jet INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2024, VOL. 70, NO. 1, PP. 135-144
Manuscript received December 3, 2023; revised March, 2024. doi: 10.24425/ijet.2024.149523

Adding parallelism to sequential programs – a combined method

Wiktor B. Daszczuk, Denny B. Czejdo, and Wojciech Grześkowiak

Abstract—**The article outlines a contemporary method for creating software for multi-processor computers. It describes the identification of parallelizable sequential code structures. Three structures were found and then carefully examined. The algorithms used to determine whether or not certain parts of code may be parallelized result from static analysis. The techniques demonstrate how, if possible, existing sequential structures might be transformed into parallel-running programs. A dynamic evaluation is also a part of our process, and it can be used to assess the efficiency of the parallel programs that are developed. As a tool for sequential programs, the algorithms have been implemented in C#. All proposed methods were discussed using a common benchmark.**

Keywords—**programming languages; algorithms; concurrency; parallelism; parallel programming**

I. Introduction

THE processor development is changing. Multi-core architectures were developed instead of trying to continuously increase clock speeds, which completely changed the market for personal computers. However, many programmers continue to create sequential solutions for single-processor machines, ignoring the fact that a typical computer, even a laptop, is equipped with multiple computing units. As an effect, the application performance is similar regardless of the number of processors running within a computer, leaving most processors idle. Deep expertise is necessary to create parallel solutions, and many businesses find it time-consuming to deploy them. To help developers learn parallel processing, identify potential dangers, and help address them, practical assistance is required. While many existing application development frameworks support easy multithreading programming, there are neither integrated solutions that concentrate on both methodology and tools nor services that convert already-existing applications to new settings. Furthermore, there is a need for educational tools that would benefit both students and inexperienced cooperating engineers.

As a result, the main objective of this project was to provide a methodology and tools to help programmers automatically transform current sequential applications into parallel ones. The provision of useful information for related academic and professional activity was the other objective. We have considered and tested two solutions: a Native Approach and the.NET Framework (Parallel Extensions [1]). C# programs can use our technique and experimental tool [2]. The techniques, however, can be used with a broad class of high-level languages based on C syntax, such as C, C++, C#, and also VB. The following is a list of the contributions made by this article:

1. Automated identification of sequential parts of a program that can be subject to parallelization.
2. Algorithms for parallelization of sequential code.
3. A methodology for dynamic analysis of the performance of parallelized programs to confirm their increased efficiency.
4. Creation of procedures for evaluating the proposed parallelization dynamic analysis and recommending its adoption.
5. Case study of mergesort C# code and various types of its parallelization.

The literature overview, which covers the present situation and prospects for code parallelization using ML models, is found in Section II. The Parallel Extensions [1] tool for parallelization is discussed in Section III. The methodology and algorithms are described in Section IV. Common synchronization factors for suggested solutions are covered in Section V. Section VI covers a benchmark for applying all three methods. The article is concluded in Section VII.

II. Current Work

It is usually necessary to divide an issue into smaller, as independent of one another as possible subtasks in order to solve them in parallel and speed up their execution. The significant effort is required to synchronize dependent tasks. The literature has long discussed both the theoretical and practical parallel programming methods. Bernstein [3] provided one of the earliest explorations of parallel processing programs. He identified three key requirements for the independent and simultaneous execution of two instruction sequences. Amdahl investigated the theory of acceleration constraints in [4]. Later, Gustafson extended the Amdahl law [5] to situations involving significant parallelism. Numerous academics who were interested in the precise construction and functionality of massively parallel computers looked after this basic study [6].

Another significant advancement was the addition of Parallel Extensions [1], a framework for a parallel runtime environment, to the.NET Framework. It includes a number of useful syntactic features, classes, and properties that aid in building highly scalable parallel applications and enable academics and practitioners to generate fresh solutions based on this

First and third Authors are with Warsaw University of Technology, Institute of Computer Science, Warsaw, Poland (e-mail: wiktor.daszczuk@pw.edu.pl; w.grzeskowiak@gmail.com).

Second Author is with Fayetteville State University, Department of Mathematics and Computer Science, Fayetteville, USA (e-mail: bczejdo@uncfsu.edu).

© The Author(s). This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY 4.0, https://creativecommons.org/licenses/by/4.0/), which permits use, distribution, and reproduction in any medium, provided that the Article is properly cited.

environment. The alternative solution is OpenMP (Open Multi-Processing [7]), which contains a set of libraries, and generates compiler directives and environment variables that influence the execution of a parallel program. This guarantees the portability of the resulting application.

Building an "optimal" parallel program remains a difficulty even with current frameworks. Programmers frequently have trouble figuring out which parts of the code may be made to execute simultaneously [8]. The requirement for tools to aid in the development of parallel programs is well acknowledged [8]. The development of methods to identify fragments of sequential programs subject to parallelization has been the subject of extensive research. Described methods concern static analysis [9], dynamic evaluation [10], or a combination of them [9]. The maximum amount of task parallelism can be captured by static analysis using a control dependence graph [11][12], and dynamic analysis can be used to assess the efficiency of parallelized programs [10]. Zhong et al. focuses on simply tackling particular sides of the issue, such as description of how to find loops that can be parallelized in sequential programs [13]. More specific results concern identification of loops subject to parallelization by finding the optimal affine partitions [14]. Some methods focus on the numerous sequential code segments that can be parallelized [15], while others stress task-based, fine-grained parallelism [16]. Most tools depend on the language, frequently C++ [12]. Since most tools cannot automatically introduce parallelism to programs with complicated control and data flows, some methods focus on profiling information, to help the user in finding fragments susceptible to parallelization [17].

A different strategy is to create a new compiler based on the generated dependency network. It rewrites a program in a parallel task manner [18]. Another approach falls under the category of speculative strategies. This research is founded on the idea that we can execute consecutive iterations of a sequential loop concurrently, and it is unlikely that multiple data dependencies will arise during runtime [19]. If this assumption holds, it leads to achieving runtime parallelism [19]. In [20], which employs a similar approach, the state of speculative parallel threads is kept separate from the state of non-speculative computation.

III. Framework Used for Parallelization

A. Parallel Extensions

Selecting the translation destination, whether a framework designed for parallel applications or a native approach, represents a pivotal choice when transforming sequential programs into parallel ones. We have contemplated and experimented with two options: Parallel Extensions for the.NET Framework and a Native Approach.

The Microsoft-developed and promoted.NET Framework runtime environment includes Parallel Extensions [1]. Numerous useful syntactic elements, classes, and properties are added by the extension to aid in the development of parallel applications. The environment itself is also quite scalable, enabling full utilization of various multi-core machine types. Languages that support the .NET Framework 4.0 (or later), such as C++, C#, and VB.NET, can use Parallel Extensions.

A task, an independent functional component of the program that can run in parallel with other tasks, is the main element of the program. Correct and efficient job allocation is managed by the environment, which controls resources (processors). To achieve this, a unique planner (scheduler) was created to balance the load distribution throughout the running computing units and optimize job allocation. The task-stealing method makes sure that tasks are carried out by all available processors.

Every processor in the environment has a thread that is started with its own task queue. Requests distribute jobs from the application-filled thread queues in the global queue. Subsequent tasks that are created by a task are added to the top of the thread queue. Data caching is connected to this approach. When other threads (processors) have no jobs to complete, contained in their local queues or the global queue, they may run the tasks contained at the end of an appointed queue.

A number of unique instructions, including parallel loops like `Parallel.For`. are introduced by Parallel Extensions. The portability of Parallel Extensions is another benefit; for example, code generated on a 4-core system can utilize all of the 8, 16, or 24 cores on the machine on which it will be executed. This is not advantageous in programs where the number of threads is expressly stated. To illustrate, imagine a computational program designed with four threads hardcoded into the program. This program would function on computers with 1, 4, and 8 processor cores, but on the last one, it would only partially harness the capabilities of some of the available cores.

B. OpenMP

Another framework, OpenMP (Open Multi-Processing [7]), provides an alternative for languages like C++ and Fortran, the latter not integrated into the.NET framework. OpenMP comprises a collection of libraries, compiler directives, and environment settings that impact how programs run and ensure the adaptability of applications. This versatile system can be employed on both personal computers and supercomputers. The compiler includes parallel constructs, which can be invoked through compiler directives, making it easier to implement parallelism in the code. OpenMP does not offer language extensions. More control over parallelism is given to the programmer than with Parallel Extensions.

Threading, work separation, data environment management, thread synchronization, and runtime measurement are the fundamental building blocks of OpenMP. Despite the fact that OpenMP uses shared memory, users can define private variables. Different data sharing attributes, such as private, shared, and default, are utilized for this purpose. Additional synchronization clauses introduced by OpenMP include crucial sections, atomic blocks, and the capacity to preserve the order of loop iterations. The scheduling of concurrent jobs is another option. There are several scheduling options: guided (dynamically, batches of iterations run simultaneously), static scheduling (each iteration has a thread assigned before the start), dynamic scheduling (assigning iterations to the thread, follows the progress of previous iterations), and auto (the system decides the scheduling).

The `#pragma` compiler directive is used by OpenMP to identify the software fragments that should be parallelized. To spread loop iterations among the available processing units, the `Parallel.For` command should be used.

Simplicity, transparency for a sequential compiler, and adaptability to both fine- and coarse-grained parallelism are

characteristics of OpenMP solutions. It is unable to reliably manage exceptions or assign running threads to certain processors because of its complexity.

IV. Converting of a Sequential Program into a Parallel One

The objective of this project is to try and create algorithms that can autonomously identify sections of code within a sequential program and transform those sections into parallel threads. On a system with many processors (cores), such a customized program can run concurrently.

In the realm of sequential code structures, there are three main categories that can be made parallel: function calls, instruction paths, and loops. A list of prerequisites for adding parallelism was established. This leads to the creation of algorithms that assess whether a particular occurrence of each of these structures satisfies the criteria for parallelism. These algorithms are distinctive and come with explanatory components that can be utilized in both academic and professional contexts.

The examination relies on two approaches: a static evaluation of C# code and a dynamic comparison of the performance between sequential and parallel versions.

A. Asynchronous function call

In this section, we outline the invocations of a function, along with the conditions that must be fulfilled for such a structure to be executed concurrently alongside the code that follows the function call.

1) Function calls.

There are two sequences: during the function call and following the call, if a function is invoked. If any of the Bernstein conditions [3] are broken, they may become dependent on one another. Hence, by making use of the function result (if it provides the result), variables passed by reference, and variables affected by the function side effects (including those arising from nested function calls), we can identify the longest execution path after the function call that remains unaffected by all three of these characteristics. We call it *deferred use* when there is at least one statement with the aforementioned characteristics between the function call and the instruction. We have the option to execute the function concurrently with the code starting from the function call until the point where its first deferred usage occurs.

2) Data structures.

Each program statement must have a label that identifies explicitly it in order for the asynchronous call algorithm and subsequent algorithms to function. The algorithms employ the following data structures:

- *Call graph* is a directed graph, where nodes and edges represent functions within the source code under analysis. When there is an edge connecting nodes *X* and *Y*, it signifies that function *X* calls function *Y*. In practice, it is more like a multigraph because every function call results in an edge, and multiple edges can exist between the same pair of nodes. The edges are marked with the function call number for identification and algorithmic reasons.
- Collection of instructions *Causing Side Effects* (CSE). These instructions alter the value of a variable beyond their immediate scope or produce the program output.
- Collection of instructions *Dependent on Side Effects* (DSE). For example, these instructions may read a variable from outside their immediate scope, and that variable could potentially influence the result.

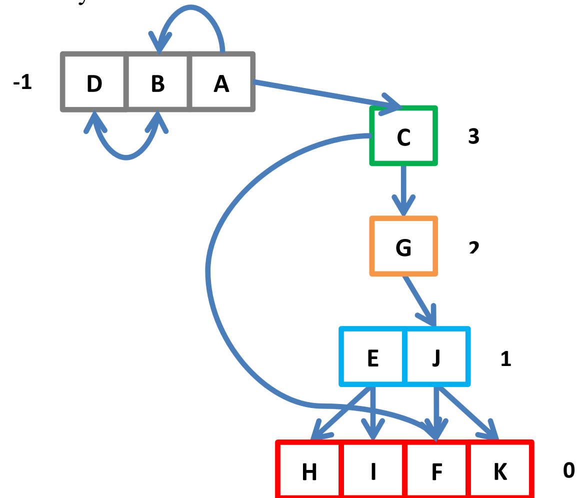

Fig. 1. Example of a call graph, with the analysis levels given in distinct colors. Every CSE/DSE entry contains a list of non-local variables accessed or resources that have been written.

3) Algorithm for asynchronous calls.

The proposed method changes the source code after determining whether asynchronous calls are feasible.

Call graph. The call graph is initially built using static analysis. Based on the documentation for the programming environment, only thread-safe library functions can be taken into consideration. The graph is created as follows: The initial set at level 0 in the analysis consists of the terminal functions within a call graph. Subsequent levels are defined as follows: The level *i* set includes all functions that directly invoke functions at level *i*-1 (and possibly levels below *i*-1) while excluding those within a call graph loop, except for recursive calls.

In the call graph shown in Fig. 1, we can discern different levels of function calls. In the topmost level, level 0 (depicted in red), we find functions *H*, *I*, *F*, and *K*. However, Function *B* does not fall into any level, denoted as level -1, because it is part of a loop (the *D-B* loop). Moving down to level 1 (represented in blue), we encounter functions *E* and *J*. Function *G* is situated in level 2, marked in orange, while *C* resides in level 3, indicated by green. Functions *A*, *B*, and *D*, grey, do not belong to any specific level. This is because they are enclosed within loops or call functions with undetermined levels.

When one function invokes another function (when the level is greater than 0), the CSE and/or DSE of the calling instructions rely on the CSE and/or DSE of the called function. If a function calls a library function, any documentation-based instructions related to the call are included in CSE. The level of a calling function that invokes a library function remains uncertain unless the library function is thread-safe (as described in [21] for .NET). Even when there are no observable side effects, using variables that are passed by reference can lead to their inclusion in the CSE or DSE. All such CSE and DSE are essentially "pulled" into the instructions that call the function. This is done because these symbols might undergo further analysis. However, there is an exception: local variables of the calling

function that are passed by reference are not included in this process.

4) Dynamic evaluation.

We can determine if enough parallelism is available until the deferred usage by looking at how long both parallel sequences take to run. We use three timestamps to evaluate this: Z_1, Z_2, Z_3. Z_1 marks the beginning of the function that will be asynchronously called, Z_2 is set when this function finishes, and Z_3 is placed at the point where the deferred usage occurs. We calculate the average differences between these timestamps across multiple program runs to get an accurate measure: $Z = Z_1 - Z_2$, $S = Z_3 - Z_2$. We also need to determine the average thread-starting time, P, for asynchronous calls. Using asynchronous calls makes sense when the times it takes for two measures, Z and S, are similar, and both Z and S are substantially larger than a third value, P: $Z \approx S, Z \gg P, S \gg P$, the asynchronous call is reasonable.

5) Introducing asynchronous call

The transparent mode of the asynchronous function call, which means that it is identical to sequential execution, must be preserved in the program. We should store the result of a function in a local variable within the calling function (if the called function produces a result) because threads usually do not directly return results.

B. Parallel statement paths

To execute the instructions within a function concurrently, we can organize them into instruction paths. Imagine the current program state as Z and the next statement to be executed as I. Then, the program state after executing statement I, referred to as $Z\{I\}$, can be described as a function $I(Z)$, where $Z\{I\} = I(Z)$. If we have two consecutive statements to execute, say I_1, I_2, the resulting system state after executing both, denoted as $Z\{I_1; I_2\}$, can be expressed as: $Z\{I_1; I_2\} = I_1(I_2(Z))$.

For a program having state Y, if we can apply two statements: I_0 and I_1, then we can say that I_1 directly depends on I_0 when $Y\{I_0; I_1\}$. differs from $Y\{I_1; I_0\}$. Indirect dependence of I_k on I_0 is in the case of an instruction sequence I_0, I_1, ..., I_{k-1}, I_k, where every I_j directly depends on the previous I_{j-1}, j=1,…,k. Instructions that are independent of each other can be executed simultaneously in parallel because the order of their execution does not affect the final result.

A series of dependent instructions that are organized according to the source program make up an instruction path. If two paths do not contain crosswise dependent assertions, then they are independent of one another. The maximum level of parallelism achievable for a specific sequence is determined by how many distinct independent paths can be identified within it.

The example function in which parallel paths are present is shown below. The two paths are found, and one of them is highlighted.

```
1.  void fun()
2.  {
3.   int k = 5;
4.   int l = 10;
5.   int p = fun_0();
6.   k = fun_1() ? p + 1 : p - 1;
7.   fun_2(k);
8.   l += 15;
9.   for(int i = 0; i < MAX; ++i)
10.  {
11.    l += 1;
12.  }
13.  if (k == 22)
14.  {
15.    p = k * 8;
16.  }
17.  return;
18. }
```

The sets for individual instructions are: CSE(3)={k}, CSE(4)={l}, CSE{5}={p} (we assume that CSE(*fun_0*) is empty, and DSE(*fun_0*) contains only variables not used in *fun*), CSE(6)={k}, DSE(6)={p} (similar assumptions to *fun_1*), DSE(7)={k} (we assume that CSE(*fun_2*) contains only variables not used in *fun*, and DSE(*fun_2*) contains only k), CSE(8)={l}, DSE(9)={i,l}, CSE(9)={i,l} (l pulled in from 11), DSE(13)={k,p}, CSE(13)={p} (p pulled in from 15), DSE(15)={k}, CSE(15)={p}.

The extraction of parallel paths is performed as follows: instruction 3 creates the set I (instruction 3, modified variable k), 4 creates the set II (instruction 4, modified variable l), 5 creates the set III (instruction 5, modified variable p). Instruction 6 merges the sets I and III, as it is a read/write conflict on p and write/write conflict on k, so the resulting set I contains instructions 3,5,6 and variables k,p. Instruction 7 enlarges set I by itself (accessed k), and 8 enlarges set II by itself (accessed and modified l). Instruction 9 adds itself and variable i to set II, and so on. Finally, set I contains instructions 3,5,6,7,13,15 and variables k,p, while set II contains instructions 4,8,9,11 and variables l,i. Therefore, two independent paths are identified that can be subject to parallelization.

If we want to make a function capable of running in parallel with multiple instruction paths, it can only contain one return statement. This return statement must always ensure the function's complete execution. Otherwise, it might lead to the execution of instructions in parallel paths that would not occur in a sequential mode.

1) Algorithm for independent paths

It is necessary to build an independence graph, with the labels of the instructions and the employed variables (local and non-local) serving as its nodes. The edges link the variable names used to the instruction labels. Furthermore, the content within a conditional statement is tied to the variables used in the condition itself, and the entire content within a loop is connected to the variables specified in the loop header. Code slicing techniques are the foundation of the independent path discovery algorithm [22]. Function calls must have their CDE and CSE calculated in accordance with the asynchronous calls algorithm, if there are any. They serve as the foundation for joining local and non-local variables given via reference and function call parameters. Edges connected with variables that are only read are eliminated from the graph due to the Bernstein conditions.

The possibility of parallelization exists within the independence graph when it consists of multiple separate components, with no edges connecting nodes between them. Each of these components represents the start of a unique path, and it is crucial to ensure synchronization at the end of each path within the function.

If the independence graph does not have separate components, the potential for parallelization might still exist within a cohesive section of the function. This could be something like an `if-else` instruction, an `if` instruction alone, or the content of a loop for a single iteration.

2) Dynamic evaluation

To make instructions run in parallel, we create individual threads for each independent path, execute them in parallel, and then synchronize their control flow. Parallelization becomes practical when we have at least two paths with similar execution times, which are much larger than the average time of starting parallel threads, denoted as *P*. It is a good idea to consolidate all paths with execution times close to or shorter than *P*, into longer paths. The same if their execution times are significantly shorter than the two (or more) longest ones. This helps optimize the parallel execution process.

As the instructions within these paths can overlap and run concurrently, it is challenging to estimate the path execution times of the independent paths in a sequential program. As a result, we must apply the solution to as many paths as possible, assess their execution times, and then join any that do not comply with the timing criteria.

3) Running the paths in parallel

Parallel paths should originate from a common starting point. Subsequently, for each of these parallel paths, a thread comprising instructions from of the component in the graph in the original order should be initiated.

C. Parallelized loops

The loop is often considered the most conducive component for parallelism. If the following conditions hold, it is possible to parallelize loop iterations because of their repetitive nature and behavior (where the same piece of code is executed multiple times).

1) Loops

There are two primary types of loops in high-level languages: `for` (`foreach` can be transformed to `for`), and `while` (also including `do-while`). The examination of loops considers variables access (side effects, parameters, and iteration dependency), loop nesting, and the capacity to forecast the number of iterations.

Number of iterations. The loop cannot be parallelized if it includes situations where the loop, its containing function, or the entire program terminates prematurely. Loop iterations that cannot be interrupted can be identified statically or dynamically.

The most commonly used loops are `for` loops, where the number of iterations is either set as a fixed value or determined dynamically before the loop begins. `For` loops, whose size can be predetermined or fixed before the loop begins, are frequently used to iterate through elements of collections. The initializer, condition, and iterator are all contained in the header of `for` loop. In a `for` loop, the initializer runs before the loop starts, the iterator runs after each cycle, and the condition is checked before every iteration. If the condition is false, the loop is terminated, and any subsequent cycles are skipped.

Loops `while` are commonly used when the number of iterations depends on an ongoing condition, and the exact number of iterations is uncertain. In such loops, it is typical for the values of variables used in the condition to change during the loop execution or for the loop to be terminated prematurely, as seen in loops like `while(true)`. Due to this unpredictability, we primarily focus on analyzing `for` loops.

Analysis of loop dependencies. Other loop iterations may be able to make use of side effects produced by each loop iteration. We consider any changes to function-local variables, like the loop counter, as side effects. It is important to note that modifying variables that are part of the loop condition is not allowed, as these variables are interdependent when one loop cycle triggers side effects that affect other iterations.

Consider a loop that runs *N* times, with iterations numbered from 1 onwards, regardless of the actual values of the loop iterator. We define the following sets of variables for each iteration:

- PI_n: The variables read in the loop header iterator after iteration n,
- PO_n: The variables modified in the loop header iterator after iteration n,
- CI_n: The variables that are read in the loop body in iteration n and also in the condition of the loop,
- CO_n: The set of variables modified in the loop body in iteration n.

If a variable read within the loop body is modified only in the loop iterator after each iteration, and its modification is not dependent on the variables modified in the loop body of any iteration (in other words, $PO_n \cap CO_n = \emptyset \rightarrow PO_n \cap CI_n = P_n$), and the variables CO_n do not influence any PO_m ($PO_n \neq F(CO_m)$ for any n and m), then the value of this variable for each iteration n can be computed before the loop cycle begins..

If the following criteria are true for $i, j \in (1, \ldots, N)$: $CI_j \cap CO_i = \emptyset$, $CI_i \cap CO_j = \emptyset$, $CO_i \cap CO_j = \emptyset$, which are the modified Bernstein conditions, and if any variables changed by the loop iterator are loop parameters, then this `for` loop can be parallelized. In this context, `for` loops that use a counter iterated after each iteration can be parallelized.

Nested loops. When working with multi-dimensional data, like the rows and columns of an image or matrix multiplication, we might encounter nested loops. These are loops within loops, and it's not very common to have four or more levels of nesting.

We attempt to parallelize the outermost loop in nested loops because it offers the most parallelism. When a parallelized loop Z_1 is nested inside loop Z_0, note that during each iteration of Z_0, many parallel Z_1 cycles begin in parallel. In this scenario, Z_0 has to wait for all the Z_1 iterations to finish before it can begin the next iteration. However, if we parallelize Z_0, the Z_1 cycles from different Z_0 iterations can run simultaneously. This means that parallelization and waiting for parallel threads to finish only occur once, making the process more efficient.

2) Loop parallelism

Parallelizing a loop is possible when these three conditions are satisfied:

- We must know the values of loop parameters for each iteration before starting the loop.
- Each iteration should be independent and not rely on others.
- The total number of iterations should be known in advance.

It is important to note that these criteria significantly limit the number of loops that can be parallelized.

3) Parallelized loop algorithm

Loop parameters analysis. A loop iterator is a variable read within the loop body but only modified in the loop header. Significantly, any changes made to this iterator in the header

must not be influenced by modifications occurring inside the loop.

Since we cannot predict the iterator value before the loop starts, parallelization is not possible if the variables modified in the loop iterator intersect with the variables modified inside the loop.

The next step is to check if altering the loop parameters is unrelated to changing any variables inside the loop. It is crucial for these two sets of variables, loop parameters and those updated inside the loop, to be distinct. Otherwise, parallelization of the loop is not feasible.

Iteration independence analysis. Determining the sets of variables that are read and written involves examining the DSE and CSE of called functions. If a variable is both read and written within the loop, it creates the potential for a race condition between loop cycles. This situation aligns with what is known as the Bernstein condition, where a race can occur between two iterations. Here is an example of code where this race condition can happen.

```
1.  void count(int* data, int* result,
                int number)
2.  {
3.   const int constant = 5;
4.   int lastResult = 0;
5.   for(int i=0; i<number; ++i)
6.   {
7.     int temp = Math.Pow(data[i], 3);
8.     temp += constant;
9.     lastResult = temp + data[i];
10.    result[i] = lastResult;
11.  }
12. }
```

The variable `lastResult` is read and changed in a single iteration.

Iteration count analysis. Suppose we have a loop with a condition and iterator modification in a specific form, along with some additional conditions explained below. In that case, we can determine the number of iterations in advance.

- The condition has the form: <w_1><op><w_2>, where w_1 and w_2 are either simple variable names or constant numbers, and op is an arithmetic operator.
- One of (w_1, w_2) must be the *loop counter*, whose value changes by the same amount in each iteration.
- The other variable, either w_1 or w_2, should be a simple variable or a constant called *limit*.
- We need to know the initial value of the loop counter before the loop starts.
- The loop counter must be modified using one of the following operators: `++`, `--`, `+=`, `-=` . The value added or subtracted must always be 1 in the first two cases. In the latter two cases, if it is a variable, its value must remain constant during the loop execution.
- The operator `op` in the condition must be one of the following: `==`, `!=`, `<`, `>`, `<=`, `>=`.

We can determine the number of loop iterations based on the operator used in the loop condition, the counter modification operator, and the modifier value. To simplify this, let us consider that w_1 represents the loop counter, w_{1_0} is the initial value of the counter, and w_2 is the limit. The exact analysis of this calculation can be quite extensive and detailed, so we will not delve into it here because it consumes too much space. However, it is possible to calculate the number of iterations by considering these variables and their interactions, for instance; when $w_1 = w_2, p = 0$, `op`: `+=` infinitely many iterations; when $w_1 \neq w_2, w_2 < w_{1_0}$, `op`: `++` unknown number of iterations; and when $w_1 \neq w_2, w_2 > w_{1_0}, p > 0$, `op=++` $\rightarrow \frac{w_2 - w_{1_0}}{p}$ iterations. Only the latter scenario is viable. Out of the cases we examined, we identified 18 scenarios where the loop resulted in more than one iteration. In these cases, we were able to determine the number of iterations, as in the code (=`number`):

```
1.  void countShortcuts(FILE* files,
                        MD5* shortcuts, int number)
2.  {
3.     for(int i=0; i<number; ++i)
4.     { shortcuts[i] = CountMD5(files[i]); }
5.  }
6.
7.  MD5 CountMD5(FILE* file)
8.  { //long calculations }
```

4) Dynamic evaluation

A number of program runs using various data sets are required for evaluation, with the time taken for each loop cycle being recorded. A log must be output because there may be thousands of cycles. It is reasonable to use parallelization if, for the average situation, at least two iterations of a loop take a similar amount of time, and this time is significantly longer than the time it takes to create and synchronize threads, then parallelization makes sense..

5) Introducing loop parallelism

Every iteration must begin with a separate variable that represents the counter in order to run the algorithm. By adding the modifier to w_{1_0}, and applying *n*-1 modifications, the assigned value comes. The condition is not required because parallelism consumes it.

V. Common Considerations

Even though we can use asynchronous calls as they are, sometimes we might end up creating many parallel threads simultaneously in parallel pathways or loops. This number could exceed the actual number of processors or cores, or the parallel threads that the operating system has assigned to the program. To manage this efficiently, creating threads in smaller groups closer to the system limits is a good idea.

In parallelization, synchronization is a significant problem. Synchronization points are provided by environments like Parallel Extensions, where the thread ends are gathered before the beginning of the subsequent instruction. To ensure proper synchronization in asynchronous calls, we must establish synchronization points before a deferred usage occurs. These synchronization points are essential, especially when a function or a portion of it ends in parallel paths or if a loop is parallelized. In cases where a built-in synchronization mechanism is unavailable, we can utilize existing synchronization mechanisms. The instruction following the required synchronization point typically involves waiting on a lowered semaphore to coordinate the execution flow. Before the threads branch, a thread counter is set up, and each completed thread reduces the counter in a critical section. When the counter hits 0, the semaphore is raised. The event mechanism is available in C#.

Our techniques have some limits, including the use of virtual functions, exceptions, and nested function calls like

$f1(f2())$.The virtual function approach was discarded because it is impossible to predict which method, either from the base class or the inheriting class, will be called. This unpredictability stems from the limitations of static code analysis, which cannot determine the specific type of object that might be assigned to a given variable.

A side effect is any alteration in an object or resource state that does not stem from the call parameters or interactions with the environment. When multiple statements induce side effects on the same resource, like a computer screen, they cannot be executed simultaneously without additional synchronization efforts. In simpler terms, if two actions mess with the same thing, they must be carefully coordinated to avoid conflicts. The developer should decide which resource operations can be carried out concurrently.

A. Synchronization

In specific high-level programming languages like C#, we have an alternative to using threads. Instead of manually managing threads, we can use language features or libraries that provide parallel processing capabilities. In the case of C#, we can swap out a traditional `for` loop with a parallelized version, specifically the `Parallel.For` loop found in the Parallel Extensions library. This allows for harnessing the power of parallel processing without getting into the details of thread management.

For running the sets of instructions in parallel in our benchmark (see Section 6), we use C# threads with `thread.Start()` and `thread.Join()`. In Unix-like environments, processes can replace threads, using `fork()` and `wait()` operations, but we must remember that in this case the calculations are slower because the process data is duplicated in `fork()`. If the result of calculations is a single value, it can be passed as the exit code. The program is:

```
1.  int i = fork();
2.  if (i==0)
3.  {
4.     //do the child work
5.     exit(EXIT_SUCCESS);
6.  }
7.  else
8.  {
9.     int wstatus;
10.    do
11.    {
12.       int w = waitpid(i, &wstatus,
                  WUNTRACED | WCONTINUED);
13.       if (w == -1) exit(EXIT_FAILURE);
14.    } while (!WIFEXITED(wstatus));
15.    int result = WEXITSTATUS(wstatus);
16. }
```

If the results of the child program cannot be passed using the exit code, the data can be provided to the parent using a file, shared memory area, or a pipe.

In other environments, a call of a child program from the parent program can be applied, and the programs should read the same input data. For synchronization, a simple semaphore initialized to 0 can be used. For example, if we want to run 6 loop cycles, one of them can be performed by the main thread (say, cycle 0). All the other cycles 1..5 can be performed by the child programs. The results can be passed as above by files, shared memory area, or a pipe. The parent program:

```
1.  start semaphore s(0);
2.  for (int i=1; i<6; ++i)
3.  { startChildProgram(i); }
4.  loopContent(0);
5.  for (int i=1; i<6; ++i) s.P();
```

The child program *i*:

```
1.  loopcontent(i);
2.  s.V();
```

B. Nesting

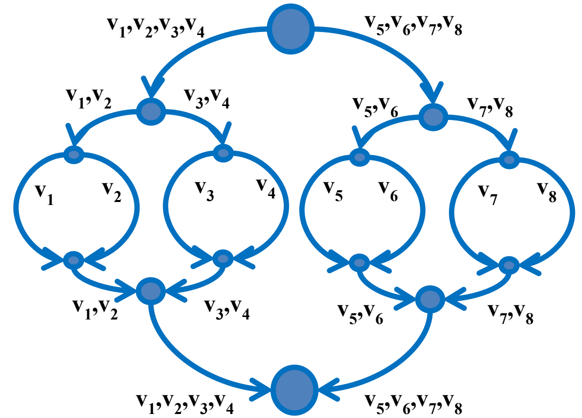

Fig. 2. Nesting independent paths

The proposed methods can be hierarchically mixed. This means that, for example, parallel paths can comprise an asynchronous function call or a parallelized loop. Additionally, independent paths can be split into more detailed independent paths that concern subsets of variables accessed in the “parent” path. This is described graphically in Fig. 2, in which a set of variables is divided into subsets in a hierarchical way.

The outer paths concern the sets of variables v_1-v_4 and v_5-v_8, respectively. The path concerning v_1-v_4 is split into shorter subpaths in which v_1,v_2 and v_3,v_4 are accessed independently, which do not occur in the starting and ending fragments of the parent path. More deeply, the subpath concerning v_1 and v_2 is split into shorter subpaths concerning individual variables v_1 and v_2 separately. Fig. 3 shows symbolically how the cases can be nested.

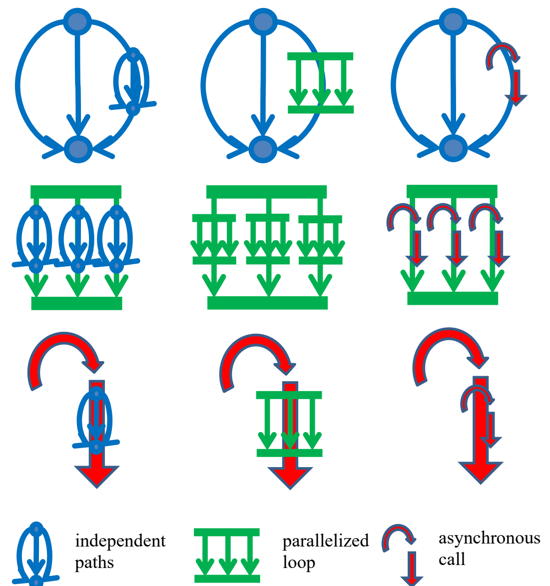

Fig. 3. Nesting the cases

VI. CASE STUDY

We have developed an experimental tool designed for loop parallelization, and we have put it to the test on various programs, including tasks like matrix multiplication, N-dimensional space point neighborhood search, image recognition, and information entropy computation.

What is particularly noteworthy is that most of the loops we encountered could not be transformed into structures suitable for parallelization. These loops often relied on external resources, like images or data collections processed by the application. We observed similar challenges when examining the other two sequential constructs discussed in this article, namely asynchronous calls and parallel paths.

In the cases of matrix multiplication and neighborhood search, we found that out of the 16 loops available, only 8 could be effectively parallelized. However, it is essential to note that in the case of matrix multiplication, introducing parallelism led to a notable efficiency boost of approximately 40% for the corresponding code. The multiplication program contains 3 hierarchical loops. The tool replaced the outer loop `for (int i = 0; i < matARows; i++) {…}` with the parallel loop `Parallel.For (0; matARows; i => {…})`.

In some instances, the loop iterations were very brief, and when we attempted to implement parallelization, it actually extended the overall execution time of the loop. Therefore, we decided to elaborate a benchmark common for all three techniques of parallelization, and show how to write the code that is tractable for introducing parallelism.

A. The benchmark

To illustrate the application, we prepared a benchmark of a sequential process in C# that can be parallelized in different ways. We chose the mergesort algorithm, as it is tractable to be cut into independent subprocedures. The basic C# code is:

```
1.  string[] mergeSort(string[] v)
2.  {
3.     if (v.Length <= 1) return v;
4.     int middle = v.Length / 2;
5.     string[] v1 =
           v.Skip(0).Take(middle).ToArray();
6.     v1 = mergeSort(v1);
7.     string[] v2 = v.Skip(middle)
8.        .Take(v.Length - middle).ToArray();
9.     v2 = mergeSort(v2);
10.    return merge(v1, v2);
11. }
```

The sets of individual instructions are: DSE(3)={*v*}, CSE(4)={*middle*}, DSE(4)={*v*}, CSE(5)={*v1*}, DSE(5)={*v*,*middle*}, CSE(6)={*v1*}, DSE(6)={*v1*}, CSE(7)={*v2*}, DSE(7)={*v*,*middle*}, CSE(9)={*v2*}, DSE(9)={*v2*}, DSE(10)={*v1*,*v2*}. The function `mergeSort` is recursive, but it does not produce any side effect other than modifying the variable that holds the result, so we can define CSE(`vi=mergeSort(vi)`)={*vi*}.

B. Asynchronous call

We can apply an asynchronous call to the first invocation of `mergeSort` in line 6. The variable *v1* receiving the result is used in line 10, so we have a deferred use effect. In C#, a thread must execute a function, so we define the first execution of the sort as the separate function `mergeSortParallel` and extract the first nested call of `mergeSort` to the new function `mergeSortThread`. The thread cannot pass a result; thus, the nonlocal variable *v1s* is used instead of the local variable *v1*. The function `mergeSort` remains unchanged. The parallelized function containing asynchronous call is given below.

```
1.  string[] v1s;
2.  void mergeSortThread()
3.  { v1s = mergeSort(v1s); }
4.
5.  string[] mergeSortParallel(string[] v)
6.  {
7.     if (v.Length <= 1) return v;
8.     int middle = v.Length / 2;
9.     v1s = v.Skip(0).Take(middle).ToArray();
10.    Thread thread = new Thread(new
                   ThreadStart(mergeSortThread));
11.    thread.Start();
12.    string[] v2 = v.Skip(middle)
13.       .Take(v.Length - middle).ToArray();
14.    v2 = mergeSort(v2);
15.    thread.Join();
16.    return merge(v1s, v2);
17. }
```

C. Parallel paths

To apply parallelization of independent paths, we need the sets of instructions not in conflict (read/write or write/write). Of course, it could be continued to sort two parts in parallel, but here, we decided to create four parallel paths performing partial sorts of the array quarters. For parallelization, we must extract the calculation of quarters size to the calling function, as this variable is modified in the sorting function and used in every partial sort. This causes all partial sorts to be in read/write conflict with the quarters size calculation. A similar situation applies to merging the quarters, so the merging is extracted to the calling function. Like previously, the independent paths are extracted as the separate function `mergeSortMemberGang`. Generally, every path should have its own function, but we use a common function because sorting the quarters is performed identically. The parameter is passed as an object because such is the requirement for a parametrized thread start. The function prepared for parallelization is:

```
1.  string[][] vg = new string[4][];
2.  void mergeSortMemberGang(object i)
3.  { vg[(int)i] = mergeSort(vg[(int)i]); }
4.
5.  void mergeSortGang(string[] v, int part)
6.  {
7.     vg[0] = v.Skip(0).Take(part).ToArray();
8.     mergeSortMemberGang((object)0);
9.     vg[1] = v.Skip(part).Take(part).ToArray();
10.    mergeSortMemberGang((object)1);
11.    vg[2] =
             v.Skip(part*2).Take(part).ToArray();
12.    mergeSortMemberGang((object)2);
13.    vg[3] = v.Skip(part*3).Take(v.Length-
                            3*part).ToArray();
14.    mergeSortMemberGang((object)3);
15. }
16.
17. string[] mergeSortControl(string[] v)
18. {
19.    if (v.Length <= 1) return v;
20.    if (v.Length <= 3) return mergeSort(v);
21.    int part = v.Length / 4;
22.    mergeSortGang(v, part);
23.    vg[0] = merge(vg[0], vg[1]);
24.    vg[2] = merge(vg[2], vg[3]);
25.    return merge(vg[0], vg[2]);
26. }
```

The parallelized code of four independent paths is:

```
1.  mergeSortGang(string[] v, int part)
2.  {
3.      Thread[] threads = new Thread[4];
4.      object[] obj = new object[4] { 0,1,2,3 };
5.      vg[0] = v.Skip(0).Take(part).ToArray();
6.      threads[0] = new Thread(
7.         new ParameterizedThreadStart
                          (mergeSortMemberGang));
8.      threads[0].Start(obj[0]);
9.      vg[1] =
              v.Skip(part).Take(part).ToArray();
10.     threads[1] = new Thread(
11.        new ParameterizedThreadStart
                          (mergeSortMemberGang));
12.     threads[1].Start(obj[1]);
13.     vg[2] =
            v.Skip(part*2).Take(part).ToArray();
14.     threads[2] = new Thread(
15.        new ParameterizedThreadStart
                          (mergeSortMemberGang));
16.     threads[2].Start(obj[2]);
17.     vg[3] = v.Skip(part*3).
                 Take(v.Length-3*part).ToArray();
18.     threads[3] = new Thread(
19.        new ParameterizedThreadStart
                          (mergeSortMemberGang));
20.     threads[3].Start(obj[3]);
21.     for (int i = 0; i < threads.Length; ++i)
22.        threads[i].Join();
23. }
```

D. *Parallel loop*

The solution prepared for loop parallelization is quite similar to that with independent paths; the difference is in calling the quarter sort inside the loop rather than in the instruction sequence. As we work with the fragment of the function, the calculation of the quarter size and merging can be restored:

```
1.  string[][] vf = new string[4][];
2.  void mergeSortMemberFor(object i)
3.  { vf[(int)i] = mergeSort(vf[(int)i]); }
4.
5.  string[] mergeSortFor(string[] v)
6.  {
7.     if (v.Length <= 1) return v;
8.     if (v.Length <= 3) return mergeSort(v);
9.     int part = v.Length / 4;
10.    for (int i = 0; i < 4; ++i)
11.    {
12.        if (i < 3) vf[i] = v.Skip(i*part).
                         Take(part).ToArray();
13.        else vf[3] = v.Skip(3*part).
                 Take(v.Length-3*part).ToArray();
14.        mergeSortMemberFor((object)i);
15.    }
16.    vf[0] = merge(vf[0], vf[1]);
17.    vf[2] = merge(vf[2], vf[3]);
18.    return merge(vf[0], vf[2]);
19. }
```

The loop can be parallelized using `Parallel.For`, but here we show how it can be converted to explicit threads.

```
1.  string[] mergeSortFor(string[] v)
2.  {
3.     if (v.Length <= 1) return v;
4.     if (v.Length <= 3) return mergeSort(v);
5.     int part = v.Length / 4;
6.     Thread[] threads = new Thread[4];
7.     for (int i = 0; i < 4; ++i)
8.     {
9.         if (i < 3) vf[i] = v.Skip(i*part).
                          Take(part).ToArray();
10.        else vf[3] = v.Skip(3*part).
                  Take(v.Length-3*part).ToArray();
11.        threads[i] = new Thread(
12.           new ParameterizedThreadStart
                          (mergeSortMemberFor));
13.        threads[i].Start((object)i);
14.    }
15.    vf[0] = merge(vf[0], vf[1]);
16.    vf[2] = merge(vf[2], vf[3]);
17.    return merge(vf[0], vf[2]);
18. }
```

E. *Tests results*

We tested the four parallelized solutions compared with the sequential ones on the text files downloaded from the Gutenberg database at https://zenodo.org/record/3360392 : 6MB, 72MB, and the later one repeated three times (215MB). For all those files, the run time reduction was independent on the file size. The asynchronous call solution took approx. 60% of sequential program, and for both independent paths and parallelized loop, it was around 45%. The tests were run on 4-core processor. This environment somehow disturbs the tests, because we have totally five threads (including the main one) and the operating system running in parallel, but it shows the potential of parallelization.

On the base of the presented cases, we can formulate the hints for building the functions that are tractable for parallelization. For asynchronous call, the deferred use should be as far from the function call as possible. For independent paths, initial write operations on common data (that are only read along the rest of the function) and aggregation of results should be extracted to the calling function, leaving sets of instructions more independent. For parallel loop, the loop cycles should be independent between themselves, and independent of loop control variables. Additionally, among loop control instructions, only `continue` can be applied to preserve the total number of cycles. If there are exceptions that can be raised, they should be extracted to the calling function.

VII. Summary and Future Work

This article primarily focuses on the practical methodology for transforming sequential programs written in C-like languages into parallel programs. It delves into a comprehensive analysis of three fundamental components in sequential code: function calls, instruction paths, and loops. Through this in-depth exploration, the article introduces algorithms designed to identify and parallelize these code constructs, along with the necessary conditions for introducing parallelism within them. The analysis can go deeper and discover nested cases, as shown in Fig. 3. Especially, identifying nested independent subpaths requires additional steps in the algorithm (Fig. 2).

Our approach enables parallelism without the need for extra resource synchronization methods. Instead, it achieves this by thoroughly analyzing relevant program structures and assessing their independence. Future research might involve investigating dependencies and using these findings to automatically introduce parallelism along with resource synchronization. Such an approach could potentially expand the range of structures that can be parallelized.

To sum it up, automating the integration of parallelism is a highly complex subject that demands a comprehensive analysis of numerous aspects and components. Future solutions might avoid this requirement by using completely different machine learning techniques. New data-driven approaches, using ML customized deep learning models [23] or large language models

trained on cross-lingual keywords [24], are emerging. We have conducted initial experiments using a general-purpose language model in ML to convert sequential code into parallel code. However, our early experiments show limitations in direct model use, and further model training is needed. Additionally, the explanation of the transformations made by these models, would assist in identifying limitations of ML approach and thus lead to further improvements [25].

ACKNOWLEDGEMENTS

This article is an extended version of the paper presented on Depcos-Relcomex 2023 conference [26].

REFERENCES

[1] C. Campbell, R. Johnson, A. Miller, and S. Toub, *Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures*, 1st ed. Redmond, WA: Microsoft Press, 2010. ISBN: 978-0-7356-5159-3

[2] ECMA_International, “C# language speification,” 2002. [Online]. Available: https://www.ecma-international.org/publications-and-standards/standards/ecma-334/.

[3] A. J. Bernstein, “Analysis of Programs for Parallel Processing,” *IEEE Trans. Electron. Comput.*, vol. EC-15, no. 5, pp. 757–763, Oct. 1966. doi:10.1109/PGEC.1966.264565

[4] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *spring joint computer conference on - AFIPS ’67 Atlantic City, NJ, April 18-20, 1967*, 1967, p. 483. doi:10.1145/1465482.1465560

[5] J. L. Gustafson, “Reevaluating Amdahl’s law,” *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988. doi:10.1145/42411.42415

[6] M. Danelutto, J. D. Garcia, L. M. Sanchez, R. Sotomayor, and M. Torquati, “Introducing Parallelism by Using REPARA C++11 Attributes,” in *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), Heraklion, Greece, 17-19 Feb 2016*, 2016, pp. 354–358. doi:10.1109/PDP.2016.115

[7] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP - Portable Shared Memory Parallel Programming*. Cambridge, MA: MIT Press, 2007. ISBN: 978-0-262-53302-7

[8] R. Atre, A. Jannesari, and F. Wolf, “Brief Announcement: Meeting the Challenges of Parallelizing Sequential Programs,” in *29th ACM Symposium on Parallelism in Algorithms and Architectures, Washington, DC, 24 - 26 July 2017*, 2017, pp. 363–365. doi:10.1145/3087556.3087592

[9] D. Dig, “A Refactoring Approach to Parallelism,” *IEEE Softw.*, vol. 28, no. 1, pp. 17–22, Jan. 2011. doi:10.1109/MS.2011.1

[10] Z. Li, “Discovery of Potential Parallelism in Sequential Programs,” PhD thesis, Technische Universität Darmstadt, Department of Computer Science, 2016. [Online] Available: https://tuprints.ulb.tu-darmstadt.de/5741/7/thesis.pdf

[11] F. Allen, M. Burke, R. Cytron, J. Ferrante, and W. Hsieh, “A framework for determining useful parallelism,” in *2nd international conference on Supercomputing - ICS ’88, St. Malo, France, 1 June 1988*, 1988, pp. 207–215. doi:10.1145/55364.55385

[12] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, “Cpp-Taskflow: Fast Task-Based Parallel Programming Using Modern C++,” in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Rio de Janeiro, Brazil, 20-24 May 2019*, 2019, pp. 974–983. doi:10.1109/IPDPS.2019.00105

[13] Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke, “Uncovering hidden loop level parallelism in sequential applications,” in *2008 IEEE 14th International Symposium on High Performance Computer Architecture, Salt Lake City, UT, 16-20 Feb 2008*, 2008, pp. 290–301. doi:10.1109/HPCA.2008.4658647

[14] A. W. Lim and M. S. Lam, “Maximizing parallelism and minimizing synchronization with affine partitions,” *Parallel Comput.*, vol. 24, no. 3–4, pp. 445–475, May 1998. doi:10.1016/S0167-8191(98)00021-0

[15] Z. Li, R. Atre, Z. Huda, A. Jannesari, and F. Wolf, “Unveiling parallelization opportunities in sequential programs,” *J. Syst. Softw.*, vol. 117, pp. 282–295, Jul. 2016. doi:10.1016/j.jss.2016.03.045

[16] G. Tagliavini, D. Cesarini, and A. Marongiu, “Unleashing Fine-Grained Parallelism on Embedded Many-Core Accelerators with Lightweight OpenMP Tasking,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 9, pp. 2150–2163, Sep. 2018. doi:10.1109/TPDS.2018.2814602

[17] S. Rul, H. Vandierendonck, and K. De Bosschere, “A profile-based tool for finding pipeline parallelism in sequential programs,” *Parallel Comput.*, vol. 36, no. 9, pp. 531–551, Sep. 2010. doi:10.1016/j.parco.2010.05.006

[18] A. Fonseca, B. Cabral, J. Rafael, and I. Correia, “Automatic Parallelization: Executing Sequential Programs on a Task-Based Parallel Runtime,” *Int. J. Parallel Program.*, vol. 44, no. 6, pp. 1337–1358, Dec. 2016. doi:10.1007/s10766-016-0426-5

[19] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai, “A cost-driven compilation framework for speculative parallelization of sequential programs,” *ACM SIGPLAN Not.*, vol. 39, no. 6, pp. 71–81, Jun. 2004. doi:10.1145/996893.996852

[20] Chen Tian, Min Feng, V. Nagarajan, and R. Gupta, “Copy or Discard execution model for speculative parallelization on multicores,” in *41st IEEE/ACM International Symposium on Microarchitecture, Como, Italy, 08-12 Nov. 2008*, 2008, pp. 330–341. doi:10.1109/MICRO.2008.4771802

[21] Microsoft, “Thread-Safe Collections.” [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/collections/thread-safe/.

[22] M. Harman and R. Hierons, “An overview of program slicing,” *Softw. Focus*, vol. 2, no. 3, pp. 85–92, 2001. doi:10.1002/swf.41

[23] Y. Shen, M. Peng, S. Wang, and Q. Wu, “Towards parallelism detection of sequential programs with graph neural network,” *Futur. Gener. Comput. Syst.*, vol. 125, pp. 515–525, Dec. 2021. doi:10.1016/j.future.2021.07.001

[24] OpenAI, “ChatGPT: Optimizing Language Models for Dialogue.” [Online]. Available: https://openai.com/blog/chatgpt/.

[25] A. Barredo Arrieta *et al.*, “Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI,” *Inf. Fusion*, vol. 58, pp. 82–115, Jun. 2020. doi:10.1016/j.inffus.2019.12.012

[26] D. B. Czejdo, W. B. Daszczuk, and W. Grześkowiak, “Practical Approach to Introducing Parallelism in Sequential Programs,” in *18th International Conference on Dependability of Computer Systems DepCoS-RELCOMEX, Brunów, Poland, 3-7 July 2023*, 2023, pp. 13–27. doi:10.1007/978-3-031-37720-4_2