL	19.6	9
	fft3d.c-196	✓	DOALL	21.8	9
CG	cg.c-296	✓	DOALL	98.3	9
	cg.c-458	✓	DOALL	98.2	4
MG	mg.c-550	✓	DOALL	40.6	1
	mg.c-951	✓	DOALL	23	3
	mg.c-488	✓	DOALL	14.6	
	mg.c-695	✓	DOALL	6.6	23
LU	ssor.c-115	✓	DOALL	56.2	0
	setiv.c-53	✓	DOALL	43.5	5
	buts.c-75	✓	DOALL	11.6	3
	blts.c-80	✓	DOALL	10.6	3
	jacu.c-54	✓	DOALL	8.7	2

TABLE 4 Loops with inter-iteration dependences

App.	Exec. Time (%)	DOACROSS	Implemented	# CUs
rgbyuv	99.9	✓	Pipeline	3
tinyjpeg	99.9	✓	Pipeline	2
kmeans	99.5	✓	Pipeline	4
nqueens	∼100	✓	Reduction	1
CG	96.9	✓	Reduction	4
BT	99.1	✗	-	-
SP	99.1	✗	-	-
FT	49.3	✗	-	-
CG	98.4	✗	-	-
MG	49.8	✗	-	-

DOACROSS: Table 4 shows the loops with inter-iteration dependences. If a loop's first CU with respect to serial execution order is dependent on its last CU, then the loop requires a sequential execution and cannot be parallelized. Such results were found in NPB in applications *CG* (one of the opportunities), *BT, SP, FT,* and *MG*. We identified DOACROSS parallelism in three cases, *rgbyuv, tinyjpeg*, and *kmeans* from Starbench. These were parallelized using a pipeline where each CU is considered a stage in the pipeline. Our parallel implementation for *rgbyuv* led to a maximum speedup of 2.29 with four threads. We verified that the DOACROSS parallelism identified in *tinyjpeg* and *kmeans* is already implemented using pipelines in their parallel versions. We also found two DOACROSS parallelism cases in *nqueens* of BOTS and *CG* of NPB. These were implemented using reduction in the parallel versions of their respective benchmarks.

5 | CONCLUSION

This paper discusses an approach to identify code sections called computational units (CU) in sequential programs. A CU follows a read-compute-write pattern. We use cautious property to detect CUs statically from the source code. A CU graph consisting of CUs and dependences between them is used as basis for parallelism detection. We use Bernstein condition's to identify the tasks that can run in parallel to each other from a CU graph or an expanded CU graph. For loop parallelism, CUs are used to focus on identifying DOALL and DOACROSS loops by analyzing the inter-iteration dependences that may prevent parallelization of the loop. We evaluated our approach by analyzing applications from BOTS, NPB, PARSEC, and Starbench benchmark suites. We identified 12 task parallelization opportunities in BOTS for SPMD task parallelism and all 12 opportunities have been implemented in parallel version. We identified 25 DOALL loops that are hotspots in NPB and all the loops have been parallelized. For MPMD task parallelism, we implemented the opportunities identified and reported the speedup. We also analyzed loops with inter-iteration dependences to discover DOACROSS loops and implemented their parallel version wherever necessary. We show that the count of dependences and parallel tasks in a CU graph do not necessarily make the parallelization process more difficult. In the future, we would like to quantify the effort required to parallelize the opportunities discovered by our approach using some metrics. Additionally, CU graph can be made more versatile by mapping it onto parallel constructs like TBB[2] Flow Graph to automatically generate parallel code.

ORCID

Rohit Atre http://orcid.org/0000-0003-0979-5308
Ali Jannesari http://orcid.org/0000-0001-8672-5317

REFERENCES

1. OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.0*. Champaign, IL: OpenMP Architecture Review Board; 2008.
2. Reinders J. *Intel Threading Building Blocks*. 1st ed. Sebastopol, CA: O'Reilly & Associates; 2007.
3. Burke M, Cytron R, Ferrante J, Hsieh W. Automatic generation of nested, fork-join parallelism. *J Supercomput*. 1989;3(2):71-88.
4. Sarkar V. Automatic partitioning of a program dependence graph into parallel tasks. *IBM J Res Dev*. 1991;35(5.6):779-804.
5. Rul S, Vandierendonck H, De Bosschere K. A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Comput*. 2010;36(9):531-551.
6. Huang J, Jablin TB, Beard SR, Johnson NP, August DI. Automatically exploiting cross-invocation parallelism using runtime information. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '13); 2013; Washington, DC.
7. Bernstein AJ. Analysis of programs for parallel processing. *IEEE Trans Electron Comput*. 1966;EC-15(5):757-763.
8. Kennedy K, Allen R. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Burlington, MA: Morgan Kaufmann Publishers; 2002.
9. Darema F. The SPMD model: past, present and future. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 8th European PVM/MPI Users' Group Meeting Santorini/Thera, Greece, September 23-26, 2001 Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2001:1.
10. Wilkinson B, Allen M. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Upper Saddle River, NJ: Prentice Hall; 1999.
11. Duran A, Teruel X, Ferrer R, Martorell X, Ayguade E. Barcelona OpenMP tasks suite: a set of benchmarks targeting the exploitation of task parallelism in OpenMP. Paper presented at: 2009 International Conference on Parallel Processing; 2009; Vienna, Austria.
12. Bailey DH, Barszcz E, Barton JT, et al. The NAS parallel benchmarks. *Int J High Perform Comput Appl*. 1991;5(3):63-73.
13. Bienia C, Kumar S, Singh JP, Li K. The PARSEC benchmark suite: Characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08); 2008; Toronto, Canada.
14. Andersch M, Juurlink B, Chi CC. A benchmark suite for evaluating parallel programming models. In: Proceedings of Workshop on Parallel Systems and Algorithms (PARS); 2011; Rüschlikon, Switzerland.
15. Li Z, Jannesari A, Wolf F. An efficient data-dependence profiler for sequential and parallel programs. Paper presented at: 2015 IEEE International Parallel and Distributed Processing Symposium; 2015; Hyderabad, India.
16. Li Z, Atre R, Ul-Huda Z, Jannesari A, Wolf F. Unveiling parallelization opportunities in sequential programs. *J Syst Softw*. 2016;117:282-295.
17. Li Z. *Discovery of Potential Parallelism in Sequential Programs* [PhD thesis]. Darmstadt, Germany: Technische Universität Darmstadt; 2016.
18. Ul-Huda Z, Jannesari A, Wolf F. Using template matching to infer parallel design patterns. *ACM Trans Archit Code Optim*. 2015;11(4):1-21.
19. Ul-Huda Z, Atre R, Jannesari A, Wolf F. Automatic parallel pattern detection in the algorithm structure design space. Paper presented at: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS); 2016; Chicago, IL.
20. Atre R, Jannesari A, Wolf F. The basic building blocks of parallel tasks. In: Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores (COSMIC '15); 2015; San Francisco, CA.
21. Atre R, Jannesari A, Wolf F. Brief announcement: Meeting the challenges of parallelizing sequential programs. In: Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '17); 2017; Washington, DC.
22. Bobbie PO. Partitioning programs for parallel execution: a case study in the Intel iPSC/2 environment. *Int J Mini Microcomput*. 1997;19(2):84-96.
23. Garcia S, Jeon D, Louie CM, Taylor MB. Kremlin: rethinking and rebooting gprof for the multicore age. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11); 2011; San Jose, CA.
24. Zhang X, Navabi A, Jagannathan S. Alchemist: A transparent dependence distance profiling infrastructure. Paper presented at: 2009 International Symposium on Code Generation and Optimization; 2009; Seattle, WA.
25. Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07); 2007; San Diego, CA.
26. Ketterlin A, Clauss P. Profiling data-dependence to assist parallelization: framework, scope, and optimization. In: Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'45); 2012; Vancouver, Canada.
27. Ceng J, Castrillon J, Sheng W, et al. MAPS: An integrated framework for MPSoC application parallelization. Paper presented at: 2008 45th ACM/IEEE Design Automation Conference; 2008; Anaheim, CA.

28. Ottoni G, Rangan R, Stoler A, August DI. Automatic thread extraction with decoupled software pipelining. In: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38); 2005; Barcelona, Spain.

29. Raman E, Ottoni G, Raman A, Bridges MJ, August DI. Parallel-stage decoupled software pipelining. In: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '08); 2008; Boston, MA.

30. Armstrong TG, Wozniak JM, Wilde M, Foster IT. Compiler techniques for massively scalable implicit task parallelism. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14); 2014; New Orleans, LA.

31. Pingali K, Nguyen D, Kulkarni M, et al. The tao of parallelism in algorithms. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11); 2011; San Jose, CA.

32. Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO '04); 2004; Palo Alto, CA.

33. Lattner C. LLVM and Clang: Next generation compiler technology. Paper presented at: BSDCan - The BSD Conference; 2008; Ottawa, Canada.

How to cite this article: Atre R, Ul-Huda Z, Wolf F, Jannesari A. Dissecting sequential programs for parallelization—An approach based on computational units. *Concurrency Computat Pract Exper*. 2018;e4770. https://doi.org/10.1002/cpe.4770