Reducing Search Space of Auto-Tuners Using Parallel Patterns

Christoph A. Schaefer
University of Karlsruhe (TH)
Institute for Program Structures and Data Organization (IPD)
Am Fasanengarten 5, 76131 Karlsruhe, Germany
cschaefer@ipd.uka.de

Abstract

Auto-tuning is indispensable to achieve best performance of parallel applications, as manual tuning is extremely labor intensive and error-prone. Search-based auto-tuners offer a systematic way to find performance optimums, and existing approaches provide promising results. However, they suffer from large search spaces.

In this paper we propose the idea to reduce the search space using parameterized parallel patterns. We introduce an approach to exploit context information from Master/Worker and Pipeline patterns before applying common search algorithms. The approach enables a more efficient search and is suitable for parallel applications in general.

In addition, we present an implementation concept and a corresponding prototype for pattern-based tuning.

The approach and the prototype have been successfully evaluated in two large case studies. Due to the significantly reduced search space a common hill climbing algorithm and a random sampling strategy require on average 54% less tuning iterations, while even achieving a better accuracy in most cases.

1. Introduction

Tuning of parallel applications is important, as finding the best configuration of tuning parameters is far from easy and often non-intuitive [12, 18]. Tuning parameters include all performance-relevant variables of a program, such as the number of worker threads, the choice of load balancing strategies, the number of threads per stage in a pipeline, or size of data partitions.

1.1. Need for Auto-Tuning

The concept of automatic performance tuning (auto-tuning) has been studied for several years, especially in the field of numerical programs [5, 19]. The growing diversity of parallel programs and the large variety of application areas for parallelism requires more general auto-tuning methods.

For this purpose, search-based auto-tuners represent a promising systematic approach [2, 13, 16, 17]. A search-based tuner is a library or a stand-alone application that dynamically executes a parameterized application several times (called tuning iterations) and monitors performance. After each tuning iteration, it calculates a new parameter configuration based on the performance results of the previous iteration. Doing so, it tries to find a parameter configuration that yields the best performance on a given target platform.

Experiments with real-world parallel applications have shown that using appropriate tuning techniques, a significant performance gain can be achieved on top of the performance improvement attained by the parallelization itself (cf. Section 4, [12, 14]).

However, the size of the search space that is defined by the cross-product of all parameter domains in the program is crucial for the efficiency of the search algorithm. As the search space grows exponentially, even a smart search algorithm may need a long time to find the best – or at least a sufficiently good – parameter configuration. For example, consider the initial search space of our first case study in Section 4, that consists of nearly 240,000 parameter configurations. If a sophisticated search algorithm tests only 1% of the configurations, it will still perform 2,400 tuning iterations.

Some research approaches therefore suggest the use of models instead of search-based techniques to predict the performance depending on certain non-functional parameters [10, 11, 3]. However, model-based approaches are often limited to certain algorithm types, application domains and hardware platforms. In addition, studies have shown that search-based tuning is more effective than model-based optimization [4].

1.2. Search Space Reduction

Search-based auto-tuners are usually not aware of purpose and impact of the tuning parameters. Thus, useless parameter configurations might be tested.

We therefore propose to reduce the search-space of auto-tuners using parameterized parallel patterns. A pattern sets all its parameters in a certain context we can exploit. If this context information can be made available to an auto-tuner, it can focus the search on the most promising parts of the search space. The idea is to specify the search space more intelligent and thus reduce it *before* applying common search algorithm.

We investigate the Master/Worker and the Pipeline pattern – two commonly used patterns – regarding their performance-relevant parameters, analyze the coherence of the parameters and identify options to sample the pattern-related search space more efficiently. In addition, we present a suitable implementation approach introducing the concept of *tuning wrappers*.

In two large case studies – one of them deals with a commercial application – we show the efficiency of our prototype: The significantly pruned search space results in approx. 54% less tuning iterations required by two search algorithms, namely hill climbing and random sampling. Except for one case, both strategies achieved a better accuracy compared to conventional tuning without context information.

2. Exploiting Context Information of Parallel Patterns

A tuning parameter appears to an auto-tuner as a set of values to choose from. Typically, the auto-tuner does not know anything about purpose and dependencies of a particular parameter. Consequently, it has to tune all the program's parameters in concert with each other. This may result in testing useless parameter configurations.

But actually each tuning parameter belongs to a parallel section that implements a certain type of parallelization strategy, that in turn can be expressed as a pattern. Parallel patterns describe recurring parallel computation or decomposition problems of similar structure and synchronization behavior and provide a suitable solution [8, 9].

Patterns represent enclosed parallel sections that can be considered as independent from each other if they are not nested or sharing resources. In earlier work [14] we have shown that independent parallel sections can be tuned separately, which cuts the initial search space into smaller pieces.

In particular, a parameterized pattern offers context for each of its tuning parameters. If this context information is expressible in some way, an auto-tuner can focus the search on crucial parts of the search space evolved by a particular pattern.

In the following sections, we introduce two commonly used patterns. We discuss how their tuning parameters cohere as well as how we can exploit the coherence and other characteristics of the pattern to enable more intelligent search-based tuning.

In addition, we use the concept of history values to take advantage of the experience gained in earlier tuning runs. History values of a certain parameter depend on the hardware platform and are stored in the context of the pattern the parameter belongs to.

For all considerations we assume a shared memory multicore platform.

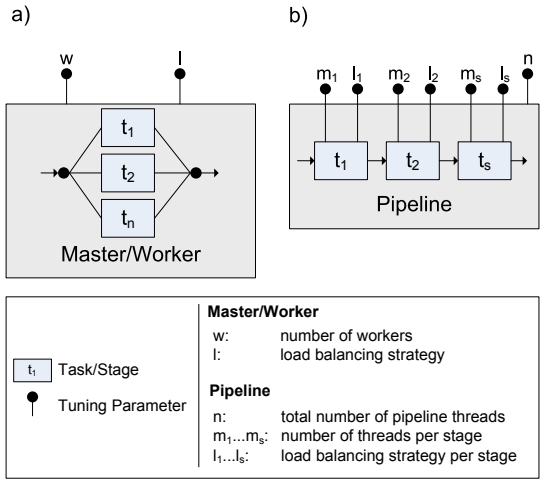

Figure 1. Schematic representation of parameterized parallel patterns: a) Master/Worker, b) Pipeline with data parallel stages.

2.1. Master/Worker Pattern

The Master/Worker (M/W) pattern is a frequently used strategy to implement task parallelism. A task is a program element that can execute independently. The master creates a number of worker threads, assigns one or more tasks to each of them and starts the workers. The master waits or does other jobs until all workers have finished their tasks. After collecting the results, the master stops all workers and proceeds with the rest of the program.

The performance of the M/W pattern mainly depends on two tuning parameters: the number of workers (w) and the load balancing strategy (l). Although the M/W pattern offers additional tuning options, such as the data partition size for particular workers, in literature [3] as well as according to our experience, w and l are considered to be the most influencing parameters regarding performance.

Figure 1 a) sketches a parameterized M/W pattern.

The optimal number of workers for a particular hardware platform coheres with the available number of processor cores (p) as well as with the total number of tasks (t). However, we do not know the exact correlation. To reduce the search space for w, we suggest to use the history parameters α and β to restrict the search to a range around p. That is, $\lfloor \alpha \cdot p \rfloor \leq w \leq \lfloor \beta \cdot p \rfloor$, where $0 < \alpha \leq 1$, $\beta > 1$. The values for α and β are history values depending on a particular value of p. For example, if the best values for w always were between 6 and 10 on an 8-core machine ($p = 8$), the stored values for α and β are 0.75 and 1.25 respectively. To consider the correlation with t, we can use a fix range around t and add the corresponding values to the value range of w. Based on our experience we suggest $\lfloor 0.9t \rfloor \leq w \leq \lfloor 1.1t \rfloor$. Obviously, we can include t in our analysis only, if t is a constant value. That is, the number of tasks in the program must be fixed for the particular M/W section.

The load balancing strategy defines how the tasks are assigned to the workers. In the M/W pattern, the load balancing strategy does not depend on the number of workers, but on the running times of the tasks. In literature, different strategies are discussed [3]. We focus on static and dynamic load balancing.

With static load balancing the master equally distributes the tasks to the workers. That is, each worker gets $\lfloor t/w \rfloor$ tasks. While the dispatching overhead for the master is low, the load balancing may be suboptimal if the tasks have unequal running times. Therefore, static load balancing usually yields good performance with tasks requiring approximately the same time to complete.

Dynamic load balancing moves more control to the workers. Initially, each worker gets one task. Whenever a worker has finished a task, it requests the next available task from the master. This means a lot more overhead for the master. However, this strategy performs well with tasks having unequal running times.

Without context, we would have to test both values for l (static, dynamic) with all values of w. Using the knowledge about M/W, we can test each strategy just once with a constant value for w (usually $w = p$) and finally use the strategy that yields the better performance. Doing so, we need only 2 instead of $l \cdot w$ samples. If the number of tasks (t) is smaller than the number of workers (w), we can even omit testing, as each worker gets a maximum of one task only. In this case, the load balancing strategy doesn't matter.

After applying the aforementioned considerations, the remaining search space of the M/W pattern consists only of the reduced value range of w. A common search algorithm, such as hill climbing, can now find the optimal value in a short time.

2.2. Pipeline Pattern

The Pipeline pattern schedules the tasks according to a pipelining strategy. That is, the tasks are executed in a fixed order, i.e. in stages, while two consecutive stages are connected by a FIFO-based data type; the output of each stage is the input of the next. Each stage runs at least in one separate thread.

To yield good performance, all stages must be equally balanced regarding their running time. We can influence the running time of a stage if we assign more than one thread to the stage. For this purpose, the stage's task must in itself provide parallelism, such as task or data parallelism.

Above all, we have to consider the total number of threads (n) the pipeline can use concurrently.

Figure 1 b) sketches a parameterized pipeline with data parallel stages. Typically, an auto-tuner would search for an optimal configuration within the entire parameter space of the pipeline. If the number of stages is s, this would result in $2s + 1$ parameters.

However, a pipeline often lacks performance due to a stage that is significantly slower than the others. Thus, a suitable heuristic would be to adjust the running time of the fastest and the slowest data parallel stage only.

Therefore, we first have to find the best value for the total number of pipeline threads (n) as a limit. As described for the M/W pattern, we can use history parameters to limit the search for n to $\lfloor \alpha \cdot p \rfloor \leq n \leq \lfloor \beta \cdot p \rfloor$, where $0 < \alpha \leq 1$, $\beta > 1$, and p the number of cores.

Now we can start the search to find the optimal values for the number of threads in the slowest ($m_{slowest}$) and the fastest stage ($m_{fastest}$) by sampling the resulting search space $dom(m_{slowest}) \times dom(m_{fastest})$. The search is finished if both stages have approximately the same running time. We assign the rest of the available threads (that is $n - m_{slowest} - m_{fastest}$) to the other data parallel stages – either equally or inversely proportional to their particular running times. After setting the number of threads for each stage, we can fine-tune the slowest stage by adjusting the load balancing strategy as described in the previous section.

If necessary, we can iterate this process to adjust the remaining stages accordingly.

Of course, our heuristic can be improved and extended to be suitable for other runtime characteristics of the pipeline stages. However, the approach prunes the search space of the pipeline to smaller stage-related parts that are easier to handle for search algorithms.

2.3. Nested Parallelism

Our approach so far allows only limited nested parallelism. The heuristic for the pipeline pattern in the previous section considers nesting, as each stage can contain its own

parallel section. However, the heuristic is not applicable for all kinds of nested parallelism.

A more general approach would be to separately determine the runtime characteristics of each child component of the nested structure. To do so, we have to treat each component according to its underlying pattern as described in the previous sections and apply a search-based tuning process. For each component we get samples that map a runtime value to a parameter configuration. Using the samples from all child components, an optimizer can find the best combination of all child components' configurations depending on the tuning parameters of the parent component.

2.4. Findings

The heuristics we discussed above show that parallel patterns provide context information we can exploit to improve search-based tuning. The auto-tuner does not have to tune a "black box" anymore by adjusting a number of anonymous parameters. Using the inherent context as well as history parameters, we can reduce the initial search space. The remaining space is significantly smaller and thus more feasible for a common search algorithm, as shown in our case studies in Section 4.

3. Implementation Concept

In this section we present a possible approach to build a context-aware auto-tuner. The approach is based on our auto-tuning framework *Atune* [14].

3.1. Atune

Atune is a search-based auto-tuning framework that adjusts parameter values between consecutive executions of a parallel program. We extended the auto-tuning principles known from numerics and algorithmic engineering to work with a wide range of parallel programs.

We assume that we have an existing parallel program instrumented with our tuning instrumentation language *Atune-IL* [14]. The pragma-based language is used to specify tuning information such as parameters and their value ranges, blocks, or measuring points within the code of the parallel application.

Atune's auto-tuning cycle contains the following steps:

(1) The *Atune-IL Pre-Processor* parses the instrumented program and builds up a data structure containing the tuning information.

(2) The tuning information is passed to *Atune-OPT*, the actual optimizer. Using exchangeable search algorithms, Atune-OPT computes a tuple of values that represents a valid configuration of parameters. As the optimization process itself is not the focus this paper, we refer to existing approaches for details [17, 13].

(3) The computed parameter values are inserted into the code of the program. The output of this stage is an executable variant of the original program.

(4) Next, the *Atune Backend* starts the new program variant and monitors it. Data from all measuring points is recorded, aggregated, and stored.

(5) In the last step the recorded monitoring results are transformed to a format usable by Atune-OPT. The auto-tuning cycle (steps 2-5) is repeated until some predefined condition is met, depending on the employed search algorithm.

3.2. Pattern-based Parallel Sections

Our approach requires implementations of parameterized parallel patterns. According to the schematic representation in Figure 1, we have developed a library containing the components for the M/W and the Pipeline pattern. For both pattern types, a well-defined interface exists to enable application developers to implement their own pattern components.

We decided to specify the tasks to be processed using pointers to methods that implement the tasks. The list of method pointers is passed to the pattern component.

The pattern components enable us to build consistent parallel sections in a systematic and standardized way. As we know the behavior of the pattern and its corresponding tuning parameters, we can provide this information to the auto-tuner. In the next section we show how this could be done.

3.3. Tuning Wrappers

The pattern components described in the previous section provide tuning parameters. Nevertheless, the components are not auto-tunable yet. Therefore, we have to mark the tuning parameters and indicate the existence and type of the parallel pattern. For that reason, we use our instrumentation language Atune-IL. As described in Section 3.1, Atune-IL provides instrumentation constructs to specify tuning information within the program's code.

To automate and standardize the instrumentation process, we introduce the concept of *tuning wrappers*. A tuning wrapper is a static class that encapsulates the call to an instance of a pattern component. For each pattern type, a corresponding type of tuning wrapper exists. A tuning wrapper contains the declarations of the tuning parameters, the call to the particular instance of the component, and all necessary Atune-IL statements. In addition, a tuning wrapper

indicates a separate block in terms of the Atune-IL specification [14]. That is, if the encapsulated parallel section is independent from others, it can be tuned separately.

Within the program, the tuning wrapper is always called instead of the encapsulated pattern component.

Tuning wrappers are designed to be created automatically by a tuning wrapper generator. The wrapper class is generated in a separate code file. The call to the wrapper is placed within the program's code at a point that is annotated by the application programmer.

To use tuning wrappers together with pattern components, an application programmer has to perform the following steps for each pattern-based parallel section within the program:

(1) Create an instance of a particular pattern component and pass the task method pointers.

(2) Add an annotation for the tuning wrapper generator specifying the new wrapper's name and type as well as the name of the pattern component instance created above.

(3) Start the tuning wrapper generator.

(4) Modify the Atune-IL statements in the generated wrapper class if necessary.

Listing 1 shows the code example for creating an instance of a M/W pattern component and the annotation to create a corresponding tuning wrapper (according to steps 1 and 2).

Listing 1. Code example to create a Master/-Worker instance and a corresponding tuning wrapper

```
...
IMasterWorker myMw =
  new MyMasterWorker(taskMethodPointerList);

#pragma tuningwrapper
  CREATE MyMwTuningWrapper
  TYPE MasterWorker
  INSTANCE myMw
...
```

The tuning wrapper generator acts as a pre-processor that parses the program's code for all `pragma tuningwrapper` annotations. For each annotation found in the program it generates a tuning wrapper class and a call statement replacing the annotation.

For illustration, Figure 2 depicts the generated code of a M/W tuning wrapper class. The wrapper method is surrounded by the Atune-IL statements `STARTBLOCK` and `ENDBLOCK` to specify a block for tuning parameters. The `STARTBLOCK` keyword `MasterWorker` indicates that this is a wrapper for a M/W pattern component. The wrapper method declares the two tuning parameters of the M/W pattern component and initializes them with default values. Each parameter declaration follows the Atune-IL statement `SETVAR` specifying appropriate tuning information. These statements will be replaced by a variable assignment in step 3 of the auto-tuning cycle (cf. Section 3.1). The `SETVAR` statement with the context `GET_numTasks` represents a constant program parameter that is read-only for the autotuner. Here, the statement contains the number of tasks. The value of the program parameter must be correctly specified by the application programmer. Finally, the wrapper passes the tuning parameter values and the tasks pointer to the M/W instance and executes it (see box in Figure 2).

The generated code for a Pipeline tuning wrapper looks similar to the M/W example.

```
static class MyMwTuningWrapper :
          IMasterWorkerTuningWrapper
{
   #pragma atune STARTBLOCK MasterWorker

   public void Execute(
          MasterWorker mw,
          List<TasksPointer> taskPointers)
   {
      int numThreads = 2;
      #pragma atune SETVAR numThreads TYPE int
            VALUES 2-32 CONTEXT numThreads

      LBStrategy lbs = LBStrategy.Dynamic;
      #pragma atune SETVAR lbs TYPE generic
            VALUES „LBStrategy.Dynamic;
                    LBStrategy.Static"
            CONTEXT lb

      #pragma atune SETVAR numTasks
          VALUE 8 TYPE int CONTEXT GET_numTasks

      #pragma atune GAUGE mwExecTime1
      mw.Start(numThreads, lbs, taskPointers);
      #pragma atune GAUGE mwExecTime1
   }
   #pragma atune ENDBLOCK
}
```

Figure 2. Generated code of a Master/Worker Tuning Wrapper.

The generated call to the tuning wrapper in Figure 2 is shown in Listing 2. The tuning wrapper generator has replaced the annotation by the appropriate calling statement.

Listing 2. Code after the tuning wrapper generator was started

```
...
IMasterWorker myMw =
  new MyMasterWorker(taskMethodPointerList);

MyMwTuningWrapper.Execute(myMw);
...
```

4. Experimental Results

As a proof of concept we have implemented a prototype based on our Atune framework and successfully evaluated it in two case studies.

4.1. Prototype

The prototype extends Atune-OPT (cf. Section 3.1) to be able to exploit the context information of M/W patterns.

We had to modify the part of Atune-OPT that prepares and interprets the information retrieved from the Atune-IL parser. Atune-OPT is now able to recognize if the program contains `MasterWorker` blocks inside tuning wrappers. If such a block is detected, Atune-OPT handles the block's parameters according to the M/W context information and tunes them respectively. Each M/W block is considered to be independent and therefore tuned separately.

As the actual optimization process is still search-based, Atune-OPT would include parameters of unknown parallel sections into the tuning session as well. Obviously, these parameters have to be treated conventionally, since no additional information is available.

Up to now, the prototype covers context-aware tuning of M/W sections. However, we are currently working on the implementation of tuning capabilities targeting the Pipeline pattern. The preliminary results are promising.

4.2. Case Studies

We performed two case studies to show the efficiency of our approach. The subjects of the case studies are two large applications we briefly sketch below.

The program of the first case study is Agilent's MetaboliteID (MID) [1], a commercial application for biological data analysis. Basically, MID compares mass spectrograms to identify metabolites.

The second case study deals with GrGen, that is currently the fastest graph rewriting system [6]. As a benchmark, we simulate the biological gene expression process on the E.coli DNA [15]. The model of the DNA results in an input graph representation consisting of more than 9 million graph elements.

We have parallelized both applications by integrating our parameterized M/W pattern component (cf. Section 3.2) and made them auto-tunable using corresponding tuning wrappers. Each M/W section provides the tuning parameters w (number of threads) and l (load balancing strategy) introduced in Section 2.1. Although both applications provide additional tuning parameters, we focus on the parameters in the M/W sections and set the remaining parameters to their default values. The first part of Table 1 lists relevant application characteristics.

To evaluate our approach, we compared the efficiency of search strategies with and without context-aware search space reduction. The search strategies were 1) random sampling based on the Mersenne-Twister algorithm and 2) a common implementation of the hill climbing algorithm. We intentionally chose two basic algorithms to show that using our approach, even ordinary search strategies achieve acceptable results.

We determined the efficiency by counting the number of tuning iterations a search strategy needs to finish. While the hill climbing algorithm converges after a certain number of iterations, random sampling needs an iteration limit. To be comparable, we set this limit to the number of iterations required by the hill climbing algorithm. In addition, we measured the mean error rate, that indicates how close a search strategy converges towards the real optimum. We repeated each tuning run 20 times to get statistically significant results.

Table 1 summarizes the results for conventional and context-aware tuning per case study. All tests were performed on an 8-core machine (2x Intel Xeon E5320 Quad-Core, 1.86 GHz/Core, 8 GB RAM, Microsoft Windows 2003 R2). Therefore, our database for history values contained the results of earlier M/W tuning runs of different applications performed on an 8-core machine. Based on the M/W tuning history, Atune-OPT sets $\alpha = 0.9$ and $\beta = 1.8$ to limit the value range of w (cf. Section 2.1).

Results with MID. In MID we exploited parallelism using three independent M/W sections providing a total of six tuning parameters (w_i and t_i, $i = \{1, 2, 3\}$). We set all $w_i = [2, \ldots, 32]$ (according to a maximal number of 30 tasks) and all $l_i = ['static', 'dynamic']$, resulting in a search space size of 238,328 parameter configurations. Using context-aware tuning, the search space was divided into the three considerably smaller sub-spaces of each independent M/W section, that in turn were reduced using the heuristics and history values for the M/W pattern. Finally, the search space contained only 48 parameter configurations. As are result, hill climbing required 56% less tuning iterations and even achieved a lower error rate. Random sampling produces similar results regarding the error rate. The speed-up obtained by the best configuration was 3.1; the worst speed-up was 1.6. That is, the performance gain achieved solely by tuning was 193%.

Results with GrGen. GrGen offered two sections that were parallelizable using our M/W pattern component, covering a total of four parameters (w_i and t_i, $i = \{1, 2\}$). Again, we set all $l_i = ['static', 'dynamic']$, but $w_1 = [2, \ldots, 40]$ and $w_2 = [2, \ldots, 32]$ due to different numbers of tasks. The resulting search space had a size of 2,914 parameter configurations. After applying our context-aware approach, the search space was pruned to 35. The number of tuning iter-

ations required by hill climbing was reduced by 53% with a lower error rate. Using random sampling the mean error rate was 2% higher than without context-aware tuning, which is still acceptable. The tuning performance gain was 427%.

Table 1. Application Characteristics and Experimental Results

	MID	$GrGen$
Application Characteristics		
Avg. Running Time (ms)	85,000	45,000
Approx. Size (LOC)	150,000	100,000
# Independent M/W Sections	3	2
Search Space Reduction		
Size (conventional)	238,328	2,914
Size (context-aware)	48	35
Reduction	$> 99\%$	98%
Efficiency using Hill Climbing		
Avg. # Iterations (conventional)	30	17
Mean Error Rate (conventional)	8%	10%
Avg. # Iterations (context-aware)	13	8
Mean Error Rate (context-aware)	7%	9%
Efficiency using Random Sampling		
Avg. # Iterations (conventional)	30	17
Mean Error Rate (conventional)	15%	19%
Avg. # Iterations (context-aware)	13	8
Mean Error Rate (context-aware)	13%	21%
Performance Results		
Best Obtained Speed-up	3.1	7.7
Worst Obtained Speed-up	1.6	1.8
Avg. Tuning Performance Gain	193%	427%

The results show that context-aware tuning is worth its effort. In most cases, both search algorithms we used for evaluation achieved better results in a significantly shorter time, as less tuning iterations were required. In addition, the attained tuning performance gain indicates that auto-tuning is necessary to get the best performance of an application.

5. Related Work

Auto-tuning has been previously investigated mainly in the area of numerical software and high-performance computing. Therefore, many approaches, such as ATLAS [19] or FFTW [5], focus on tuning either certain types of algorithms or programs belonging to a particular application domain.

Newer research projects target a broader range of parallel programs. We discuss two of the most relevant approaches.

Active Harmony [17] is a framework for dynamic adaptation of applications regarding network and resource capacities. The application must use the system's tuning API. Active Harmony manages the values of the different tunable parameters and changes them for better performance using a search-based strategy. The project focuses on the selection of the most appropriate algorithm for a given problem. The algorithms have to be implemented in separate exchangeable libraries. The approach allows to tune more general types of applications.

The Monitoring, Analysis and Tuning Environment (MATE) [10] focuses distributed parallel applications. MATE uses a model-based approach to optimize scientific Master/Worker applications at runtime. MATE relies on performance models providing conditions to describe the application behavior and to allow the system to find the optimal values. While MATE's performance prediction for distributed Master/Worker applications is precise, tuning other types of parallel applications requires definitions of new appropriate performance models. MATE does not target multicore systems.

6. Conclusion and Future Directions

In this paper, we propose the idea to reduce the search space of auto-tuners using context information of parameterized parallel patterns. Our investigation of the M/W and Pipeline pattern shows that using tuning parameters in a pattern context, the search space can be specified more precisely. In addition, pattern- and platform-dependent history information of parameters from earlier tuning session can help reduce the search space significantly as well.

As a proof of concept we present a prototype implementation of a context-aware auto-tuner based on our auto-tuning framework Atune. Beside the conventional search-based optimization it is capable to exploit context information from M/W sections. In two case studies the prototype offers promising results.

It turned out that context-aware tuning based on parallel patterns is a feasible approach. However, there is still a lot of research work to do. We have identified several opportunities for future research.

- Beside the patterns analyzed in this paper, there are more parallel patterns to be studied, such as the Wavefront or the Devide&Conquer pattern. We have to investigate how the knowledge about these patterns is usable for tuning. Existing approaches dealing with

performance diagnosis of patterns, such as [7], represent good starting points.

- As real-world applications usually contain different types of parallel sections as well as nested pattern structures, reasonable combinations of patterns have to be explored to learn how their tuning parameters interact. In particular, our experience shows that patterns exploiting data parallelism are often nested within structural patterns, such as the Pipeline or Devide&Conquer pattern. Therefore, we have to understand the performance-related coherence of such pattern structures to develop suitable heuristics for efficient tuning.

- The integration of pattern-based performance models might be a promising approach to further reduce the number of tuning iterations. Performance modeling of parallel patterns is already discussed in literature [3, 11, 10]. The proposed models usually depend on particular hardware platforms, programming paradigms, or application domains. However, modern parallel applications often comprise several types of parallel patterns. Further on, tuning techniques should not be limited to certain hardware platforms and application domains. Therefore, a possible option is to use pattern-based models within our approach. The proposed pattern heuristics and the search-based tuning can be used to determine initial parameters and performance samples for the models. Feeding the results of the search-based approach to appropriate models, the optimal configuration of the program might be calculated more precisely in less time. This requires the adaption of existing models as well as creating new ones. In addition, it might be useful to have more than one model available for a particular pattern, as the auto-tuner can choose the most suitable model regarding the underlying platform or available external parameters.

- To improve handling, a smooth integration of our approach into existing search-based auto-tuners must be provided. Especially, a flexible specification of pattern heuristics and information about how to tune patterns is necessary.

Acknowledgements. The author would like to thank the Agilent Technologies Foundation for the financial support.

References

[1] Agilent Technologies: MassHunter MetaboliteID Software. (2008)

[2] K. Asanovic et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report, University of California, Berkeley, 2006.

[3] E. Cesar et al. Modeling Master/Worker Applications for Automatic Performance Tuning. *Journal of Parallel Computing*, volume 32, pages 568–589, 2006.

[4] A. Epshteyn et al. Analytic Models and Empirical Search: A Hybrid Approach to Code Optimization. *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, volume 4339/2006, pages 259–273, 2006.

[5] M. Frigo et al. FFTW: An Adaptive Software Architecture for the FFT. *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384, 1998.

[6] R. Geiß et al. GrGen.NET. `www.info.uni-karlsruhe.de/software/grgen/`. University of Karlsruhe, IPD Prof. Goos, 2008.

[7] L. Li et al. Knowledge Engineering for Automatic Parallel Performance Diagnosis. *Concurrency and Computation: Practice and Experience*, volume 19, pages 1497–1515, 2007.

[8] B. L. Massingill et al. A Pattern Language for Parallel Application Programs. *Proceedings of the 6th International Euro-Par Conference on Parallel Processing*, volume 1900/2000, pages 678–681, 2000.

[9] T. G. Mattson et al. Patterns for Parallel Programming. Addison-Wesley, 2004.

[10] A. Morajko et al. Design and Implementation of a Dynamic Tuning Environment. *Parallel and Distributed Computing*, volume 67, pages 474–490, 2007.

[11] Y. L. Nelson et al. Model-guided Performance Tuning of Parameter Values: A Case Study with Molecular Dynamics Visualization. *Proceedings of the IPDPS*, pages 1–8, 2008.

[12] V. Pankratius et al. Software Engineering For Multicore Systems: An Experience Report. *Proceedings of the 1st IWMSE*, pages 53–60, 2008.

[13] A. Qasem et al. Automatic Tuning of Whole Applications using Direct Search and a Performance-based Transformation System. *The Journal of Supercomputing*, volume 36, pages 183–196, 2006.

[14] C. A. Schaefer et al. Atune-IL: An Instrumentation Language for Auto-Tuning Parallel Applications. Technical Report, University of Karlsruhe (TH), 2009.

[15] Schimmel, J. et al.: Gene Expression with General Purpose Graph Rewriting Systems. *Proceedings of the 8th GT-VMT Workshop* (2009)

[16] V. Tabatabaee et al. Parallel Parameter Tuning for Applications with Performance Variability. *Proceedings of the Supercomputing Conference*, 2005.

[17] C. Tapus et al. Active Harmony: Towards Automated Performance Tuning. *Proceedings of the Supercomputing Conference*, 2002.

[18] O. Werner-Kytl et al. Self-Tuning Parallelism. *Proceedings of the 8th International Conference on High-Performance Computing and Networking*, pages 300–312, 2000.

[19] R. C. Whaley et al. Automated Empirical Optimizations of Software and the ATLAS Project. *Journal of Parallel Computing*, volume 27, pages 3–35, 2001.