Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2010) 1–9

Procedia Computer Science

International Conference on Computational Science, ICCS 2010

Dynamic Tuning of Algorithmic Parameters of Parallel Scientific Codes

Pilsung Kang[a,*], Naresh K. C. Selvarasu[b], Naren Ramakrishnan[a], Calvin J. Ribbens[a], Danesh K. Tafti[b], Srinidhi Varadarajan[a]

[a]*Department of Computer Science, Virginia Tech, Blacksburg, VA 24061*
[b]*Department of Mechanical Engineering, Virginia Tech, Blacksburg, VA 24061*

Abstract

We present a dynamic method for tuning algorithmic parameters of parallel scientific programs. By treating tuning as a separate concern in the software development process, our method supports personalized development of optimization schemes for existing programs that are not easily supported by conventional tuning techniques. We use a compositional framework to transparently combine tuning code with the original program without direct modification of an existing code base. In this way, the inserted tuning module can dynamically search for optimal values of algorithmic parameters, accounting for runtime factors such as input problem size and parallel characteristics of a given execution platform, as well as the architectural or runtime properties of a single machine of the platform. Applying our method to a parallel CFD (computational fluid dynamics) simulation, we demonstrate how a set of performance-critical parameters can be dynamically tuned, achieving up to 26% performance improvements over average cases.

Keywords: dynamic tuning, parallel simulation

1. Introduction

Algorithmic parameters can have a critical influence on the performance of a scientific application. A typical example involves domain decomposition methods, where a large problem domain is partitioned into small subdomains or blocks, with a dominant step in the algorithm corresponding to independent solves on each of the subproblems, which are then combined in some fashion to drive the global solution forward. Since a carefully chosen subdomain size (e.g., chosen to match the execution platform's memory hierarchy in some way) can lead to a significant speedup, tuning of the decomposition method is a common optimization strategy in this setting. Manual tuning of algorithm parameters not only is time-consuming in scientific computing, where several days or weeks of simulation is not unusual, but also requires a substantial level of understanding of both the target algorithm and the underlying hardware. This is particularly difficult with today's fast evolving hardware architectures such as multi- or many-core CPUs with complex memory hierarchies. Therefore, tuning support to automatically optimize scientific codes over parameter search spaces is becoming more relevant [1, 2]. For instance, vector and matrix operations – basic computational

*Corresponding author
Email address: `kangp@cs.vt.edu` (Pilsung Kang)

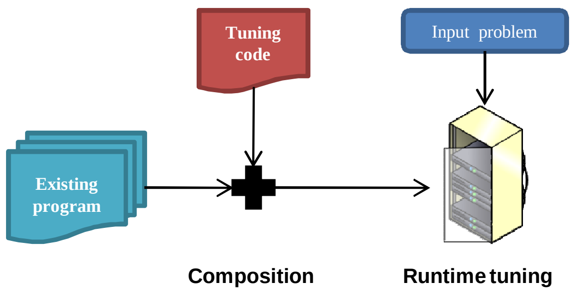


Figure 1: Composition of Dynamically Tuned Application with Existing Code

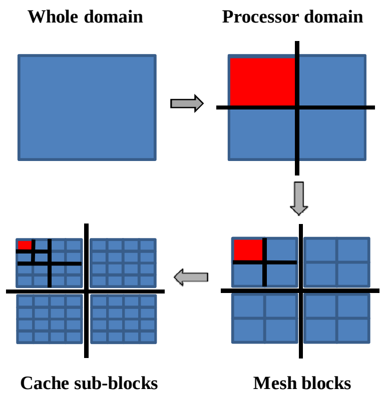


Figure 2: Hierarchy of Data Decomposition in GenIDLEST

kernels in numerical computing – are a prime target for automatic performance tuning (or 'auto-tuning'). These methods generate highly optimized codes by parameterizing performance-critical function and benchmarking the target system, thereby automatically determining the best possible configuration [3, 5].

Not all applications lend themselves to traditional approaches to auto-tuning, however. Scientific programs are often written using domain-specific data structures and algorithms, where patterns of computation cannot be mapped to well-studied basic linear algebra operations, thereby preventing direct application of automatically optimized software libraries. In particular, tuning support for algorithmic parameters in existing scientific codes is lacking, where even a modest effort in parameter tuning can provide significant speedups. Another limitation with conventional auto-tuning approaches is that they mainly consider only static hardware characteristics of a single machine and do not include runtime factors that develop when an application executes with an actual input problem on a given computing platform. In addition, in the high-performance computing context, parallel runtime factors arising from communication patterns and synchronization costs, which are a critical determinant of application performance, are not usually considered by traditional auto-tuning approaches.

To address these issues, we propose a dynamic method for tuning algorithmic parameters of existing scientific codes. By using a language-independent compositional framework, our method supports separate development and transparent integration of tuning code for a given program, so that at runtime the newly inserted code can perform application-specific tuning operations, such as dynamic search of parameter space and performance behavior analysis, to determine optimal parameter values for a given input. Specifically, we target existing programs whose performance-critical algorithms show distinct performance behavior depending on runtime factors and, at the same time, are not well supported by auto-tuning techniques, although our method can be applied to any program in general with tunable algorithmic parameters.

2. Implementing Tuning Support for Existing Parallel Programs

To implement tuning support for existing codes, we essentially take a compositional approach to enhance a given program with an application-specific plug-in module that performs dynamic tuning on the program's algorithmic parameters. The key concepts are described here.

2.1. Tuning as a Separate Concern

While current optimization or tuning techniques are certainly beneficial for basic numerical operations (e.g., matrix-vector multiplication), they are not directly applicable to programs with application-specific algorithms that do not use such operations. Without support for tuning, the user of such applications often employs heuristics or parameter values that are empirically proven "sufficient", which, however, may turn out to be sub-optimal for different execution platforms. On the other hand, manually implementing application-specific tuning strategies for an existing program requires rewriting relevant parts of original program code base, where the modification process can disturb the program's original structure and design purposes by intermixing both new and old codes and blurring logical boundaries between components.

To secure modularity in the design and implementation of tuning strategies over existing programs, we treat tuning as a *separate concern* in evolving software, where tuning plans are organized separately and their implementations are written and managed in independent modules while the main program maintains its original form and structure, unaffected by individual tuning efforts. Tuning code modules are then later plugged into the main program through a compositional framework, as illustrated in Figure 1, which combines both codes to generate an application with a newly added tuning capability that optimizes parameter values in computation algorithms of interest at runtime.

2.2. Tuning with Collective Consideration of Runtime Factors

To account for runtime factors that affect performance behavior depending on different properties (e.g., size) of actual problem instances and parallel characteristics of a given execution platform, we use a dynamic method for tuning target programs. Specifically, we implement a dynamic tuning procedure that searches for optimal values of application-specific algorithmic parameters in the beginning of program execution by periodically measuring and analyzing the runtime performance profile.

3. Settings for Implementing Dynamic Tuning

We showcase our dynamic tuning method in the context of GenIDLEST [6], a CFD simulation code written in Fortran 90 with MPI to solve the time-dependent incompressible Navier-Stokes and energy equations. Specifically, we aim to tune the multilevel Additive Schwarz preconditioned Krylov method used to solve the elliptic pressure Poisson equation which accounts for a large fraction of the total computational time.

3.1. Target Parameters for Tuning

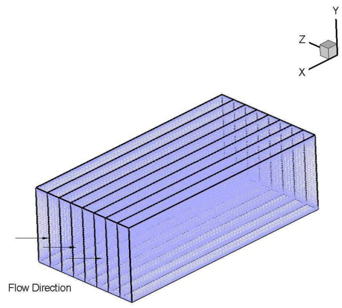
For flexibility, the preconditioner of the GenIDLEST code provides many options and parameters which can be tuned for optimal performance. The target parameters for dynamic tuning are described in the following.

Cache sub-block size in Schwarz preconditioner: Figure 2 shows the hierarchy of the data structure in GenIDLEST in which the computational domain is decomposed into overlapping computational blocks. Each computational block is further divided into smaller sub-blocks. The Schwarz preconditioning is applied by iteratively smoothing the solution in each of the sub-blocks using a multilevel approach. The size of the sub-blocks directly impacts "cache performance" which is critical to the overall time-to-solution. However, because of the complexity of hierarchical memory subsystems and the mathematical characteristics of the system matrix, optimal sub-block sizes can vary substantially on different chip architectures and for different physical problems. In the GenIDLEST code, the input parameter set (ni_blk,nj_blk,nk_blk) represents the number of sub-blocks (and hence the size of each sub-block) in the x-, y-, and z-directions, respectively.

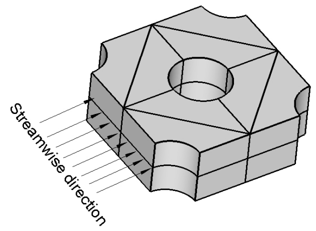
Inner relaxation sweeps: The inner relaxation sweep input parameter, nswp_in_blk, specifies the number of sweeps or iterations performed by the smoother each time a sub-block is visited. In general, increasing this value improves cache performance. However, taking more sweeps also translates to more floating point operations which may not have the desired favorable effect on convergence characteristics and hence could increase the CPU time. The default value is set to 5 in complex flows. In simpler flows, higher values may give better overall CPU time.

3.2. Input CFD Problems

We consider two CFD problems that show distinct physical characteristics to evaluate our tuning method: a turbulent straight channel (Figure 3(a)) and a pin fin array (Figure 3(b)). A detailed understanding of flow and heat transfer characteristics of these problems are important in various applications and affect the design and use of these applications. The computational domain of each problem is shown in (Figure 3(c)). For the turbulent straight channel problem as an example, grids of $64\times64\times64$ computational cells were used in the x-, y-, and z-directions, respectively, which are divided in the z-direction into eight $64\times64\times8$ blocks.



(a) Straight Channel



(b) Pin Fin Array

Problem	Straight channel	Pin fin array
Block size	64×64×8	72×40×48
# Blocks	8	16
Whole domain	64×64×8×8	72×40×48×16

(c) Problem Domain

Figure 3: Problems under Consideration for GenIDLEST Dynamic Tuning

Cluster	Anantham	System G
CPU	AMD Opteron 240	Intel Xeon E5462
Clock (GHz)	1.40	2.80
# Sockets	2	2
# Cores per Socket	1	4
L1 Data Cache	64KB	32KB
L2 Cache	2×1MB	4×6MB (shared by 2)
Memory	1GB	8GB
Interconnect	100Mbps Ethernet	40Gbps InfiniBand
MPI&Compiler	MPICH2 1.0.8 with GNU Compilers 4.2.5	OpenMPI 1.2.8 with Intel Compilers 11.0

Table 1: Architectural Summary of Evaluated Platforms for Dynamic Tuning

3.3. *Execution Platforms*

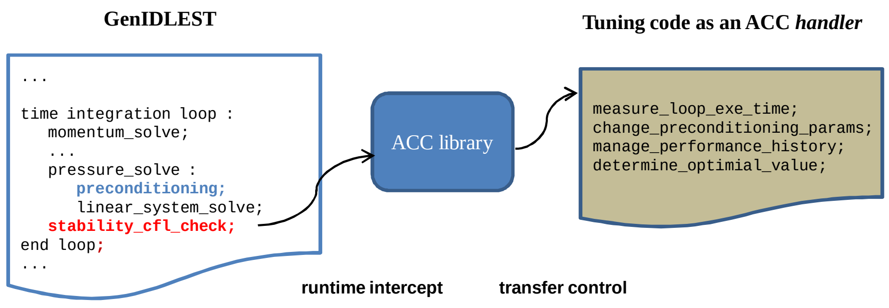
We perform dynamic tuning of GenIDLEST simulations on two cluster systems, called Anantham and System G, respectively. A summary of architectural features of each of the evaluated systems is shown in Table 1.

4. Dynamic Tuning Implementation

The development process of our dynamic tuning method involves identifying control points at which an existing program will be instrumented to plug in tuning code, designing and implementing dynamic search strategies in a separate module, and combining the implemented tuning code into the original program. This section describes these development aspects of our dynamic tuning method.

4.1. *Composition of Dynamically Tuned Software*

To allow separate development of application-specific tuning code and to compose a dynamically tuned application with a given program, *tuning control points* are identified in the original code, such that the execution control



```
...

time integration loop :
   momentum_solve;
   ...
   pressure_solve :
      preconditioning;
      linear_system_solve;
   stability_cfl_check;
end loop;
...
```

```
measure_loop_exe_time;
change_preconditioning_params;
manage_performance_history;
determine_optimial_value;
```

Figure 4: Dynamic Tuning of GenIDLEST through the ACC Framework

is intercepted and transferred to the tuning module. These control points are the places at which application performance is regularly measured and the considered algorithmic parameters are updated by tuning operations. While the programmer is responsible for determining control points in a program, they can be conveniently chosen at the end of the simulation loop in typical scientific applications, such that tuning operations can be performed at regular intervals. However, for concurrently executing processes in a parallel environment, tuning global parameters requires global synchronization, which can degrade application performance. To mitigate the potential performance slowdown, we use a piggybacking technique (see [7]) on existing global functions that execute synchronously near the loop end in the original program, thus effectively amortizing the synchronization overhead by placing separate barriers close together. For the GenIDLEST code, we choose as the tuning control point the stability check function, `calc_cfl()`, which is called at the end of the time integration loop, and which calculates CFL (Courant-Friedrich-Levi) numbers at every preset number of iterations.

To plug in tuning code to GenIDLEST, we use the *Adaptive Code Collage (ACC)* framework [8] (formerly called *Invoke*), a language-independent compositional tool based on the function call interception technique that operates on the compiler-generated assembly source code, and which does not require direct modification of existing code. Therefore, we define an ACC *handler* function for `calc_cfl()` and register it through the ACC registration API, so that the intercepted calls to `calc_cfl()` are diverted to the associated handler, which then performs separately implemented tuning operations. Figure 4 shows the tuning control point in GenIDLEST; the execution control is intercepted at runtime by catching the `calc_cfl()` calls and passed to the tuning module through the ACC library.

4.2. Dynamic Tuning Procedure and Search Strategies

A simple dynamic tuning strategy is to use a two point measure-and-compare scheme. At the first measurement point, execution time for a certain computational interval is measured with the current value of an algorithmic parameter. A new value of that parameter is then selected and the same interval is timed again at the second measurement point, so that the performance between the two points can be compared, and so that the relative improvement or degradation in performance, as a function of the algorithmic parameter, can be estimated. However, this two point scheme may easily fail to determine the right path to a potential optimum in the parameter search space, since, particularly in scientific simulations, physics of the simulated problem (e.g., turbulence in CFD simulations) is not known *a priori* and can change at different phases of computation, resulting in changes in performance behavior even for the same parameter value. To complement the two point tuning scheme to cope with unknown computational progress, we use one more measurement point to obtain the slope of change in execution time for the current parameter value. This slope information is used in the search heuristic, in a way similar to the threshold function in empirical optimization [9].

Figure 5 depicts our dynamic tuning procedure, which comprises four stages (three measurement stages and one compare-and-decide stage). For each algorithmic parameter under tuning consideration, exploring a point in the search space takes four stages: stage 1 (line 4–7) initiates measuring performance with the current value, stage 2 (line 8–11) completes measurement and repeats one more examination to measure the slope in performance behavior for the same parameter value, stage 3 (line 12–18) completes the second measurement and starts measurement with a newly selected parameter value, and stage 4 (line 19–27) completes the new measurement and compares performance behavior based on the measured history and decides whether the search point is better than the current one.

4.2.1. Tuning the Cache Sub-block Size Parameter

An important issue in dynamic tuning is balancing the trade-off between tuning cost and program performance: spending too much time on exhaustive search of parameter space for absolutely optimal tuning points can adversely affect the overall time-to-solution of the simulation with diminishing returns, whereas searching only a small part of parameter space to reduce tuning cost can end up with sub-optimal results, failing to achieve the desired speedup. Since the search space of the sub-block parameter is 3-dimensional, and can grow quite large with only a small increase in problem size, exhaustive search of every combination of (x,y,z) values is not feasible. To reduce the search space, we first eliminate considering the z-direction (nk_blk) of the parameter space and use the value as specified by the user, since, once ni_blk and nj_blk are tuned, further variations in the sub-block size by changing the z component will be small and are unlikely to cause any drastic changes in the GenIDLEST performance behavior.

Secondly, to further reduce the search space of the remaining (ni_blk,nj_blk) pair, we employ a two-phase tuning scheme that performs *coarse search* in the first phase and *fine search* in the second phase. In the coarse search phase,

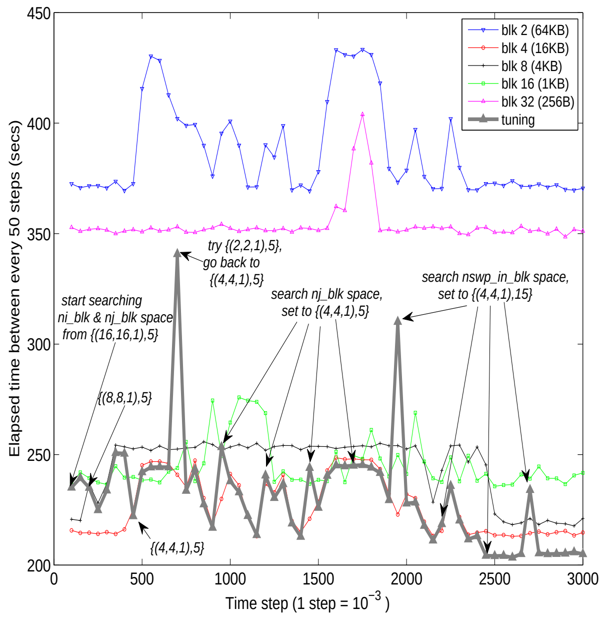
```
**Parameters** $p_1, p_2, ..., p_i, ..., p_k$
**if** $p_1, ..., p_{i-1}$ *are marked tuned and* $p_i$ *is not* **then**
    **switch** *stage* **do**
        **case** *1*
            $elapsed\_time \leftarrow 0$;
            start measuring elapsed time with the current value of $p_i$;
            break;
        **case** *2*
            $elapsed1 \leftarrow elapsed\_time, elapsed\_time \leftarrow 0$;
            start measuring elapsed time with the current value of $p_i$;
            break;
        **case** *3*
            $elapsed2 \leftarrow elapsed\_time, elapsed\_time \leftarrow 0$;
            choose a new value for $p_i$ in the search space;
            broadcast the new value of $p_i$ from root to all processes;
            change $p_i$ with the received value;
            start measuring elapsed time with the new value of $p_i$;
            break;
        **case** *4*
            $elapsed\_new \leftarrow elapsed\_time$;
            compare performance of the new and the old values of $p_i$;
            determine either the old or the new value as the current optimum;
            broadcast the selected optimum of $p_i$ from root to all processes;
            change $p_i$ with the received value;
            update the search space of $p_i$;
            **if** *the search space of* $p_i$ *is empty* **then** mark $p_i$ as tuned;
            break;
        **otherwise** break;
    **end**
**end**
```

Figure 5: Procedure for Dynamic Parameter Tuning at the Intercepted Control Point

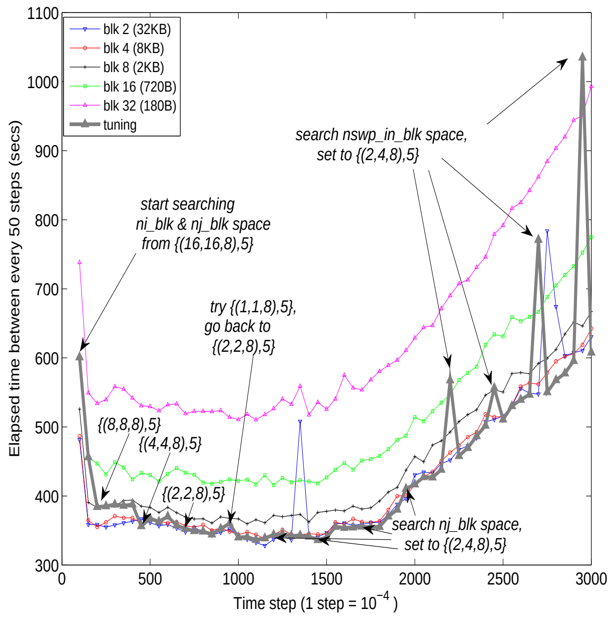
the 2-dimensional pair is treated like a 1-dimensional variable by changing both ni_blk and nj_blk together by same factor (such as divide-by-2). In addition, exploiting the fact that after a certain optimal point the GenIDLEST performance becomes worse as the number of sub-blocks increases (the sub-block size becomes smaller), we decrease the number of sub-blocks (so that the sub-block size becomes bigger) in the coarse search phase, starting from large initial values for the pair, which will exhibit a certain period of improved performance throughout the search. Once a point is reached where performance starts deteriorating, the search process backs off to the previous point and continues by changing both ni_blk and nj_blk with a decrement of 2 at each exploration step, in an attempt to gradually converge to an optimum in the subspace of (ni_blk,nj_blk). With the (ni_blk,nj_blk) value found in the coarse search phase, the fine search phase starts to tune nj_blk further, with ni_blk's value fixed now, by searching a 4-point neighborhood in the y-direction with an increment (or decrement) of 2, and selecting the point in the neighborhood that shows the best performance.

4.2.2. Tuning the Inner Relaxation Sweep Parameter

Once tuning of the cache sub-block parameter is complete, we start tuning the inner relaxation sweep parameter. Unlike the search space of the sub-block size parameter that is directly dependent on the input problem size, the possible number of smoothing iterations on each sub-block is not limited or restricted by the problem size, which allows flexibility in defining its search space. In such a case, practical experience with the target code, together with understanding of tuned algorithms, helps to narrow down the parameter search space, enabling effective search strategy design. Therefore, we assume the default value of 5 as a reasonably obtained one from GenIDLEST simulation practice, and use it as a starting guess. In addition, we examine the four-point neighborhood of the default value by defining the search space as {1,5,10,15,20}.



(a) Straight Channel Problem on Anantham



(b) Pin Fin Array Problem on System G

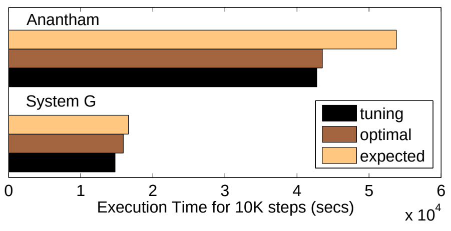
Figure 6: Dynamic Tuning Progress until 3000 Time Steps

5. Experimental Results

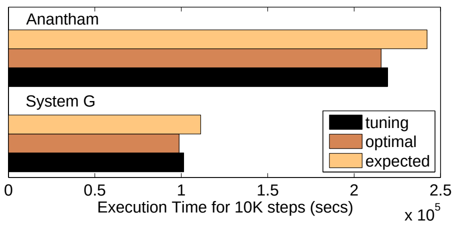
Figure 6(a) shows the dynamic tuning progress of GenIDLEST for the straight channel problem on Anantham for the first 3000 time steps, where the thick gray solid line represents the elapsed time measured at every 50 steps during the tuning process. Plots measured for a set of different parameter configurations without tuning are shown together for comparison. With an initial value of {(16,16,1),5} for {(ni_blk,nj_blk,nk_blk),nswp_in_blk}, the enhanced GenIDLEST program starts tuning first with the sub-block parameter by periodically measuring elapsed times and exploring a new parameter value, where the parameter is changed first to (8,8,1), then to (4,4,1), and so forth at each exploration step according to the search scheme. The sub-block parameter finally settles down to (4,4,1) at time step 1700. Compared with other plots for fixed sub-block parameter values in the figure, the tuning progress plot closely follows the execution time profile with the same parameter value after each update of the parameter, which shows the effectiveness of our dynamic tuning method. Tuning of inner relaxation sweep parameter follows after the sub-block parameter is handled, which completes with the value 15 at time step 2700. Similarly, Figure 6(b) shows tuning progress for the pin fin array problem on System G.

To evaluate our method, we performed dynamically tuned GenIDLEST simulations with each of the CFD problems for 10000 time steps and measured their execution times on each cluster. For comparison, we also measured the performance of GenIDLEST simulations with fixed parameter sets (without tuning). The fixed-parameter simulations were performed for each of the four configurations from *blk2* to *blk16*. As well as the total execution time for 10000 steps, elapsed times between every 50 steps were also measured for each simulation. Figure 7 compares the performance of dynamically tuned GenIDLEST simulations against that of the simulations without tuning. In the figure, 'tuning' represents the total execution time of the tuned simulations, 'optimal' represents a hypothetically best time that can be achieved by switching between the four configurations (from *blk2* to *blk16*) at every 50 steps. The 'optimal' time is calculated by taking the best elapsed time among the configurations at each 50^{th} step and then by summing up these piecewise minimum times for the whole 10000 steps. Finally, the 'expected' time represents an expectation value for the total execution time by randomly choosing a fixed parameter set out of the four configurations.

For the straight channel problem, our method performed better than 'expected' by 26% and slightly better than 'optimal' by 2% on Anantham, and also outperformed both 'expected' by 13% and 'optimal' by 8% on System G.



(a) Turbulent Straight Channel



(b) Pin Fin Array

Figure 7: Performance Comparison of the GenIDLEST code for 10000 time steps

Although the four configurations are only a small fraction of the whole search space, considering that they are practical samples, the results that our dynamic tuning method outperformed even ‘optimal’ cases show its effectiveness. For the pin fin problem, our method performed better than ‘expected’ by 10% but slightly worse than ‘optimal’ by 2% on Anantham, which is coincidentally the same on System G also. Further investigations of adaptation strategies taking into account the specificities of the domain being solved could lead to greater savings.

5.1. Tuning Cost

The runtime function call interception overhead incurred by ACC is small, amounting to less than 140 CPU cycles per intercepted call [8]. In addition, the piggybacking technique to intercept and compose at global operations in parallel MPI programs using ACC is reported to cause almost negligible overhead [7], which also leads to small tuning overhead in our experiments where we intercept only a few dozens of calls in the beginning of the simulations to implement our dynamic tuning strategy. Furthermore, we measured the search cost from exploring new parameter values to be very small, ranging from 0.4% (pin fin array problem on Anantham) to 0.9% (pin fin array problem on System G), where we counted those exploration steps that consumed more time than the previous 50 time steps.

6. Related Work

Auto-tuning: Auto-tuning techniques aim to generate automatically optimized numerical libraries to match the underlying hardware architecture. Model-driven auto-tuning uses analytical models for programs and machine architectures to perform efficient compiler transformations over program control structures such as loop blocking [10], loop unrolling, and software pipelining [11]. However, these compiler techniques may not always generate near-optimal code since the compiler models of the underlying processor architectures are usually too simplified and general to support special math kernels. In addition, the compiler models used may not be up-to-date with the newest hardware. Empirical auto-tuning techniques, in contrast, parameterize the code for a given algorithm over the parameter search space and benchmarks each variant on a given platform to determine the best possible algorithmic option. For example, ATLAS [3] and PHiPAC [4] generate automatically tuned dense linear algebra routines. In contrast, the main objective of our method is to support application programmers to separately reason about domain-specific dynamic tuning strategies and implement them onto existing programs, thus allowing for collective consideration of various runtime factors such as input problem attributes and parallel performance behavior of a whole execution environment.

Computational steering: Computational steering is the practice of manually altering parameters of scientific simulations at runtime through interactive feedback from the user [12, 13]. Since computational steering enables dynamic parameter change decisions to take effect at runtime, it can be beneficial for runtime performance tuning and “what-if” studies for certain problems without requiring stop-and-restart of computation from scratch. However, implementing efficient middleware or runtime support to instantly update the program states at the computation backend with the user’s feedback at the steering frontend is a major challenge in large, distributed systems such as the Grid where communication between remote sites becomes a main source of performance bottleneck. While our method similarly targets dynamic tuning of algorithmic parameters, we support the design and implementation of tuning strategies at the application composition phase, so that at runtime the tuning operations are automatically performed without visualization or interactive feedback system support.

7. Summary and Conclusions

Our compositional approach addresses tuning as a separate concern in software development processes and allows one to separately design and implement dynamic strategies for scientific programs whose domain-specific algorithms are not easily supported by automatically tuned libraries. Unlike most auto-tuning techniques that consider only architectural characteristics of a single machine, our dynamic tuning method includes runtime factors as well such as input size and parallel performance behavior of a given execution platform, thereby enabling optimization of those parameters whose values cannot be determined *a priori* before application launch. Since performing exhaustive search at runtime is not viable with intolerable overhead for a large search space, design of a dynamic search strategy requires trade-off between search space and performance overhead; as demonstrated we are able to achieve upto 26% performance improvement in a practical scientific code with small tuning costs.

[1] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms, in: SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, ACM, New York, NY, USA, 2007, pp. 1–12.

[2] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, K. Yelick, Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures, in: SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–12.

[3] R. C. Whaley, A. Petitet, J. J. Dongarra, Automated Empirical Optimizations of Software and the ATLAS Project, Parallel Computing 27 (2001) 3–35.

[4] J. Bilmes, K. Asanovic, C.-W. Chin, J. Demmel, Optimizing Matrix Multiply using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology, in: ICS '97: Proceedings of the 11th International Conference on Supercomputing, ACM, New York, NY, USA, 1997, pp. 340–347.

[5] R. Vuduc, J. W. Demmel, K. A. Yelick, OSKI: A Library of Automatically Tuned Sparse Matrix Kernels, in: Proc. of SciDAC 2005, Journal of Physics: Conference Series, Vol. 16, Institute of Physics Publishing, 2005, pp. 521–530.

[6] D. Tafti, GenIDLEST - A Scalable Parallel Computational Tool for Simulating Complex Turbulent Flows, in: Proceedings of the ASME Fluids Engineering Division (FED), Vol. 256, ASME-IMECE, 2001.

[7] P. Kang, N. K. C. Selvarasu, N. Ramakrishnan, C. J. Ribbens, D. K. Tafti, S. Varadarajan, Modular, Fine-Grained Adaptation of Parallel Programs, in: ICCS '09: Proceedings of the 9th International Conference on Computational Science, Springer, 2009, pp. 269–279.

[8] M. A. Heffner, A Runtime Framework for Adaptive Compositional Modeling, Master's thesis, Blacksburg, VA, USA (2004).

[9] K. Seymour, H. You, J. Dongarra, A Comparison of Search Heuristics for Empirical Code Optimization, in: iWAPT 2008: the 3rd International Workshop on Automatic Performance Tuning, 2008, pp. 421–429.

[10] J. Dongarra, R. Schreiber, Automatic Blocking of Nested Loops, Tech. rep., Knoxville, TN, USA (1990).

[11] K. Kennedy, J. R. Allen, Optimizing Compilers for Modern Architectures: A Dependence-based Approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[12] R. Marshall, J. Kempf, S. Dyer, C.-C. Yen, Visualization Methods and Simulation Steering for a 3D Turbulence Model of Lake Erie, SIGGRAPH Comput. Graph. 24 (2) (1990) 89–97.

[13] S. G. Parker, C. R. Johnson, SCIRun: A Scientific Programming Environment for Computational Steering, in: Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM), ACM, New York, NY, USA, 1995, p. 52.