Performance Prediction for Multicore Environments— An Experiment Report

Markus Frank
Technische Universität Chemnitz
09107 Chemnitz, Germany
markus.frank@informatik.tu-chemnitz.de

Marcus Hilbrich
s-lab – Software Quality Lab,
Universität Paderborn
33102 Paderborn, Germany

Abstract

Multicore systems are a permanent part of our daily life. Regardless whether we consider nowadays desktop PC's, notebooks, or smart phones: all devices are running on multicore CPUs. To use such hardware in an efficient way, we need parallel enabled software. But the development of such software is more complex and more error-prone than developing sequential software.

To handle the rising complexity, it is necessary to develop software in an engineering way. In such a process, software architects have to plan and analyze software designs on model level. Software architects can use approaches like Palladio to simulate and analyze early phase software designs. However, it is uncertain how Palladio can handle multicore systems.

In this paper we evaluate the current state of Palladio regarding multicore awareness based on an experiment. We implemented an easy to parallelize use case, modeled, and simulated it using Palladio.

We predicted the performance of an 16 core system with an accuracy of 79 %, but noticed a decreasing accuracy for a rising number of cores. Based on the experiment, we discuss the need to model attributes like memory, memory bandwidth, and caches, which are currently included.

1 Introduction

For more than a decade, multicore processors are widely used. Nowadays these kind of processors are common in all kinds of devices like smartphones, laptops or desktop PC's. To use the full performance of multicore processors, software developers have to build their software in a way they support even low level parallelism. But the development of such software is more complex and more error-prone. To still ensure high quality an engineer-like software development process is used, where software architects plan software systems with the help of software models in an early development phase. To enable software architects to model and analyze parallel software, tool support is needed. One available approach for this purpose is Palladio[1].

To run simulations in Palladio, software and hardware models must be enriched with additional information like resource requirements of a method call (i.e how much CPU time is required) or information about the target hardware (i.e. how fast the CPU is) [3].

Currently Palladio considers CPU, network, memory, and HDD as hardware resources. However, when it comes to multicore systems, it is assumed that there are more relevant factors like, memory size, memory bandwidth and caches [1].

To better understand the relevant factors for predicting multicore systems, we performed an experiment. Within our experiment, we aim to evaluate the capabilities of Palladio to model and to predict the performance of parallel software systems.

We were able predict the performance of an 16 core system with an accuracy of 79 %, even if we had to use some workarounds and faced significant manual modeling overhead. Further, we observed an decreasing accuracy for a rising number of cores. Which indicates that performance predictions for systems with more cores will be more off. In this paper we also present initial thoughts on how to increase the accuracy of model based performance prediction for multicore systems.

2 Experiment Setup

To evaluate the capabilities of Palladio to model and to predict the performance of parallel software systems, we raised two questions needed to be answered in our experiment. (Q_1) Is it possible to model multicore systems with Palladio? (Q_2) How precise are the predictions? To answer these questions, we choose to face the problem from two sides. On the one hand side, we implement a matrix multiplication as easy to parallelize code example and measure the execution time on a dedicated hardware. As hardware we use a server machine with 2 CPU's each containing 8 physical cores at 2,4 GHz each and 20 MB Cache per CPU. As operation system, we used Ubuntu Server 14.10. During the experiment, we disabled functions like hyper-threading or frequency scaling via the BIOS.

On the other hand, we model the same example with Palladio and perform a simulation. As metric we only focus on the execution time of the actual matrix multiplication. To evaluate the accuracy, we compare the measurements with the simulation result.

For a matrix multiplication there are different variants in which order the data in the memory is accessed. The performance of each variant can variate a lot. This is because the data is always read in cache lines from the memory, the manner how the matrix is stored in the memory and how it is accessed is important and has an impact on the performance. Hence, we determined the quickest solution and applied only this for the measurements and simulation. To archive meaningful results, we plan to execute the matrix multiplication several times and for 2, 4, 8, and 16 worker threads which corresponds to the number of CPU cores we use.

[1] http://www.palladio-simulator.com/

3 Conducting the Experiment

Implementation During the implementation phase, we first implemented the sequential version of the matrix multiplication. Listing 1 shows the implementation of the actual matrix multiplication (line 2-6). We implemented six different variants of the multiplication, according to the six different orders the values of a matrix can be multiplied with each other (aka. the order of the three for loops).

```
1 // omp parallel for schedule(static) 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: Fastest Matrix Multiplication Variant

During program execution, two matrixes are initialized and filled with random integer values. Afterwards, the two matrixes are multiplied several times, depending on the number of variants and the experiment repetition number. For each run and each variant, the execution time is measured. All attributes like matrix size, considered variants, and repetition number are parametrized and can be adjusted.

After the sequential implementation, we parallelized the code. For this we used the omp4j[2] framework. omp4j is an open-source implementation of OpenMP and is used as Java preprocessor. For our experiment, we used omp4j because it is user friendly and easy to use. Altogether, we ended up adding only one annotation to the existing code to instruct the omp4j preprocessor, that the for-loop has to be parallelized and to set the number of worker threads (see Lst. 1, line 1).

Regarding the execution time, we want to focus only on the actual matrix multiplication. So we measured as close as possible to the for-loops and ignored setup routines like filling the matrix with random numbers. The full source code, measurements, models and simulation results are public available via our gitlab project[3].

Measuring In a first run, we measured the execution time of the different variants and determinate the fastest variant (shown in Lst. 1). Focusing only on the fastest variant, we performed additional runs and variated the number of working threads, which basically influences the number of used CPU cores.

Table 1 gives the mean, minimum, and maximum execution time for each number of worker threads, as well as the standard deviation and the speed-up. The measured speed-up is linear but grows less than the number of cores Further, we measured a maximum mean speed-up of 11.66 and a total maximum speed-up of 13.14.

Modeling with Palladio After the implementation, we focus on modeling the application in Palladio. For that, we start again with the sequential Scenario.

To model the sequential scenario, we used one component named `MatrixMultiplicator`. It contains the SEFF for the actual matrix multiplication. Instead of modeling the three for-loops we decided to abstract the behavior and modeled a single internal action for

Worker Threads	Execution Time (s) Mean	Min	Max	STDEV	Speed-up
1	18.64	17.76	20.10	3.18	1.00
2	10.31	9.58	10.64	2.45	1.80
4	5.45	5.06	6.27	2.27	3.42
8	2.95	2.74	3.23	0.88	6.32
16	1.60	1.53	1.85	0.40	11.66

Table 1: Measurement summary

calculating the matrix. This was due to the fact, that Palladio could not handle the great amount of internal action calls, if using three for-loops. We determined the resource demand based on the data we gained form the measurements of the sequential matrix multiplication. Since the measurements did not deviate a lot, we used the mean execution time to determine the CPU resource demand. Further we used the Linux O(1) exact scheduler introduced by [2], which also considers additions environment parameters like time for context switches.

After the sequential model, we enhanced the model to fit the parallel scenario. This process involved a lot of manual work, since the parallel constructs in Palladio are very basic. The idea behind our approach is to model each worker thread as separated branch, where each branch gets the same amount of work. This is a valid assumption, because the openMP parallel loop construct is realized alike.

So instated of one internal action, we needed now 2 till 16 internal actions, which run in parallel with the same amount of work load. To achieve this, we added a passive resource to model the thread pool. Afterwards we extended the SEFF with a loop-action shown in figure 1. Within the loop action a fork-action is placed. Further, we modeled the waiting conditions. Therefore, we used the passive resource and the acquire and release action. We set the number of loop iterations to equal the size of the thread pool. Each iteration acquires an token from the passive thread pool resource first. Afterwards the fork is executed. Within the fork the internal action for the calculation is placed. The resource demand for each internal action, is the total resource demand for the multiplication divided by the number of available threads. Which is in our case the number of performed multiplications times the resource demand for a single multiplication, divided by the thread pool size. In the next step the thread is released again. After the first loop we placed a second loop. The second loop ensures, that the main thread waits, till all sub threads are finished. Additionally, we had to adjust the number of CPU replicas in the resource environment, to the number of available cores. In our case either 2, 4, 8 or 16.

Unfortunately, this approach has a couple of drawbacks. Fist, the exact schedulers cannot work with passive resources. So we had to use processor sharing instead. The second drawback is a bug in the current Palladio version, which will lead to an simulation runtime exception, if using release and acquire operations in fork actions.

To still get usable results we explicit modeled the loop action for our scenario, which meant to model 2 to 16 worker threads manually, by manual adding either 2, 4, 8 or 16 behavior compartments each with an internal action. This is a time intensive and error-prone process.

[2] http://omp4j.org/

[3] `https://gitlab.hrz.tu-chemnitz.de/marfra--tu-chemnitz.de/ssp_ramBW`

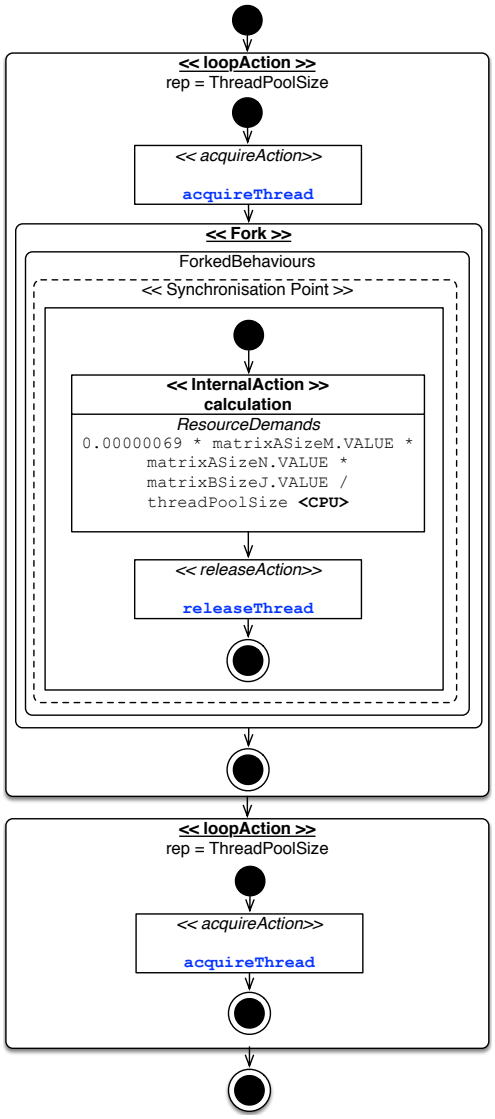

Figure 1: SEFF Definition of the Thread Model

Simulation We performed a simulation for the second modeled version for all number of worker threads (1 to 16). Since we used a resource demand based on the mean of the measured values for the sequential execution of the matrix multiplication, we expected almost static simulation results. Table 2 shows the results in comparison to the measured execution time for the different numbers of worker threads.

4 Interpretation

In Sec. 3 we showed, that the speed-up grows less than the number of cores. There are different reasons for not reaching an ideal speed-up. One, we assume to have the greatest impact, is synchronization. In most cases the matrixes are not read or wrote directly from the memory but from the cache. So every time the result matrix is updated, the cache entry becomes invalid and has to be expensively synchronized.

Evaluation of Q_1: During the modeling phase, we could show, that modeling multicore systems are basically possible. However, a lot of manual and error-prone modeling by hand was needed, since every thread had to be modeled individually. Concepts to directly add parallel constructs (e.g. OpenMP parallel loop constructs) in Palladio are desirable for the future.

Worker Threads	Mean Execution (in s)	Mean Simulation (in s)	Accuracy
1	18.64	18.63	0.99
2	10.31	9.41	0.91
4	5.45	4.76	0.87
8	2.95	2.43	0.82
16	1.60	1.26	0.79

Table 2: Simulation Results

Evaluation of Q_2: To evaluate the precision of the simulation results, we performed a number of simulations according to the number of worker threads. The best accuracy we achieve for the sequential scenario, which is only logical since we used the measurements, gained form the sequential run, as foundation for the resource demand. But noteworthy is the decreasing accuracy with a growing number of worker threads. Which indicates, that predictions for even more worker threads will be even worse. Which shows that further concepts like synchronization overhead, have to be considered in the model to increase the accuracy.

First measurements with matrixes sizes significantly larger than the cache size, further indicate, that the simulations for such matrixes are even more off. We assume reasons for that are limiting factors like caches, memory size, and memory bandwidth are not considered in the model yet.

5 Conclusion and Outlook

Within our experiment setup, we evaluated the capabilities of Palladio to simulate multicore systems. We showed that it is possible to model multicore systems within Palladio and we achieved an prediction accuracy of 79 % for a 16 core system. However, we showed that the modeling language should be enhanced with parallel constructs to reduce manual modeling overhead. Further, we observed a decreasing prediction accuracy for higher numbers of cores and discussed initial thoughts on possible reasons.

In the future, we will examine the reason for the inaccuracy further and also consider assumed limiting factors like memory bandwidth. For this, we plan to follow the systematic method on how to stepwise enrich software and hardware models with additional information needed for more accurate simulations proposed by [2].

References

[1] S. Becker, T. Dencker, and J. Happe. Model-driven Generation of Performance Prototypes. In *SPEC International Performance Evaluation Workshop*, pages 79–98. Springer, 2008.

[2] J. Happe. *Predicting Software Performance in Symmetric Multi-core and Multiprocessor Environments*. Dissertation, University of Oldenburg, Germany, August 2008.

[3] R. Reussner, S. Becker, J. Happe, H. Koziolek, K. Krogmann, and M. Kuperberg. The palladio component model. *Interner Bericht*, 21(S 5), 2007.