Tool-based Interactive Software Parallelization: A Case Study

Andreas Wilhelm, Faris Čakarić, Michael Gerndt
Technical University of Munich
Munich, Germany
{wilhelma,cakaric,gerndt}@in.tum.de

Tobias Schuele
Siemens Corporate Technology
Munich, Germany
tobias.schuele@siemens.com

ABSTRACT

Continuous advances in multicore processor technology have placed immense pressure on the software industry. Developers are forced to parallelize their applications to make them scalable. However, applications are often very large and inherently complex; here, automatic parallelization methods are inappropriate. A dependable software redesign requires profound comprehension of the underlying software architecture and its dynamic behavior. To address this problem, we propose Parceive, a tool that supports identification of parallelization scenarios at various levels of abstraction. Parceive collects behavior information at runtime and combines it with reconstructed software architecture information to generate useful visualizations for parallelization. In this paper, we motivate our approach and explain the main components of Parceive. A case study demonstrates the usefulness of the tool.

1 INTRODUCTION

Multicore processors pose a major challenge to software development. Today, even the cheapest IoT devices are equipped with multiple processing units. As a result, software and system vendors are forced to parallelize legacy applications to make them scalable. Unfortunately, parallelization is tedious and error-prone. Few programmers can realize the full power of parallelism; most programmers only achieve a small fraction of the potential performance gains [21]. The difficulties associated with parallelization are related to inherent complexity of parallel programming and the lack of appropriate tools to support developers and architects.

Various approaches to aid developers in parallelizing applications have been proposed. For example, automatic parallelization has been considered the silver bullet [3, 5, 15]. Automatic parallelization approaches allow programmers use their existing sequential programming models to exploit the benefits of modern computer architectures. To preserve application semantics, the employed analyses must make conservative decisions; thus, automatic parallelization is only applicable to small code blocks and simple loops without intricate dependencies [20]. Semi-automatic parallelization leverages developers knowledge to enable additional parallelism [4, 10]. With semi-automatic parallelization, the input informs the tools which dependencies can be safely ignored. Unfortunately, identifying such dependencies is not trivial and sometimes not possible without redesigning the sequential application toward concurrency. A promising alternative is guided parallelization, which gives the developer full control over parallelization and provides means to identify dependencies.

Parallelization of industrial applications requires an in-depth understanding of the software and its problem domain. Typically, such applications consist of various components at different levels of abstraction that heavily interact with each other. Examples are client-server applications that process a large number of requests. These applications usually contain database managers, message queues, central logging facilities, and the actual business logic. To exploit parallelism and achieve good scalability and low latency, all components must be thread-safe. However, the necessary refactoring requires a solid understanding of the overall software architecture and its inherent dependencies.

In this paper we propose Parceive, a tool for guided, interactive software parallelization. Its goal is to improve developer comprehension of their applications that enables identification of possible parallelization scenarios. To achieve this, the proposed tool collects information about program behavior during runtime and combines it with software architecture information. The analysis results are then used to provide effective visualizations (Figure 1) and analyses that attempt to answer questions relative to parallelization. The resulting parallelization scenarios can represent a blueprint for a potential refactoring or redesign.

The remainder of this paper is organized as follows. In Section 2, we describe Parceive and its main components. In Section 3, we explain our approach using an open-source use case. We discuss the implications and limitations of our approach in Section 4 and provide conclusions and suggestions for future work in Section 5.

2 RELATED WORK

The proposed tool-based approach considers a wide range of research areas, such as parallelization, software architecture reconstruction, and software visualization. We are not aware of any study that has covered these research areas collectively; however, several studies share similar objectives.

Parallelization Tools. Purely static approaches are overly conservative to parallelize industrial applications. The resulting parallelism is primarily limited to the level of instructions or loops. Consequently, recent approaches have increasingly relied on dynamic information (e.g., data dependencies) gathered during program execution. Although being input-sensitive, dynamic analysis returns precise results that lead to solutions with greater parallelism.

One approach is to identify parallelization scenarios automatically and rank them according to their parallelism potential. For example, Kremlin [9] determines the critical path across code regions

ICSE-SEIP '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5659-6/18/05…$15.00
https://doi.org/10.1145/3183519.3183555

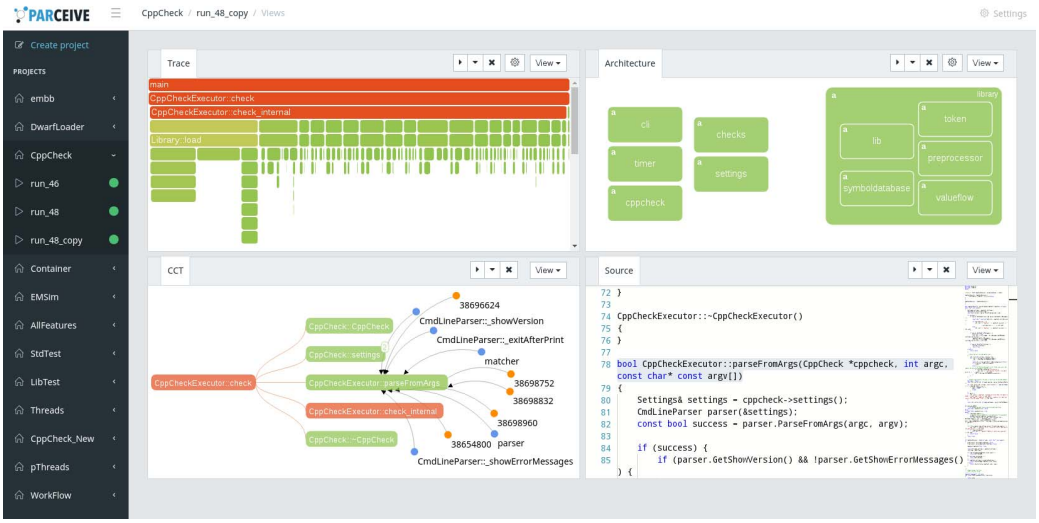

Figure 1: Parceive's graphical user interface: (top left) Performance View; (top right) Architecture View; (bottom left) CCT View; (bottom right) Source Code View

to rate the so-called self-parallelism of predefined language constructs. In contrast, DiscoPop [14] discovers parallelism on the level of so-called computational units that represent instructions following a read-compute-write pattern, which enables the identification of parallel tasks that are not well aligned with the source-code structure. Kremlin and DiscoPop rely on static instrumentation, which makes them inappropriate to detect data dependencies on components without available source-code (e.g., third-party libraries). Alchemist [27] utilizes dynamic instrumentation to unveil parallelization opportunities across loops, function calls, and if-statements. Kremlin, DiscoPop, and Alchemist attempt to relieve their users by minimizing required interaction. However, such interaction is often essential to identify effective parallelization scenarios by exploiting developer's knowledge.

The SUIF Explorer [15] is among the first parallelization tools that incorporate user interaction. The system includes a compiler that automatically parallelizes loops without loop-carried dependencies. In addition, profiling, dynamic analysis and program slicing is used to pinpoint sequential loops. By adding annotations to the source-code, the compiler is able to consequently parallelize these loops. The Intel Advisor [11] and Pospector [12] go beyond loop parallelization. They perform an initial hotspot detection, let the user annotate potential loops or tasks for parallel execution, and apply dependency analysis of the annotated regions by executing the code. One drawback of Intel Advisor and Prospector is that they require recompilation of the annotated source-code, which may be infeasible in some industrial settings.

Similar to Parceive, all of the mentioned approaches attempt to identify parallelization scenarios. However, our proposed tool additionally considers architectural aspects (e.g., dependencies between components) and provides comprehensive visualizations of various information from different perspectives.

Software Architecture Reconstruction (SAR). Although there is a substantial body of studies published [6], we are not aware of other approaches addressing SAR relative to software parallelization. Here, the usage of static debug information, dynamic execution traces, and human expertise enables, to the best of our knowledge, unique analyses, such as data dependence analysis across arbitrary components. In addition, our rule-based language allows an efficient and precise SAR process at industrial scale.

According to the categorization by Ducasse and Pollet [6], our proposed approach mainly targets understanding, analysis, and conformance checking. Understanding is the primary goal of SAR approaches, which are mostly based on an extract-abstract-present cycle [24] (as our approach). Analysis frameworks may steer SAR frameworks to expose intricate system relations and quality attributes. For example, QADSAR [23] applies user-defined performance characteristics to an architectural model supporting analytical what-if scenarios. In the approach of Arias et al. [2], a meta-model provides explicit guidance for the reconstruction process to analyze runtime interactions at software architecture level. In contrast, our proposed analysis framework enables tailored analysis of architecture quality properties without requiring tedious modeling effort. Conformance checking is a popular goal of SAR to check discrepancies between the conceptual and the concrete architectures. Recent approaches increasingly incorporate behavior information for conformance checking [7, 19, 22]. Although Parceive does currently not support explicit models for conformance checking, such as the reflexion model [17], the analysis framework enables a top-down approach by querying arbitrary structural and behavioral information.

Software Visualization. Historically, trace visualization have been used to analyze software parallelism [13]. Vampir and HPCToolkit are tools from the HPC community that provide visualization environments for performance data [1, 18]. Such visualizations enable detailed understanding of dynamic processes in massively parallel systems. These tools focus on parallel software to highlight performance and communication issues; however, Parceive attempts

to support the parallelization of legacy applications by considering data dependencies.

Many SAR approaches support both static and dynamic visualization of software architecture components at varying levels of abstraction [6]. Related work offers highly interactive views that are tailorable to specific stakeholders [7, 8, 16, 19]. For example, Softwarenaut [16] enables recovery of architectural views from a software system through interactive exploration of its hierarchical decomposition. Softwarenaut provides a graph-based representation of artifact nodes and their relationships. Compared to the Architecture View of Parceive, Softwarenaut's view additionally supports evolutionary program analysis, has advanced interaction features (e.g., grouping and highlighting), and allows collaborative SAR sessions. However, Softwarenaut requires pre-processed relationship information, which makes it inappropriate for showing intricate dependencies that must be queried from large trace data.

3 PARCEIVE

In this section, we describe Parceive, a tool that supports the identification and analysis of parallelization scenarios in existing applications. The key to successful parallelization, which often involves extensive refactoring, is profound understanding of the target software. Parceive helps users understand the static and dynamic aspects of their software. The tool collects data about specific events during execution. This data is then combined with information about the underlying software architecture to provide valuable insights at higher level of abstraction. Finally, a set of specialized analyses and visualizations emphasize interesting aspects regarding parallelization. This enables Parceive users to obtain knowledge about their software's behavior, its structure, and the existing dependencies between the software's entities. This knowledge can be used to incrementally refine different parallelization scenarios. In the following, we describe Parceive's architecture and explain its main components.

3.1 Overview

Parceive is a standalone tool that runs on multiple platforms. Currently, it can analyze applications written in C/C++ that are provided as executables [1]. To capture the dynamic behavior of a program, the analyses operate at a binary level. Differing from static approaches that analyze source code, this approach can handle typical problems associated with parallelization, such as pointer aliasing and dynamic memory. Note that the executables must provide debug information to enable mapping between the machine code and the source code. This mapping enables Parceive to reconstruct identifiers from the analysis results, which is critical for understanding software.

The Parceive architecture comprises a backend and frontend (Figure 2). The backend contains a tracing tool that collects data about events relevant to parallelization during execution, such as function calls and memory accesses. The frontend includes a tool for software architecture reconstruction and a visualization component. The former applies a set of abstraction rules to the debug information. Here, the objective is to extract parts of the underlying software architecture from low-level binary information. The visualization component provides a graphical user interface built on the web-based Electron framework[2], which ensures cross-platform portability. The graphical user interface is used to manage and initiate analyses, configure software architecture reconstruction, and interact with the visualizations via multiple views.

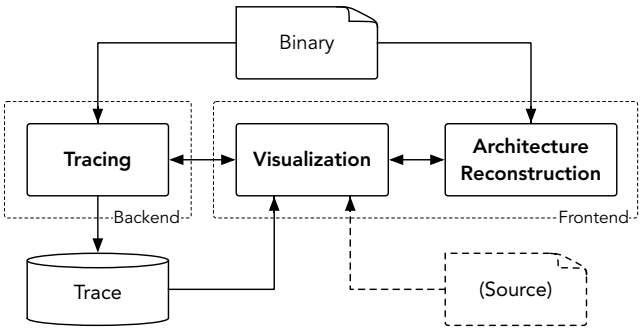

Figure 2: Main components of Parceive

3.2 Tracing

Our approach relies on binary analysis to obtain runtime information, which could be obtained via sampling or tracing. Sampling periodically inspects program execution and aggregates the collected data, whereas tracing triggers such inspections on predefined events. Note that sampling incurs less overhead than tracing; however, it is not sufficiently rigorous to capture all relevant runtime events conclusively. Thus, we implemented a tracing tool that utilizes dynamic binary instrumentation [26]. This allows the insertion or manipulation of machine instructions during runtime. Operating on binary files is often the only practical approach to analyze large industrial applications. Other approaches, such as compiler-based instrumentation, require special compilers and adaptations of the build system, which is not necessarily feasible in many project environments.

The tracing tool operates in three phases. In the initial phase, prior to instrumentation of the machine code, symbols are extracted from from debug information. These symbols are used to resolve variable names for accessed memory locations and link the dynamic analysis results to entities in the extracted software architecture. During the second phase, the tracing tool instruments the application binaries. This instrumentation adds callbacks to analysis functions that will be triggered by certain events. In the final phase, the application is executed and monitored by the tracing tool. The analysis functions track the control and data flow, and record relevant runtime information. When the execution is complete, the resulting trace data is used as input for the visualization.

Parceive stores trace data in a relational database whose model must satisfy two conditions. First, it must facilitate efficient recording of runtime information, i.e., writing to the database should have low overhead and consume limited memory. Second, it must contain all relevant information required to identify parallelization scenarios, such as precise information about the executed call paths

[1]The design of Parceive is largely language-agnostic and can be extended to support other programming languages.

[2]https://electron.atom.io/

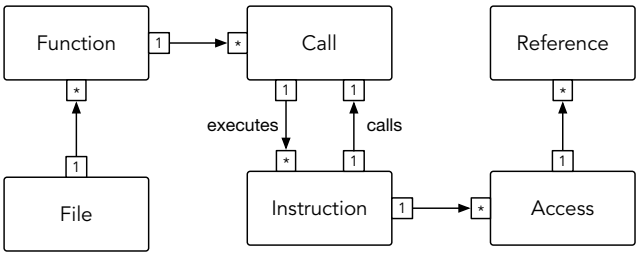

Figure 3: Meta-model for storing trace information

and memory accesses. The meta-model for storing trace data is shown in Figure 3. The entity attributes include runtime data (e.g., trip counts for loops), symbol information (e.g., variable names), and runtime information (e.g., the execution times of function calls). Our tailored analyses require a relational structure that support arbitrary queries on the entities, which is key for many use cases.

3.3 Visualization

As mentioned in Section 3.1, Parceive enables users to analyze their applications in an interactive manner via multiple views. The analyses are organized into projects and runs. Here, each project represents a set of runs for a given application, where a run represents a single execution of the application with individual input arguments and analysis parameters. The visualization interoperates with the tracing tool to initiate execution of the application. After successful execution, the visualizations show different aspects of the analysis results. Each view utilizes an integrated framework that enables efficient loading of runtime information and interaction between views [25]. In the following, we briefly describe the views (except for the Architecture View, which is described in Section 3.4) and their roles in the proposed approach.

Performance View. The Performance View shows the functions called during execution and their execution times (Figure 5). The primary purpose of the Performance View is to identify relevant components of the application in terms of parallelization benefit. The function calls are arranged chronologically from left to right. Here, the hierarchy from top to bottom depicts their positions in the call stack. Users can zoom into arbitrary periods of the timeline to explore even short call paths. Moreover, the Performance View allows multiple calls to be selected and spotted in other views, e.g., to show accesses to shared variables. Here, spotting means to highlight the selected call nodes (or representative nodes) and to hide the remaining nodes in the other views to enable a succinct representation.

CCT View. The Calling Context Tree (CCT) View targets comprehension of the dynamic function call behavior of an application. The CCT View displays a tree that represents individual function calls and their accessed memory locations (Figure 6) [3]. Users can arbitrarily expand and collapse function calls to traverse the executed call paths. When a function is called multiple times in the same context, the calls are merged into groups, which reduce the number of elements to display. However, call groups can also be decomposed into single calls. The vertical arrangement of calls represents their relative execution order during runtime.

The most important use case of the CCT View relative to parallelization is the data dependency analysis. Users are often interested in the existence and location of shared data in their software. Such dependencies hamper parallelization because they can lead to race conditions. Thus, Parceive provides optimized queries to detect accesses to shared variables across arbitrarily deep call hierarchies. In the first step, the queries build a set of memory accesses for each subtree rooted by the selected function call. In the second step, the intersection of each pair of sets is computed to obtain the actual dependencies. Furthermore, these dependencies can be inspected in a separate list view that shows the call stack of each access to identify their root causes.

Source Code View. The Source Code View shows the application source code (if available). The usefulness of this view becomes apparent when it communicates with the other views. Whenever the user selects an element, the corresponding location is displayed in the Source Code View. For example, when the user clicks on a function call node in the Performance View, the Source Code View jumps to the corresponding function, and when the user clicks on a memory access node in the CCT View, the Source Code View displays the accessing instruction. Thus, the Source Code View makes it easy to follow the program's execution and understand non-intuitive data dependencies.

3.4 Software Architecture Reconstruction

The proposed approach focuses on program comprehension at various levels of abstraction, ranging from machine-code instructions to software architecture. This facilitates establishing a wholistic view of the structural and behavioral aspects of software systems. Therefore, we link the runtime information to the software architecture. The software architecture acts as a shared mental model that facilitates a top-down approach for software comprehension, which is crucial to identify scalable parallelization scenarios.

Industrial software projects differ widely in terms of architectural and implementation styles. The selection of the concrete software architecture largely depends on the experience and preferences of the project members. Analysis tools, such as Parceive, must be able to handle such high variability. Thus, we consider software architecture as a model comprising multiple user-defined perspectives. Here, each perspective is represented by a set of software artifacts that are subject to custom architecture rules. These rules map the knowledge of the architects and developers to extract a concrete perspective of the software architecture as implemented.

We extract software artifacts by utilizing Parceive's ability to parse debug symbols. Such debug symbols include data about various software entities and their relationships, e.g., namespaces, classes, functions, and variables. Once the entities are identified, they can be mapped to artifacts according to some given architecture rules. The entity hierarchy is maintained by allowing artifacts to contain sub-artifacts. In addition, artifacts can be grouped together to form higher-level artifacts. Using the debug symbols as an information source has two major advantages. First, it allows

[3] As memory locations cannot be integrated properly into a tree layout without cluttering, they are positioned using an unconstrained layout based on force simulation around the rest of the tree.

the reconstruction of software architectures without access to the entire source code, which is a common requirement in industrial environments that use many third-party libraries. Second, the symbol data contains information that is created after compilation. This filters out artifacts that are not considered due to compiler optimizations or it adds artifacts introduced by generic programming techniques (e.g., template instantiation).

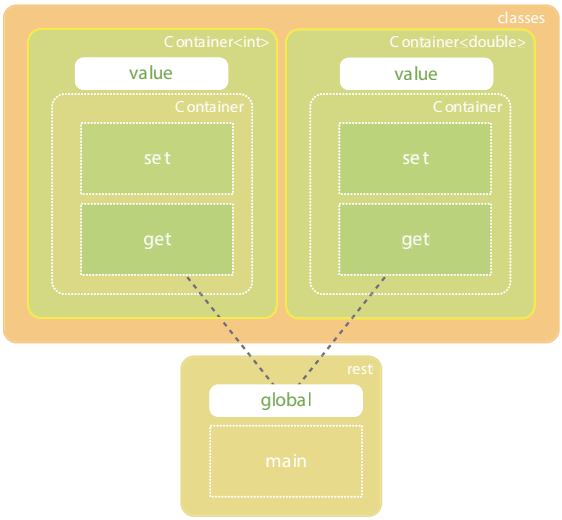

Figure 4: Architecture View showing artifact nodes and memory accesses of the global variable

The artifacts are visualized according to a node hierarchy in the *Architecture View* (Figure 4). Here, each node represents an extracted artifact. Initially, only top-level artifacts are shown; however, the user can expand any artifact to view the sub-artifacts. The node color indicates execution time. The execution time of a parent artifact is the sum of the inclusive times of its children artifacts. The lines between artifacts visualize different types of relationships that can be requested by the user, e.g., data dependencies, inheritances, instantiations, and call dependencies.

Architecture Rules. The artifact extraction process proceeds according to the architecture rules, which are formalized using a newly developed DSL that includes explicit and implicit rules. Note that the implicit rules are described later. Explicit rules are provided as as a list of statements in the following form.

<artifact_name> := *<rule>*
<artifact_name> = *<rule>*

Each statement defines a new artifact according to an atomic rule specified after the `:=` or `=` operator. The `:=` operator visualizes the created artifact, and the `=` operator defines an intermediate artifact that can be used in subsequent rules.

Note that atomic rules create a set of artifacts, which correspond to entities whose type is specified by a specific keyword and whose name matches the regular expressions given in the preceding brackets.

<entity_type> (" *<regex>* ")

Depending on the entity type, the following rules can be used.

- The `variable` rule maps each variable with a matching name to a new artifact.
- The `function` rule maps each function with a matching name to a new artifact, and it implicitly applies the variable rule to the function's local variables.
- The `class` rule maps each class with a matching name to a new artifact, and it implicitly applies the function rule to the methods and the variable rule to the member variables.
- The `namespace` rule maps namespaces with a matching name to a new artifact, and it implicitly applies the class rule to the contained classes and the function rule to the contained functions.
- The `infile` rule maps the entities contained in files with a matching filename to a new artifact, and it implicitly applies the class rule to the contained classes and the function rule to the contained functions.
- The `image` rule considers all executables or shared libraries with a matching name, and it implicitly applies the namespace, class, and function rule to the corresponding entities.

In addition, the following operators can be applied to atomic rules.

- The `!` operator is applied to an artifact or rule to create a new artifact from entities that do not have a mapping in the operand. Depending on the entity type, the corresponding atomic rule is applied.
- When applied to two operands, the `&&` operator iterates through the operand's artifacts. The result comprises the artifacts that are common to both operands.
- When applied to two operands, the `||` operator extracts artifacts found in both operands.
- The `[]` operator groups a set of given artifacts and forms a higher-level artifact.

Implicit rules are applied without user awareness. Typically, C/C++ files are divided into source and header files containing definitions and declarations. A potential consequence of this practice would be the creation of two artifacts that represent the same entity. Therefore, source and header files are grouped together. Due to template instantiation, the set of entities contains an entry for each template instance rather than a unique element representing the given function. Note that a higher-level artifact with a generic name is created whenever artifacts for the instances of such functions are to be created at the same level.

Example. Here, we demonstrate the power of the developed DSL and the Architecture View using a small program as input.

```
int global = 1;

template <typename T>
class Container {
    T value;
public:
    T get() { return value + global; }
    void set(T v) { value = v; }
};
```

Assume this class is instantiated with `int` and `double` arguments in the `main` function.

```
Container<int> c_i;
Container<double> c_d;
```

The architecture of this example is quite simple. It comprises the generic container logic in the `Container` class, the main function, and the global variable. We define the following rules to reconstruct the architecture.

```
classes := class(".*")
rest := infile(".*") && !classes
```

The first rule creates an artifact that contains one sub-artifact for each class. The second rule finds the differences between all entities and those with a mapping in the `classes` artifact. The resulting artifacts are shown in Figure 4. Note that the `classes` artifact contains two sub-artifacts that represent the two instantiations of the `Container` class. The dashed lines between the getters and the `global` variable indicate their shared memory accesses.

In another use case we may be interested in only the getters of the two instances and the global variable. In this case, we could use the following rule.

```
A := function("get") || variable("global")
```

4 CASE STUDY

Here, we demonstrate the proposed approach through an analysis of CppCheck[4], a widely used static analysis tool. The goal of this case study was to understand the underlying structure and runtime behavior of CppCheck to identify potential parallelization scenarios. Parceive assisted us in gradually refining our understanding of the relevant components and their dependencies. These findings were confirmed by the author of CppCheck, who illustrated the application's software architecture. This information led to the creation of a set of rules for the software architecture reconstruction step, which yielded valuable insights at higher levels of abstraction.

After describing CppCheck, we demonstrate the main steps that helped us identify parallelization scenarios for CppCheck and highlight the key findings. For each step, we begin with a brief discussion of our motivation and then focus on the case study. Note that the steps are not meant to be general guidelines for parallelization or the use of Parceive. Rather, they aim to support our claim that tackling the additional complexity introduced by parallelism requires good comprehension of an application's software architecture.

4.1 CppCheck

CppCheck is an open-source static analysis tool for the C and C++ programming languages. It is written in C++ and comprises roughly 200,000 lines of code. CppCheck supports a variety of checks that are performed statically at the source code level, such as coding standard conformance, identification of undefined behavior, and security issue checks. The project is actively maintained and used to analyze numerous popular applications. A famous example is the Linux Kernel, where CppCheck has found a number of bugs.

[4]http://cppcheck.sourceforge.net/

CppCheck accepts multiple source files as input and applies the same set of checks to each file. Users typically use CppCheck continuously during development to obtain quick feedback about potential issues. Thus, good overall performance is crucial relative to the acceptance of CppCheck. Parallelism is an obvious choice to reach performance goals; thus, we consider CppCheck to be a valid case study. We applied Parceive to a run of CppCheck analyzing all files in the supplied sample directory. In this run, 18 source files containing nine different issues (one "good" and one "bad" file per issue) were analyzed. We configured Parceive to consider all function calls and memory accesses, including static and shared libraries, such as the C++ standard library. With an optimized version of CppCheck (GCC 4.9, optimization flag -Og), the execution yielded 416 megabytes of trace data and took 32 seconds.

4.2 Step 1: Decomposition

A typical question arising at the beginning of parallelization is where to target focus when dealing with large software. Obviously, effective redesign must target the components of an application that would benefit most from parallelism. Here, the naive approach to find such components is hotspot detection, which identifies regions in the code that consume most of the runtime. However, detecting hotspots is insufficient without proper understanding of the problem domain. Parallelization must consider the tradeoff between costs (synchronization, etc.) and the amount of exploitable concurrency.

The next step is to decompose the problem into tasks that can be executed in parallel. For efficiency reasons, the granularity of the tasks should be considered carefully to avoid unnecessary overhead (e.g., overhead incurred by context switches between threads). It is also important to ensure that the problem is distributed evenly across the tasks to avoid load balancing issues. This step frequently includes a decomposition of the given input data into chunks. This data decomposition can be combined with task decomposition such that chunks are distributed systematically to the tasks.

CppCheck. First, we attempted to identify good candidates for task and data decomposition by inspecting the distribution of execution times. The Performance View revealed that the program execution can be separated into two parts (Figure 5), i.e., initial loading of the CppCheck library (~4%) and checking of the input files (~95%). The execution time for checking the input files varied for each of the 18 input files. However, the call sequences in this part are similar relative to execution time distribution. Generally, checking a file comprised some preprocessing (10% rel.), an initial check phase for normal tokens (~50% rel.), and a final check phase for simplified tokens (~40% rel.). This recurring pattern indicates that decomposition at the file level is a good candidate for parallelization in this case.

4.3 Step 2: Dependency Analysis

After decomposition, the tasks must be analyzed relative to dependencies. There are essentially two types of dependencies that are problematic for parallelization, i.e., control dependencies and data dependencies. A control dependency exists if a task's execution relies on the outcome of another task, which prevents simultaneous execution of both tasks. A data dependency exists if the same

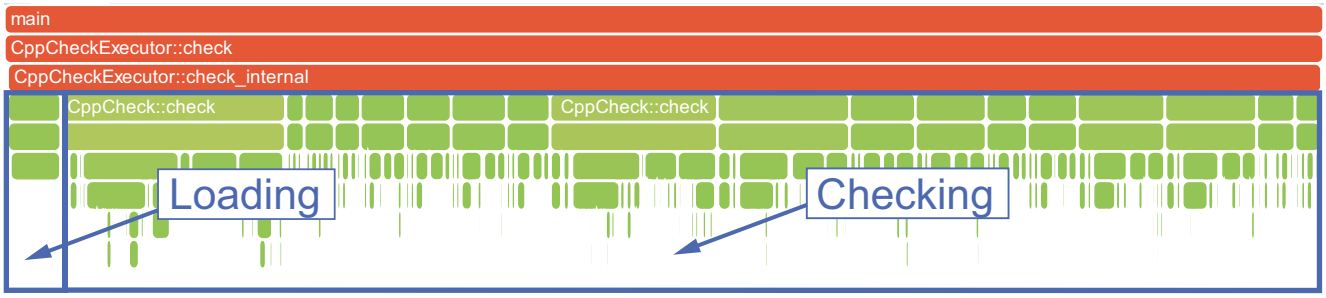

Figure 5: Performance View showing hierarchical function calls with sizes according to their execution times

memory location is accessed by different tasks. If at least one of these accesses is a write operation, this may result in race conditions. Synchronization (e.g., using locks) is one way to avoid race conditions in case of data dependencies. However, synchronization limits the scalability of an application due to wait times.

Finding data dependencies in real-world applications is difficult and one of the most challenging problems in the software industry. Programming languages such as C and C++ allow arbitrary, low-level access to memory locations and even permit pointer arithmetic. In such situations, static analysis tools cannot determine if some code may lead to data dependencies. In contrast, dynamic analysis tools provide dependable information for only a specific run. Although such information may be incomplete, it is very useful to evaluate potential parallelization scenarios. Unfortunately, detecting dependencies with dynamic analysis tools is often tedious. Usually, this requires prototypical parallelization followed by recompilation and dynamic threading analysis using various tools such as Helgrind[5]. However, Parceive solves this problem using built-in dependency analysis that can be triggered by selecting arbitrary function calls in the visualization.

CppCheck. In the second step, we refined the task parallelism scenario from the first step. Here, we identified locations to spawn the tasks. The CCT View revealed that the starting point for checking the input files is the `CppCheck::check` method. This method is called for each input file within a specific loop; thus, parallelizing that loop was identified as an option. A straightforward approach for loop parallelization is to execute each iteration by a separate task. However, a subsequent dependency analysis exposed memory locations that would be accessed concurrently (Figure 6), which could cause severe issues during runtime (e.g., segmentation faults). We found two origins for the dependencies by inspecting the memory-allocation instructions and the call stacks of the accesses (using the built-in interaction between the CCT View and the Source Code View). The first origin, which was caused by writes to the standard output, is the string implementation of the C++ standard library. The second origin, which was caused by accesses to a shared list in the `CppCheckExecutor` class, is the list implementation of the same library.

[5]http://valgrind.org/docs/manual/hg-manual.html

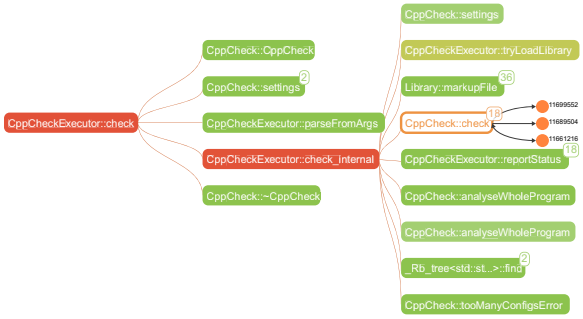

Figure 6: CCT View showing the calling context tree and accesses to shared memory locations

4.4 Step 3: Scenario Evaluation

At this point, the problem has been decomposed into tasks and dependencies have been identified. In other words, we have identified the tasks that may run in parallel, the data assigned to each task, and potential roadblocks. In the final step of the proposed approach, the parallelization scenario must be evaluated relative to effort, efficiency, and flexibility. In some cases, the design is straightforward and there is a single best implementation. In other cases, there are multiple solutions at different levels of abstraction. Typical examples are different data access patterns using fine- or coarse-grained synchronization, or using data duplication and reduction. Another typical use case occurs when a parallelization scenario involves multiple components of a given software architecture. Thus, analysis tools must help users evaluate the implications of the required software design changes.

CppCheck. After identifying the data dependencies between the tasks, we had to determine the parallelization strategy. From an operating system perspective, parallelism can be achieved using threads (i.e., shared memory parallelism) or processes. For our scenario, multi-threading means the execution of single checks in separate threads. Note that this requires synchronization of all shared accesses to avoid race conditions during execution. Dividing the work into multiple processes is to execute the file checks in separate address spaces. This strategy avoids race conditions by

design; however, it requires inter-process communication for CppCheck's logging facility (e.g., by using pipes). After inspecting all data dependencies in Step 2, we selected the multi-process scenario.

We evaluated the multi-process scenario using Parceive's Architecture View. A new issue caused by the duplicated memory regions for each child process was identified. In the present implementation, a single object of the `CppCheck` class is responsible for initiating all analyses by calling the virtual `check` methods of the derived analysis classes. The scenario would duplicate this object and, as a result, remove any existing dependency on the parent process. This would inevitably change the semantics of the application. Therefore, we had to check the `CppCheck` class for data dependencies on parts of the application that would be executed by different processes. The architecture view supports this task by displaying the memory accesses between arbitrary artifacts of the software. Thus, the tool can highlight data dependencies between the class and any other structural component, including those executed in the parent process.

Recall that rules to extract meaningful artifacts were defined in the former steps. The author of CppCheck confirmed most of the rules and helped us refine them. CppCheck's overall software architecture is divided into a command line interface (cli) and a library. The command line interface initializes the library and file handling. The library checks the input files and includes the following main components: the library class, the preprocessor, the single check classes, the symbol database, the settings manager, value flow handling, and token creation.

```
checks          = class("Check.*")
token           = class("Token.*")
preprocessor    = infile(".*preprocessor.*")
symbol          = infile(".*symboldatabase.*")
settings        = infile(".*settings.*")
valueflow       = namespace("ValueFlow")

cli            := infile(".*/cli/.*")
library        := [class("CppCheck"),
                   class("Library"), checks,
                   token, symbol, settings
                   preprocessor, valueflow]
```

After extracting the corresponding artifacts from the debug information, we investigated the memory accesses of the `CppCheck` class. We identified four (out of 26) methods that contained data dependencies to the artifact containing all check classes (Figure 7). Here, all accesses occured during file checking; thus, they are not an issue for this parallelization scenario. However, we also identified that all other artifacts in the library artifact are instantiated by the `CppCheck` object. This required analysis of these artifacts to determine possible dependencies on the command line interface. We found two dependencies, i.e., the same dependencies as we found in Step 2. This confirmed the parallelization scenario.

5 DISCUSSION

The above case study demonstrates how Parceive helps identify and evaluate parallelization scenarios for existing applications. The identified parallelization scenario is largely identical to what the author of CppCheck implemented in the Linux version of the tool.

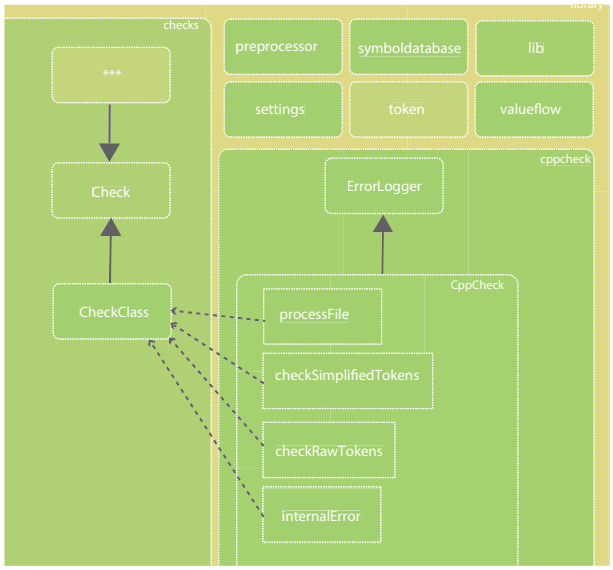

Figure 7: Section of the Architecture View showing the nested library artifact, inheritance relations, and memory accesses in `CppCheck`

Since the resulting parallelization is relatively coarse-grained, it scales well and can utilize multiple processing units efficiently.

However, the truly interesting findings are not related to the parallelization results. They were gained while conducting the described steps. In this case, an important benefit of Parceive was a progressive refinement of our understanding of the software. This gradually helped us understand both the behavior and the structure of CppCheck independent of the relevance of the information for parallelism. One example is flexible usage of dependency analysis at the architectural level and on call level. This revealed several relations, such as static or global variables, we would have missed by simply reading the source code. Another example is the coloring of artifacts in the Architecture View based on the recorded execution times. This feature identified parts of the application that were not executed during the analysis. One such component is the `ThreadExecutor` class that implements the multi-process scenario, which we identified.

Analyses that operate at the machine code level led to unexpected benefits. The displayed views gave analysis results originating from the executed binary, which may or may be not optimized. Using multiple runs with different optimization strategies and compiler flags, we could compare different compilation results. Some optimizations would influence the parallelization scenarios by removing entire call paths or memory accesses. For example, major differences were observed when the compiler used return value optimization or inlining. Another benefit of the binary analysis is the possibility of inspecting single template instantiations in C++ code. Source code analysis would not identify different variations of a template because instantiation occurs at compile time.

In addition to its use as an analysis tool, one additional aspect of Parceive that should not be underestimated is how it facilitates communication. During discussions with our colleagues, we noticed

that the visualizations are very helpful to explain various aspects of the software. As a result, we can describe the important aspects in an abstract manner separate from unimportant aspects of the software architecture to pinpoint bottlenecks in specific regions during runtime. In addition, the views and analyses allowed us to identify relationships between classes that are difficult to describe by simply looking at the source code.

The most notable limitation of our approach is the incompleteness of its dynamic analysis. The knowledge gained about the application's behavior must always be related to a specific execution. If some parts of the software were not executed, the analyses cannot identify any events in those unexecuted regions. Another restriction relative to the analysis of long-running executions is the slowdown and memory consumption incurred by Parceive. In some situations, this can make the analysis impossible because the input data is too extensive for visualization. Finally, comprehending the visualizations sometimes requires significant knowledge about the features of the target programming language and compiler. The visualizations show all nodes that exist in the machine code; thus, they may show implicit software entities that are not explicit to the target application's developer, which could result in misunderstanding of the analysis results.

6 CONCLUSION AND FUTURE WORK

Parallelization of industrial applications remains a demanding problem. Such applications typically comprise numerous components at several architecture levels. Proper comprehension of software, which is necessary for any serious redesign task, is impossible without effective and efficient tool support. In this paper, we have presented the Parceive tool and discussed how it can help to gain the important knowledge relative to parallelization. A case study demonstrated three major steps to identify a parallelization scenario based on task parallelism. Parceive enabled us to identify and decompose potential tasks, apply a dependency analysis on the respective software regions, and evaluate the parallelization scenario at a software architecture level. The analyses and visualizations gradually refined our understanding of the application, and helped us communicate about the software effectively with our colleagues.

We found the combination of behavior information and extracted software architecture information to be very useful relative to understanding the software. The resulting visualizations provided insights into the software components and their dependencies at various levels of abstraction. The interaction between different views enabled us to connect and relate different aspects of the software architecture and the application's behavior. We believe that Parceive demonstrates significant potential relative to utilizing combined information sources to broaden comprehension of software.

Note that we regularly apply Parceive to open-source legacy applications and to the proprietary software of our industry partner, which has led to various future improvements and extensions. One planned additional feature is the support for applications that already operate concurrently. Here, typical use cases could include identifying existing concurrency issues, such as race conditions, or improving the parallel design. Another important future task is performance optimization. We will attempt to minimize the slowdown incurred by the dynamic analysis to make the tool more practical when analyzing applications with long run times.

ACKNOWLEDGMENTS

We thank Daniel Marjamäki, the author of CppCheck, for his valuable insights into CppCheck's software architecture. We also thank our industry partner Siemens for supporting this research.

REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. Tallent. 2010. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience* (2010).
[2] C. Arias, A. Pierre, and A. Paris. 2013. A Top-Down Approach to Construct Execution Views of a Large Software-Intensive System. *Journal of Software: Evolution and Process* 25 (2013).
[3] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. 1993. Automatic Program Parallelization. *Proc. IEEE* (1993).
[4] F. Bodin and M. O'Boyle. 1996. A Compiler Strategy for Shared Virtual Memories. In *Languages, Compilers and Run-Time Systems for Scalable Computers*. Springer.
[5] M. Burke, R. Cytron, J. Ferrante, and W. Hsieh. 1989. Automatic Generation of Nested, fork-join Parallelism. *The Journal of Supercomputing* (1989).
[6] S. Ducasse and D. Pollet. 2009. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering* (2009).
[7] F. Fittkau, P. Stelzer, and W. Hasselbring. 2014. Live Visualization of Large Software Landscapes for Ensuring Architecture Conformance. In *ECSA*.
[8] K. Gallagher, A. Hatch, and M. Munro. 2008. Software Architecture Visualization: An Evaluation Framework and its Application. *IEEE Transactions on Software Engineering* 34 (2008).
[9] S. Garcia, D. Jeon, C. Louie, and M. Taylor. 2011. Kremlin: Rethinking and Rebooting gprof for the Multicore Age. In *ACM SIGPLAN Notices*. ACM.
[10] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. 2006. Semi-automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *International Journal of Parallel Programming* (2006).
[11] Intel. 2017. Parallel Studio XE 2018. (2017). https://software.intel.com/en-us/intel-parallel-studio-xe [Online; accessed Oct-2017].
[12] M. Kim, H. Kim, and C. Luk. 2010. Prospector: A Dynamic Data-Dependence Profiler to Help Parallel Programming. In *HotPar*.
[13] T. LeBlanc, J. Mellor-Crummey, and R. Fowler. 1990. Analyzing Parallel Program Executions using Multiple Views. *J. Parallel and Distrib. Comput.* (1990).
[14] Z. Li, R. Atre, Z. Huda, A. Jannesari, and F. Wolf. 2016. Unveiling Parallelization Opportunities in Sequential Programs. *Journal of Systems and Software* (2016).
[15] S. Liao, A. Diwan, R. Bosch Jr, A. Ghuloum, and M. Lam. 1999. *SUIF Explorer: An Interactive and Interprocedural Parallelizer*. ACM.
[16] M. Lungu, M. Lanza, and O. Nierstrasz. 2014. Evolutionary and Collaborative Software Architecture Recovery with Softwarenaut. *Science of Computer Programming* 79 (2014).
[17] G. Murphy, D. Notkin, and K. Sullivan. 1995. Software Reflexion Models: Bridging the Gap between Source and High-level Models. *ACM SIGSOFT Software Engineering Notes* 20 (1995).
[18] W. Nagel, A. Arnold, M. Weber, H. Hoppe, and K. Solchenbach. 1996. VAMPIR: Visualization and Analysis of MPI Resources.
[19] A. Nicolaescu, H. Lichter, A. Göringer, P. Alexander, and D. Le. 2015. The ARAMIS Workbench for Monitoring, Analysis and Visualization of Architectures based on Run-time Interactions. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*.
[20] G. Ramalingam. 1994. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.* (1994).
[21] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. 2012. Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications?. In *ISCA*.
[22] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. 2006. Discovering Architectures from Running Systems. *IEEE Transactions on Software Engineering* 32 (2006).
[23] C. Stoermer, A. Rowe, L. O'Brien, and C. Verhoef. 2006. Model-centric software architecture reconstruction. *Software: Practice and Experience* 36 (2006).
[24] A. Van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. 2004. Symphony: View-Driven Software Architecture Reconstruction. In *WICSA*.
[25] A. Wilhelm, V. Savu, E. Amadasun, M. Gerndt, and T. Schuele. 2016. A Visualization Framework for Parallelization. In *VISSOFT*.
[26] A. Wilhelm, B. Sharma, R. Malakar, T. Schuele, and M. Gerndt. 2015. Parceive: Interactive parallelization based on Dynamic Analysis. In *PCODA*.
[27] X. Zhang, A. Navabi, and S. Jagannathan. 2009. Alchemist: A Transparent Dependence Distance Profiling Infrastructure. In *CGO*. IEEE.