An Architectural Template for Parallel Loops and Sections

Markus Frank
markus.frank@iste.uni-stuttgart.de
University of Stuttgart
Universitätsstraße 38
70569 Stuttgart

Alireza Hakamian
alireza.hakamian@iste.uni-stuttgart.de
University of Stuttgart
Universitätsstraße 38
70569 Stuttgart

Abstract

The **P**alladio **C**omponent **M**odel uses UML-like diagrams to specify architectural software designs, which are used for early design-time analyses of software performance metrics. As a current drawback of the PCM, it does not support the specification of massive parallel software behaviour like OpenMP parallel loops. For **S**oftware **P**erformance **E**ngineers this results in complex modelling workarounds, or it is not possible to model the software's behaviour at all, which results in inaccurate analyses and semantic discrepancies.

In this paper, we present a light-weight PCM metamodel extension, allowing SPEs to easily annotate parallel sections (similar to OpenMP) in their software specifications. This significantly reduces the modelling effort through automation.

1 Introduction

Software **P**erformance **E**ngineers evaluate quality attributes (like response time) of software systems based on architectural models during early design-time. They use model-based approaches to simulate the software's behavior and resource consumption. One of the sophisticated approaches is the PCM approach [1] [1]. It uses UML-like diagrams to model software, hardware, and user behavior.

However, the PCM modeling language is not well suited for massive parallel software behavior known from, i.e., OpenMP parallel loops [2]. So, in order to reflect parallel loops in the software models SPEs have to manually model each thread—even though they are identical or similar—by hand. This does not only require lot of effort but is also a very error-prone process.

In this paper, we introduce a new Architectural Template for parallel loops, which allows software architects to apply reusable parallel patterns to their Palladio models [3]. With the parallel loop architectural template, SPEs can easily specify parallel loops and sections in their software models. Further, we enriched the new Architectural Template with additional properties to specify the resource demand arising from thread and synchronization overhead.

[1]Palladio Component Model: www.palladio-simulator.com

In the course of the paper, we describe parallel loops and sections (see. Sec. 2), introduce the parallel loop architectural template (see Sec. 3), and discuss limitations and future work (see Sec. 4).

2 Parallel Loops and Sections

In this section, we shortly introduce our running example—a matrix multiplication—and show how one can parallelise it with OpenMP. In the following, we show how the **S**ervice **Eff**ect Specification of the same example looks like. We use matrix multiplication as running example since it is easily parallelisable as it has no dependencies between worker threads and workload is equally distributed. For the same reason we choose OpenMP[2] as paradigm for parallelisation.

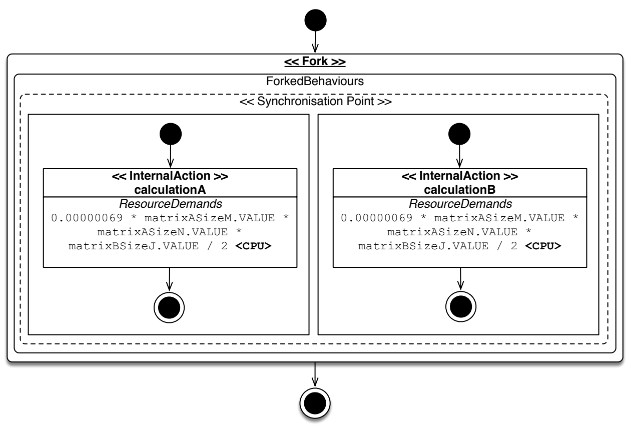
Example: Matrix Multiplication Listing 1 shows an exemplary implementation of a matrix multiplication. Three for loops are used (line 2-4.) to iterate over the cells of the input matrices A and B. The Results are stored in an additional matrix (line 5).

```
1 // omp parallel for threadNum(2)
2 for (int i = 0; i < matrixA.getWidth(); i++) {
3   for (int k = 0; k < matrixB.getHeight(); k++) {
4     for (int j = 0; j < matrixA.getHeight(); j++) {
5       result[i][j] += matrixA[i][k] * matrixB[k][j];
6 } } }
```

Listing 1: Sample implementation of a matrix multiplication in Java with OpenMP annotations

Implementing with OpenMP To parallelise the given function, we used omp4j[3], which is an OpenMP for Java precompiler. omp4j—like any other openMP framework—parallelises loops by merely annotating them (compare line 1 in Listing. 1). A programmer has to annotate the loop (or section) to parallelise. Parallel section work similar as parallel loops, but with the difference, that OpenMP does not execute each loop concurrent but each statement in the parallel section. In line 1 we further used one optional parameter `threadNum`, which defines the number of

[2]http://www.openmp.org
[3]http://omp4j.org



<< Fork >>
ForkedBehaviours
<< Synchronisation Point >>
<< InternalAction >> calculationA
ResourceDemands
0.00000069 * matrixASizeM.VALUE * matrixASizeN.VALUE * matrixBSizeJ.VALUE / 2 <CPU>
<< InternalAction >> calculationB
ResourceDemands
0.00000069 * matrixASizeM.VALUE * matrixASizeN.VALUE * matrixBSizeJ.VALUE / 2 <CPU>

Figure 1: SEFF for a two matrix multiplication with two threads

worker threads we want to use. OpenMP now splits the work among the specified amount of threads.

Modeling in Palladio While implementing the matrix multiplication is straightforward, modeling it is more challenging. To model the software behavior in Palladio, SPEs have to know the characteristics of the software. I.e., uncontended resource demands (e.g., the CPU time) for a specific task like a single multiplication—called action in Palladio—and the number of action invocation. The SPE can create a software model, with this information available.

In Figure 1 you can see a sample SEFF model of the matrix multiplication for two worker threads. The SEFF consists of two actions, which represents the workload for each worker thread. We abstracted the workload of each action by multiplying the number of loop iterations and the resource demand for a single multiplication (`0.00000069`). This has the advantage that the actual algorithm is abstracted and the performance of the analysis improves. We placed each action within a `ForkedBehaviour`, which indicates that both actions are executed concurrently.

Drawbacks The modeling of such a parallel behavior in Palladio currently suffers from two drawbacks, which we address in this paper:

I. Error-prone and time-consuming: The SPE has to model each worker thread individually, even though each thread is performing the same or similar actions. For a small number of threads (i.e., 2) this might be feasible, however for larger numbers (i.e., 128) it is not feasible, error-prone, and requires a lot of manual modeling.

II. Synchronization overhead missing: Spawning new threads and synchronizing them consumes resources as well. In our example model this overhead is not considered.

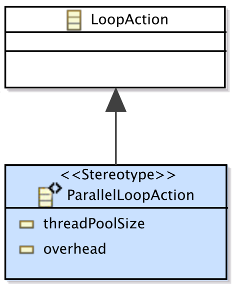


Figure 2: AT Profile for parallel loop extension

3 Parallel Loop Architectural Template

In this section we introduce a new **A**rchitectural **T**emplate that addresses the two mentioned drawbacks. Architectural Templates allow software architects to apply reusable patterns to their Palladio models [3]. Hence, there is no need to do a heavy weight extension of the PCM meta-model. Instead, the AT approach uses EMF Profiles to extend the meta-model without changing the core elements of the meta-model. The idea is that loop-actions can get a parallel loop AT annotation. This loop is then automatically transformed into a forked behavior.

AT Creation First step to use the AT method it to create a new AT. For this we follow the three steps as described in [3].

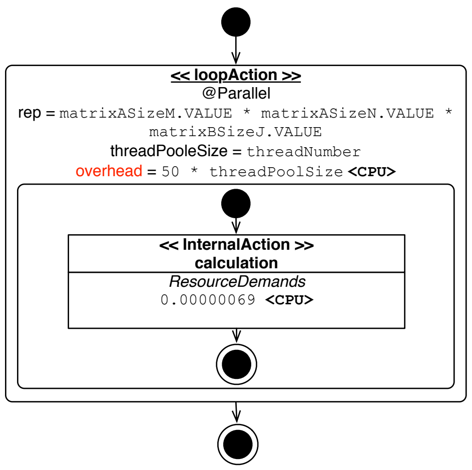
I. Create a Profile: First, we need to create a new profile. This is done in a similar way as defining UML2 Profiles. For this, we create a new stereotype class called `ParallelLoopAction` which extends the target class `LoopAction` from the PCM. In the addition, we model two properties: `threadPoolSize` and `overhead` (see Fig. 2). The thread pool size defines the number of worker threads available in the system (see Lst. 1, line 1). The overhead defines an additional CPU resource demand for each worker thread, i.e., for spawning or synchronizing a thread.

II. Define Completion: In the second step, we need to define a model-to-model transformation. This is done by a QVT-operational[4] definition. Before Palladio performs an analysis, the AT method searches for ATs in the model and executes the corresponding QVT-o rules for each AT found.

III. Register AT: In the last step, we have to add the newly created AT to the AT catalogue to make it available to the software architect via the Palladio tooling.

Description of the Parallel Loop Figure 3 represents the final result and includes all our extensions. In the figure, you can see a loop action, which has an

[4]https://www.omg.org/spec/OCL/2.4/



<< loopAction >>
@Parallel
rep = matrixASizeM.VALUE * matrixASizeN.VALUE * matrixBSizeJ.VALUE
threadPooleSize = threadNumber
overhead = 50 * threadPoolSize <CPU>
<< InternalAction >>
calculation
ResourceDemands
0.00000069 <CPU>

Figure 3: SEFF: Annotated parallel loop action

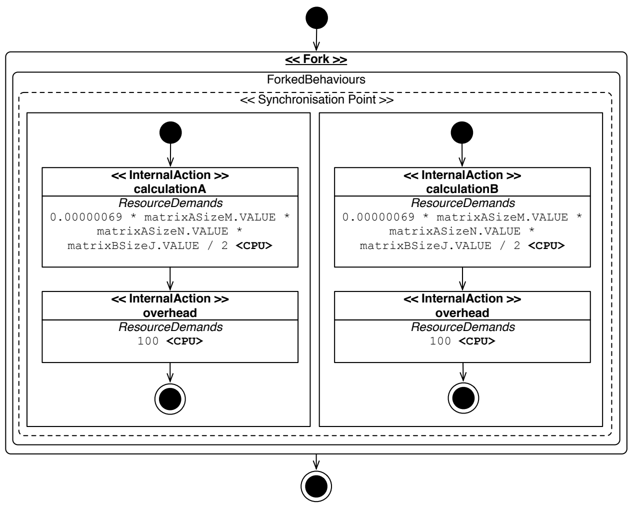


Figure 4: SEFF for a two threaded parallel loop with synchronization overhead

applied stereotype @Parallel. This indicates that the Parallel Loop AT is applied and models the open MP loop annotation. Further, the AT contains the two additional properties (thread pool size and overhead). This value can be given as constant, stochastic expressions, or function. In Figure 3 we used the number of threads for the `threadPoolSize` and as an exemplary overhead function: `50 * threadPoolSize`.

Mapping and Completion Afterwards, the transformation replaces the parallel loop via an in-place model-to-model transformation to a fork behavior. Figure 4 shows the result of the transformation of the parallel loop from Figure 3. The total resource demand is calculated from the number of loop iterations and split equally between the two worker threads. At the end of each thread, an additional action is added, which models the synchronization overhead.

4 Limitations and Future Work

The proposed AT provides a mechanism for SPEs to express OpenMP parallel constructs in a SEFF by leveraging profiles and stereotypes. Even though the approach allows flexible modeling of the synchronization overhead, estimating the overhead function is hard (even for experts). Therefore, our future goal is to provide a set of characteristic curves for different use cases, paradigms, and algorithms, enabling the SPE to pick one of the curves for a particular scenario.

Furthermore, our current approach focuses only on one aspect of parallel programs. Other relevant aspects like message passing, locks, shared variables, and resources are abstracted by the overhead function or not considered yet.

Another challenge we see in particular for parallel software running on multicore systems is that performance prediction models currently only support a single parameter, which is CPU speed. However, this might not be sufficient anymore for multicore systems and additional properties like memory hierarchies, cache sizes, and memory bandwidth are needed in addition. This means we have to adapt our performance prediction model to support the new properties. This goes hand in hand with further integration of new parallel paradigms into our modeling languages. E.g., OpenMP parallel loops is only one paradigm to think of, another one is the Actor paradigms—a message passing-based programming model.

5 Conclusion

In this paper, we introduced a new Architectural Template for parallel loops and sections known from OpenMP. This AT allows SPEs to model massive parallelism in their software models easily. In so doing, they save a lot of manual modeling effort due to automation. Further, we have included a concept to also take spawn and synchronization overhead into account. In the future, we plan to provide a repository with characteristic curves representing the overhead and speed-up curves for different use cases, paradigms, and algorithms.

References

[1] S. Becker, H. Koziolek, and R. Reussner. “The Palladio component model for model-driven performance prediction”. In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22.

[2] M. Frank and M. Hilbrich. “Performance Prediction for Multicore Environments—An Experiment Report”. In: *Proceedings of the Symposium on Software Performance 2016, 7-9 November 2016, Kiel, Germany.* 2016.

[3] S. Lehrig. “Efficiently Conducting Quality-of-Service Analyses by Templating Architectural Knowledge”. Accepted for publication. PhD thesis. University of Stuttgart, Germany, Nov. 2017.