A Visualization Framework for Parallelization

Andreas Wilhelm, Victor Savu, Efe Amadasun, Michael Gerndt
Technische Unversität München
{wilhelma, savu, amadasun, gerndt}@in.tum.de

Tobias Schuele
Siemens Corporate Technology
tobias.schuele@siemens.com

***Abstract*—Since the advent of multicore processors, developers struggle with the parallelization of legacy software. Automatic methods are only appropriate to identify parallelism at instruction level or within simple loops. For most applications, however, a scalable refactoring requires profound comprehension of the underlying software architecture and its dynamic aspects. This leads to an increasing demand for interactive tools that foster parallelization at various granularity levels. To cope with this problem, we propose a visualization framework and three tailored views for parallelism detection. The framework is part of *Parceive*, a tool that utilizes dynamic binary instrumentation to trace C/C++ and C# programs. The cooperative views allow identification and analysis of potential parallelism scenarios using seamless navigation, abstraction, and filtering. In this paper, we motivate our approach, illustrate the architecture of the visualization framework, and highlight the key features of the views. A case study demonstrates the usefulness of Parceive.**

***Index Terms*—Parallelization, Trace analysis, Software visualization, Program comprehension.**

I. Introduction

Multicore processors still pose a big challenge to the industry and the research community. The increasing computing power of such processors can only be exploited by parallel software. As a result, software and system vendors are forced to parallelize their legacy applications to make them scalable. Such a parallelization relies on two essential steps. The first step is to comprehend the software's structure as well as its dynamic behavior. The second step is a potential redesign of the software to enable parallelism. Unfortunately, both steps are tedious and error-prone for which reason most legacy applications still operate sequentially. Besides the additional complexity of parallel programming, we believe the problem largely arises from a lack of appropriate tools that support developers and architects in analyzing their software.

In the field of reverse engineering, software analysis tools are invaluable to understand the structural and behavioral aspects of complex software. These tools gather necessary data using static (at compilation time) or dynamic (at runtime) analysis approaches and visualize this data for manual interpretation. Dynamic analysis yields precise information about concrete runtime events, whereas static analysis conservatively reasons over possible behaviors by examining the source code. It has been reported that hybrid approaches provide high accuracy and soundness of the analyses [4]. We implemented a tool called *Parceive* that combines static and dynamic analyses to provide effective views for parallelization [10].

In the following, we present the underlying visualization framework and selected views of Parceive. Our framework enables efficient analyses on large traces to answer elaborate user queries. We further support an easy integration of tailored views by providing a common interface to them. This allows to address a variety of viewpoints without writing highly specific analyses. Various optimization and abstraction techniques in the visualization infrastructure ensure responsiveness and scalability, e.g., by on-demand loading, caching, trace abstraction, and communication across views. The views we present enable users to detect hotspots, infer parallelization strategies, and validate these strategies regarding data dependencies.

II. Parceive

The vision behind Parceive is to help developers in identifying parallelism opportunities and obstacles at various granularity levels. It utilizes static binary analysis and dynamic instrumentation to collect trace data. Being less conservative than purely static approaches lets us focus on concurrency-related events, e.g., memory accesses, routine invocations, and object instantiations. By a-posteriori abstraction of such fine-grained information, we infer architectural aspects from the applications. The results can be used as a starting point for architecture redesign and refactoring. However, due to the inherent incompleteness of dynamic analysis, the user is responsible for correct parallelization.

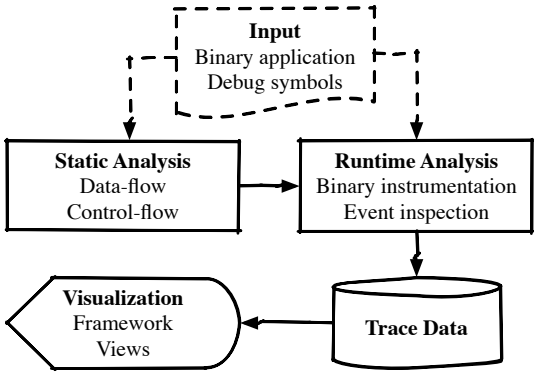

Fig. 1. The high-level components of Parceive.

Figure 1 depicts the fundamental components of Parceive and their relations. Our analyses operate on executables to retrieve the required information. These analyses can be classified into static and dynamic ones. The former inspect the data- and control-flow of single linkage objects. By incorporating debug symbols, static analysis enables our tool to gather

information about variable accesses, loop constructs, and class hierarchies. Additionally, the gathered information is used to restrict the scope of subsequent runtime analysis in order to reduce the execution overhead.

The runtime analysis instruments and inspects predefined events during execution of an application, e.g., object instantiations, method invocations, and thread handling (in case of multi-threaded applications). During such events, Parceive collects trace data and stores it in an SQL database. The database scheme enables highly specific and performant queries for different visualizations which are key for program comprehension and parallelization. Each view thereby simplifies and highlights specific aspects of the traced software.

III. Visualization Infrastructure

In this section, we present certain key aspects of our web-based visualization infrastructure. This infrastructure facilitates the integration of arbitrary views to Parceive by providing a common interface for accessing abstracted runtime traces. The biggest challenge when dealing with traces is the potentially overwhelming amount of data. This often leads to unmanageable views with unacceptable delays. Our visualization infrastructure addresses this problem by building upon a reactive client-server architecture (see Figure 2) that provides four key services: (a) trace optimization, (b) on-demand loading, (c) caching and pipelining, and (d) state management and communication.

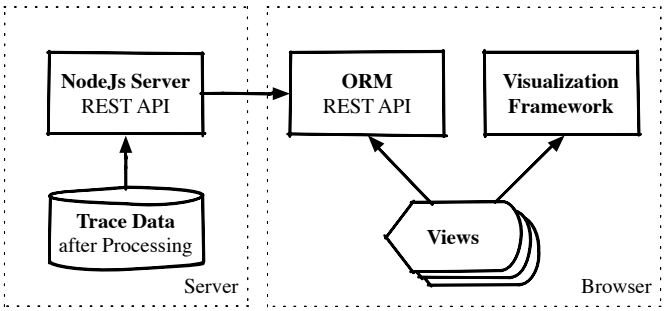

Fig. 2. The visualization infrastructure of Parceive.

a) Trace Optimization: As mentioned previously, Parceive uses a database to store trace data. Its layout is tuned for write operations to reduce the runtime overhead. All the information required by the views can be obtained via database queries. However, most of the read operations would take a disproportionate amount of time to complete without applying certain optimizations on the trace database. Hence, we perform a set of optimizations during a post-processing step to reduce lookup times for views.

The most important optimization is the generation of table indices for efficiently searching the database. Another optimization is the creation of intermediate tables to avoid expensive joins for most queries. Although this leads to some redundancy, it does not increase the overall complexity. Furthermore, reducing fragmentation within tables increases data locality and thus speeds up queries. This has considerable effect on queries that require a full table scan and also reduces the size of the trace database.

b) On-Demand Loading: On-demand loading of trace data improves responsiveness of the visualization. Often, loading entire traces into the browser is not feasible due to memory restrictions. To solve this problem, a NodeJS server has been developed that reads data on demand. The server provides a REST [5] API to manage retrieval of data. For security reasons, all SQL queries are performed by the server without supporting arbitrary queries. The implementation makes use of multiple parallel reads to the database to reduce the latency and to increase the throughput when large amounts of data are requested by the views. This lets users seamlessly explore and navigate through software producing a potentially unmanageable amount of trace data.

c) Caching and Pipelining: At the client side, we provide an Object Relational Mapper (ORM) module to simplify development and improve responsiveness. This module enables accesses to predefined entities and manages the relationships between them. The API is implemented using promises [2] which simplify asynchronous and parallel access. The greatest benefit for views using the ORM are optimizations for data loading, the most important ones being caching and pipelining. Caching avoids repeated loading of data that was accessed before, and pipelining combines multiple queries to the same endpoint into a single one. When requesting a large number of entities, pipelining heavily improves the throughput with only small latency overhead.

d) State Management and Communication: The visualization framework provides global state management and communication facilities to views. The former is based on a centralized and persistent state storage that retains the state of views across page loads. Currently, the view layout and the marked nodes are automatically stored as part of the state. In addition, each view can save tailored information at any time and retrieve it during rendering. Local storage is used to keep all the state information making it persistent. This service reduces computational effort for views by reusing results across arbitrary UI events.

The communication service enables arbitrary views to interact by triggering predefined events. These events let users explore their applications with synchronized representations from complementary viewpoints. One example is simultaneous highlighting of entities such as functions in different views. Another example is spotting of distinct entities for further inspection in separate views. This way, the number of nodes to be displayed in a view is reduced which increases scalability. Currently there the following types of communication provided:

- *Focusing* brings entities to the attention of the user by centering the representations in all views.
- *Marking* allows to communicate selected entities between views.
- *Spotting* replaces the shown entities in views by a new collection of entities.

- *Hovering* highlights entities in multiple views by reducing the opacity of all other entities.

IV. Views

In this section, we present three prototypical views built upon the presented visualization infrastructure. Therefore, The interaction of the views provide a scalable top-down approach for identifying hotspots, developing parallelization strategies, and validating these strategies w.r.t. data dependencies.

A. Performance View

The performance view is an interactive representation of a program's profiling and trace data. Its primary purpose is to assist users in identifying scenarios that may benefit from parallelization. A holistic visualization of a program's runtime behavior helps users to spot potential performance and to guide optimization efforts.

There are two visualization modes in the performance view: tracing and profiling. The tracing mode represents an icicle plot augmented with loop information, i.e., a hierarchical view of calls made during program execution. The calls are arranged chronologically from left to right, and the visibility of loop information can be toggled (on or off). Trace visualization is useful for detailed examination of a program, especially when the traced invocation order is important. The profiling mode presents the user a hierarchical view of the functions called during execution. The length of each function in the view is given by the sum of the calls' duration which is often sufficient to quickly pinpoint hot spots.

With growing size and complexity of a program, the performance data also increases, making it harder to digest it at once. For this reason, the performance view provides zooming at call level and runtime duration filtering. The latter sets the minimum runtime duration required for a call to be loaded. The minimum value depends on the percentage of the runtime duration of the current top-level call in the view. This allows to only render calls large enough to be easily visible. With the call zooming feature, users can focus on a specific call, which makes it the top-level call, recomputes a new minimum value, and loads child calls of the focused call that weren't visible before.

B. Calling Context Tree (CCT) View

The CCT view targets comprehension of the dynamic behavior of an application. It displays a call tree consisting of call nodes, loop nodes, and memory nodes (see Figure 4). Nodes for calls (or groups of calls), loop executions, and loop iterations are positioned using a tidy tree layout [9]. Children of nodes are vertically sorted by their start time to reflect their actual execution. Memory nodes accessed by arbitrary tree nodes are difficult to integrate in a tree layout. Therefore, they are positioned using an unconstrained layout based on a force simulation around the rest of the tree.

The first node present when the view is created is the call to the `main` function. Users can arbitrarily expand and collapse call nodes. When a function was called multiple times during the same function execution, the respective call nodes are merged to so-called *call groups*. Call groups reduce the number of nodes to be displayed but can also be decomposed into their single call nodes. Navigating through loop executions and loop iterations is similar to calls and allows the user to see information at any desired granularity.

The most important feature of the CCT view for parallelization is data dependency analysis. Users are often interested in the existence and the location of such dependencies between arbitrary regions of their software. Therefore, an optimized query of the visualization infrastructure is utilized to detect shared memory accesses across deep call hierarchies. Found dependencies can then be inspected in a separate source code view. The described feature allows manageable visualization by dramatically reducing the amount of shown nodes. There are some additional features that aim for better scalability and help developers parallelizing their code:

- Profiling information (relative execution time) is reflected by node colors.
- Expanding and collapsing nodes can be performed in both directions to show and hide parent or child nodes.
- Seamless zooming or panning, and focusing on single entity nodes for a clear visualization.
- Spotting of arbitrary call nodes that automatically expands the call tree up to these nodes and start with their common ancestor.

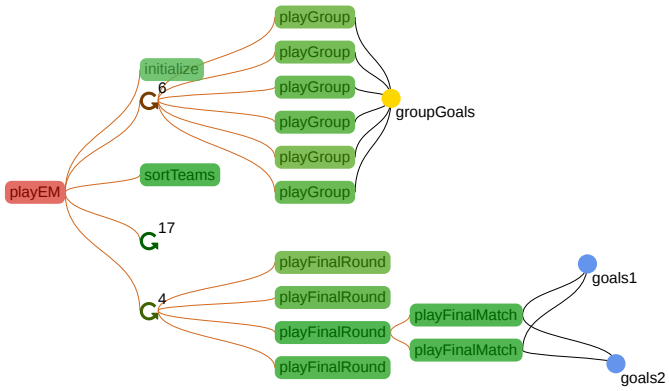

Fig. 4. The calling context tree view of Parceive.

C. Source View

The source view shows the highlighted source code that was used to build the instrumented application. The usefulness of this view becomes apparent when it is communicating with the other views presented in this paper. The simplest form of interaction is focusing, where the source view display the definition of functions, loops, and memory accesses, which makes it easy to follow the execution of a program trough the source code. Hovering provides additional information—for calls it indicates where the call originated, and for memory references where they were allocated and referenced.

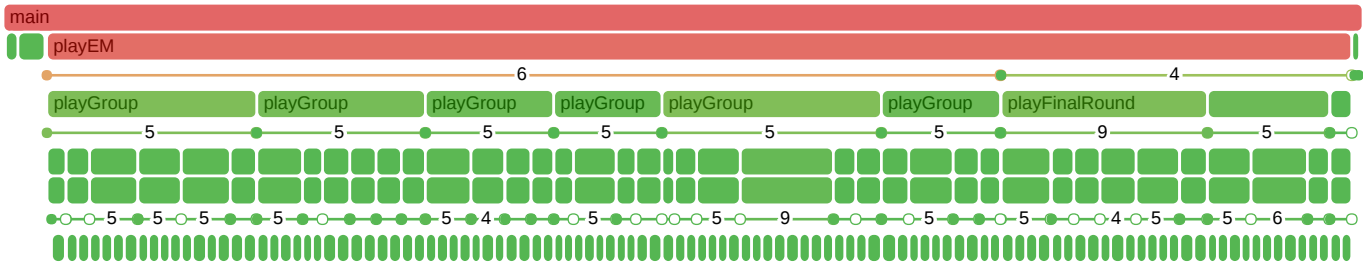

Fig. 3. The performance view of Parceive showing an EMSim trace.

V. Case Study: EMSim

In this section, we demonstrate Parceive by applying it on EMSim, a simulator for the european soccer championship. EMSim was part of the student assignments for our Master course on parallel programming. The application simulates every match of the group and final phase by relying on historical statistics of matches, teams, and players. The task was to parallelize the sequential C code, by considering the following steps: (1) the comprehension of EMSim and its behaviour, (2) the location of hotspots, (3) the identification of dependencies, and (4) the parallelization of potential code regions with appropriate libraries. When done, the students were able to upload their source code on our submission server to check for correctness and speedup. As it turned out, the biggest challenges were to estimate the load balance and to identify data dependencies. In the following, we show how Parceive would have helped the students with both issues.

In listing 1, we present an excerpt of the EMSim code that contains a major region for parallelism. The shown `playEM()` function gets multiple arrays of teams as input, one for each group of the championship. Herein, the loop in line 7 calls `playGroup()` for every group, which plays all group matches between the given teams and returns the first, second, and third winners. This loop accounts for ~60% of the overall runtime of an execution. Even though students might spot the loop as a potential opportunity for parallelism, the crux is to analyze dependencies between the calls. The first challenge is to understand the pointer arithmetic in line 10 and 11, which scatter qualified teams to the successor array. The second challenge is to inspect the (arbitrary deep) call hierarchies for shared memory accesses.

```
// play groups
for (int g = 0; g < NUM_GROUPS; ++g) {
  playGroup(
    teams + g,
    successors + (g * 2),
    successors + (NUM_SUCCESSORS - (g * 2) - 1),
    bestThirds + g
  );
}
```

Listing 1. A major region for loop-parallelism in the EMSim code

We applied Parceive on EMSim by tracing the optimized user binaries (gcc compiler flag `-og`). Other libraries that are used by EMSim were not instrumented since we can preclude conflicting accesses with the user application. The instrumentation framework and the analyses cause an execution slowdown of factor ~4.4 (20s instead of 4,5s). The produced trace data has a size of 4.3Mb, where data about memory accesses and function invocations account for 84% of the size. After producing the trace database, we can import it to our visualization and show the resulting views.

The trace view that depicts the EMSim execution is shown on the upper part of Figure 3. By doing a top-down inspection, it becomes obvious that `main()` spends most of the time in the invocation of `playEM()`. Within this function call, the execution time is mainly spent in two loops. The first loop (the one from Listing 1) performs the group phase. The second loop plays the final rounds, starting from the round of 16 to the final match.

Recall the two main purposes of the trace view, identifying hotspots and estimating load balance. In the case of EMSim, the view indicates the invocations of `playGroup()` and `playFinalRound` as significant hotspots. Without considering dependencies, a naive strategy could invoke the single calls by individual threads. Besides limited scalability (there are only 6 calls of `playGroup()`), this solution results in a non-optimal load balance since the calls vary in their execution time. But the trace view exposes another opportunity for parallelism. Both the group matches and the final matches lead to distinct calls of `playMatchGeneral()`. Hence, a second strategy is to use a master-worker pattern for scheduling these calls tron a thread-pool. The next step for the user is to validate the found parallelization strategies.

The CCT view supports users with the validation of prospective system redesigns. We present the resulting view for EMSim in Figure 3. Notice that only relevant call nodes for the investigated parallelization strategies are shown by spotting them on the trace view. Beginning with the first strategy, we are interested in possible data dependencies between the invocations of `playGroup()` and `playFinalMatch()`, respectively. The corresponding analysis (*show deep dependencies*) queries shared accesses among full call hierarchies. It turns out that each invocation of `playGroup()` accesses the global variable `groupGoals` and each of invocation of `playFinalMatch()` accesses the stack variables `goals1` and `goals2`. Furthermore, the cct view enables to localize these memory accesses such that the dependencies can be investigated and synchronized.

Even though the data dependencies for the first parallelization strategy can be easily avoided, we focus on the other strategy that put the calls to `playMatchGeneral()` in distinct tasks. The respective CCT view is shown in Figure 3. Performing the same dependency analysis as before returns no shared memory accesses between the calls. Hence, the user gained the necessary information to parallelize EMSim by integrating the master-worker pattern. A student solution achieved a speedup of 6.5 on our 8-core system. Considering the sequential pre- and post-processing regions, and the control dependent matches of the final phase (e.g., the semi-final has to be played before the final), this speedup confirms Amdahl's law. This result concludes that our views are very helpful to parallelize user applications.

VI. Related Work

Trace visualization has a long history for analyzing software parallelism [7]. Current visualizations are able to handle up to 10^7 events or terabytes of trace data [6]. However, most of the cited methods achieve these numbers only for statistical plots or by averaging pixels. Others gain scalable views at the expense of limited usability, e.g., by requiring users to extensively pan across detailed trace data. Our tool tries to create the sweet spot between full abstraction and largely unprocessed details by a more interactive approach.

Vampir and HPCToolkit are well known tools from the performance community. They provide visualization environments to represent performance data [1], [8]. This enables a detailed understanding of dynamic processes on massively parallel systems. Vampir handles terabytes of data via parallel1 preprocessing and data reduction techniques during collection. HPCToolkit maintains interactivity of its views by sampling the data rather than on-demand processing during panning and zooming. Both tools focus on parallel software to highlight performance and communication issues. In contrast, the intention Parceive is to aid with parallelization of legacy applications.

When looking at visualizations of reverse engineering frameworks, Moose certainly belongs to the most prominent ones [3]. It provides a set of services including a common meta-model, metric evaluations, a model repository, and generic GUI support for querying, browsing and grouping. Metric results can arbitrarily visualized by a set of views, e.g., polymetric views, evolution matrices, or butterfly views. Though the functionality of Moose by far exceeds the one of our tool, we often rely on highly optimized search strategies (e. g., deep dependency analysis) and individual views.

VII. Future Work

Parceive was developed using a data-centric approach, where the initial project phase focused on fine-grained trace data and its visualization. In the second project phase, we will apply abstraction on this data to gain architectural insights for software comprehension and parallelization. Additional knowledge from software architects and developers will be incorporated to specify software components. This knowledge can then be used for extended views that depict the structure of systems and their behaviour on architecture level.

Currently, we are working on multiple features that improve the usability of Parceive, or even extend possible uses. One such feature is a source code view that can focus on arbitrary regions of user code. This view allows to link nodes from any visualization to the respective definitions or usages. Another feature we are working on are views that depict parallel behavior of multi-threaded applications. Possible use-cases are scalability analysis or validation, e.g., to identify possible race conditions.

We reviewed the visualization component on a few open and closed source applications, some of them are at industrial scale. Main attention has been paid to the scalability of the framework and to the effectiveness of the views. However, there are more tests necessary to measure performance and responsiveness, and to elicit possible scaling problems. We plan to focus on an extensive set of sequential user applications with decent parallelism potential.

VIII. Conclusion

Visualization of software behavior is a crucial instrument to comprehend software systems. Unfortunately, this task often involves an overwhelming amount of trace data that easily overstrains current UIs. Therefore, we built a highly scalable and responsive visualization infrastructure for our tracing tool Parceive. The core is a client-server architecture providing asynchronous event driven data retrieval and optimized analysis queries. These techniques empower users to explore and navigate their traced applications by multiple viewpoints. This paper additionally presented three views that aids with identification and validation of potential parallelism scenarios. Currently, we are in an early stage of research. Future work will mainly cover higher-level aspects to comprehend software systems in a top-down approach.

Acknowledgements

We thank Nathaniel Knapp for comments and suggestions. Additionally, we gratefully acknowledge the support of Siemens. This work has been supported by the German Federal Ministry of Education and Research (Software Campus, grant no. 01IS12051).

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] B. Cavalier and D. Denicola. Promises/a+. Technical report.

[3] S. Ducasse, T. Gırba, M. Lanza, and S. Demeyer. Moose: A Collaborative and Extensible Reengineering Environment. *Tools for Software Maintenance and Reengineering, RCOST/Software Technology Series*, 71:27, 2005.

[4] M. D. Ernst. Static and Dynamic Analysis: Synergy and Duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[5] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[6] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer. State of the Art of Performance Visualization. *EuroVis 2014*, 2014.

[7] T. J. LeBlanc, J. M. Mellor-Crummey, and R. J. Fowler. Analyzing Parallel Program Executions using Multiple Views. *Journal of Parallel and Distributed Computing*, 9(2):203–217, 1990.

[8] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. 1996.

[9] E. M. Reingold and J. S. Tilford. Tidier Drawings of Trees. *IEEE Transactions on Software Engineering*, (2):223–228, 1981.

[10] A. Wilhelm, S. Bharatkumar, M. Ranajoy, T. Schüle, and M. Gerndt. Parceive: Interactive Parallelization based on Dynamic Analysis. In *Program Comprehension through Dynamic Analysis (PCODA), 2015 IEEE 6th International Workshop on*, pages 1–6. IEEE, 2015.