Journal of Computational Science 3 (2012) 28–45

Contents lists available at SciVerse ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Implementing modular adaptation of scientific software

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

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

ARTICLE INFO

Article history:
Received 26 July 2011
Received in revised form 19 January 2012
Accepted 29 January 2012
Available online 6 February 2012

Keywords:
Program adaptation
Scientific computing
Modular programming

ABSTRACT

Scientific software often needs to be adapted for different execution environments, problem sets, and available resources to ensure its efficiency and reliability. However, for existing programs, implementing adaptations by directly modifying source code can be time-consuming, error-prone, and difficult to manage for today's complex software. In this paper, we present a modular approach to realizing adaptation for existing scientific codes. By treating adaptation as a separate concern, our approach supports the development of application-specific, context-aware adaptation schemes without directly modifying the original code. Our approach uses a compositional framework that offers language-neutral mechanisms to integrate separately written adaptation code with existing code. Using our approach, scientific programmers can focus on the design and implementation of adaptation schemes separately from the original code development, and then compose an adaptive application whose original capabilities are enhanced in diverse aspects such as performance and stability. Our compositional approach enables fine-grained adaptation, so that an application's program behavior is controlled at the function or algorithm level by adaptation code plugged into the application. By applying our approach to real-world scientific applications to implement various adaptation scenarios, we demonstrate applicability and effectiveness for adapting scientific software.

1. Introduction

Program adaptation is a process of changing the runtime behavior of a program in a different way from its original conception to achieve a certain purpose necessitated by the user. The need for adaptation arises from different application requirements such as performance, stability, and program analysis. Changing system state by modifying global variables and adapting functional behavior of a program module (a chunk of code usually abstracted as a procedure) are typical adaptation operations.

In scientific computing, adaptation is especially relevant because scientific software often needs to change its original behavior for different execution environments, problem instances, and available resources to provide sufficient efficiency and reliability. A variety of approaches exist at different levels to implement adaptive behavior in scientific programming. At the algorithmic level, adaptive algorithms [12] or multi-method algorithms [4,6] can be adopted from the beginning of the software design process. At the framework level, adaptation can be supported from tools that perform dynamic algorithm selection or switching to find the best algorithmic option for a given problem at runtime [3,36].

While these approaches can be useful for developing adaptive applications from scratch, implementing adaptations for existing programs requires modification over original source code. For fine-grained adaptations, where the behavior of an application needs to be manipulated at the algorithmic level, the modification process can be a challenging task for scientific codes. The challenge is particularly great for legacy codes, many of which were written in early versions of Fortran, which lacked sophisticated software engineering mechanisms for modularity support, and because modern programming practices that encourage modularity and adaptability have not always been used.

To address these challenges, we present a modular approach to implementing adaptations for existing scientific programs. We take a compositional approach, where software adaptation is treated as a separate concern in the software development process and application-specific adaptation schemes can be effectively factored out in separate code. Therefore, by using our approach, the programmer can focus on the design and implementation of adaptation scenarios for a given application. Later, separately developed adaptation code is integrated with the original program through a compositional framework. Our approach is language-neutral and enables composition of an adaptive application from multiple

* Corresponding author.
E-mail addresses: kangp@cs.vt.edu, pilsungk@gmail.com (P. Kang).

1877-7503/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.jocs.2012.01.007

software modules written in different languages, including Fortran.

The remainder of the paper is organized as follows. Section 2 presents key concepts of our compositional approach to implementing modular adaptation of scientific software. Section 3 describes the *Adaptive Code Collage* (ACC), a compositional tool we use to realize adaptation. Section 4 illustrates the power of modular adaptation with a series of adaptivity scenarios applied to a computational fluid dynamics (CFD) code. (Condensed descriptions of these examples have appeared in [20,21].) Each of these CFD examples is precisely defined and automatic, in the sense that the point in the code where the adaptation will occur and the strategy used for adaptation are specified before an application is launched. A second set of adaptivity scenarios is presented in Section 5, based on a cell biology simulation code. These scenarios are more loosely defined and interactive, in that we enable users to decide at run-time the point at which adaptation will occur, and the type of adaptation that will be done. Section 6 contrasts our work with related research projects. And finally, Section 7 summarizes the contributions of the paper.

2. Compositional approach to modular adaptation

Fig. 1 shows an overview of our compositional approach. Here, functions are the basic unit where adaptation happens, i.e., adaptive behavior results from implementing adaptation operations associated with functions of interest in the original code. Such adaptation target functions are specified by the programmer as *adaptation control points*—points where our compositional framework can plug in separately written adaptation code.

Our framework offers effective and efficient mechanisms necessary for the programmer to use in describing desired adaptation logic with respect to a given application. We discuss the benefits of taking our compositional approach in implementing scientific software adaptation in the following.

2.1. Factoring out the adaptation logic

In our compositional approach, we treat adaptation as a *separate concern* in evolving software, where adaptation plans are organized separately and their implementations are written and managed in independent modules. Thus the main code base maintains its original form and structure, unaffected by individual adaptation efforts. Adaptation code modules are then later plugged into the main program through a compositional framework, which combines both codes to generate an application with a newly added adaptation capability that adjusts original application behavior at runtime. This is similar to the Aspect-Oriented Programming (AOP) paradigm [23], where the programmer identifies control points (or join points in AOP terms) in the original program, writes certain aspect operations in a separate code, and weaves in the aspect code at the control points in the original program through an AOP framework, thereby resulting in an enhanced program with a specific runtime capability.

2.2. Fine-grained adaptation

By defining adaptation control points at the interfaces of subprogram modules, our compositional approach conveniently achieves effective, fine-grained control over application behavior, where adaptation strategies can be designed to monitor and react to changes in internal program states. Global state variables can be accessed from the adaptivity code by declaring these variables as external. Our framework also provides function parameter control APIs, which enables an extra level of flexibility in fine-grained adaptation. Function parameters are usually not exposed as globals in a program, but they can hold important runtime program state for certain adaptation purposes. Through parameter control APIs, we allow dynamic program state that is communicated between modules not only to be accessed (e.g., to check computational progress) but also to be manipulated to adjust the program's runtime behavior.

2.3. Fortran support

Fortran has traditionally been the language of choice in the scientific computing domain and a large portion of legacy scientific software written in Fortran is still used to build modern scientific applications. For instance, numerical software more than a few decades old is not unusual in the listings in the Netlib repository,[1] Unfortunately, most legacy scientific computing software is tailored toward static execution. The execution path of a typical scientific application is predetermined at compile time and rarely changes in response to any runtime events. This inherent stiffness in the traditional scientific programming practice is a serious obstacle that make it hard to adapt existing scientific codes.

To support legacy Fortran programs, our approach uses a language-neutral framework that implements composition at the assembly language level. This method works for code modules written in Fortran as well as other high level languages as long as the original code can be compiled to generate assembly output.

3. The Adaptive Code Collage framework

We use a compositional framework called Adaptive Code Collage (ACC) [15,19] to combine adaptation code with a given program. The ACC framework's APIs support program composition through function call interception (FCI), a technique whereby function calls are intercepted at runtime to alter the operations performed when the call is actually made. With FCI, the desired operations may be performed either just before or after the call, or at both times. Because specific function calls are caught and manipulated at runtime to change their original behavior, FCI enables transparent code modification without directly rewriting the existing code.

The ACC framework implements FCI at the assembly language level by replacing the x86 `call` instruction in the instrumentation target code with a call to its own *interception handler*, a piece of newly inserted code responsible for modifying the original function. Therefore, instead of directly altering the original high-level language code, we apply ACC to the assembly output generated from the compiler to insert new code, thus realizing modular development of software adaptation. Moreover, due to the assembly level code modification, ACC can be applied independently of the original source language, making it suitable for Fortran legacy codes as well as programs written in C or C++.

The application development process using ACC involves three steps. First, the assembly output of the target subprogram is patched to divert the target function calls to ACC. Next, the intended adaptive plan is implemented in a new code module, in which the desired operations are described and the original calls can be modified through ACC's parameter manipulation APIs. ACC transfers execution control of the diverted calls to the new module at execution time. Finally, the user needs to set up an association between the target function and the new module by using ACC's registration APIs.

Fig. 2 shows how ACC operates to combine a new code module with a given program. Here, the `f()` and `g()` functions are specified

[1] http://www.netlib.org.

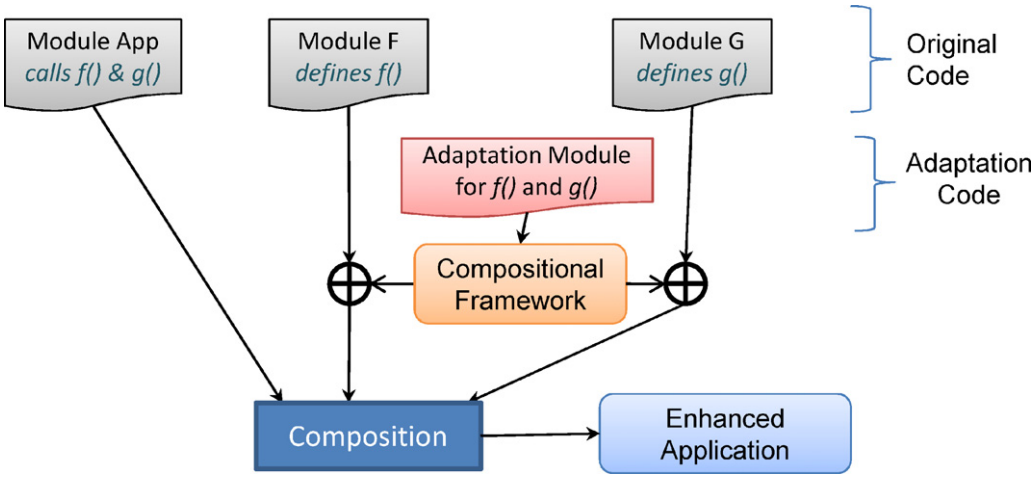


Fig. 1. Composition of an adaptive application with existing code.

as adaptation control points and their handler code, `f_handler()` and `g_handler()`, respectively, are written by the programmer to adapt their original behavior. The handlers are then registered and associated with the corresponding functions through ACC, thus effectively integrating them with the original program. At runtime, every call to the target functions is intercepted and program control is diverted to the associated handler through ACC's bookkeeping data structure.

Although we say ACC implements FCI functions, its capabilities are actually quite general, including:

- Function interception: User-specified function calls can be intercepted before or after their lifetime through assembly code patching. This means that the basic unit to support code insertion is a procedure or function.
- Registered callbacks: A piece of code (i.e., handler code) can be registered with a function so that the code can be performed whenever the associated function is intercepted. In effect, the programmer implements adaptation by writing handler code for function of interest within a given program and plugging in the code to the application.
- Parameter manipulation: Function call parameters can be accessed and modified through stack manipulation before the call is actually performed, which allows another degree of flexibility in modifying the original program behavior.
- Function parameter remapping: Through sophisticated stack manipulation, the entire parameter list of a function can be remapped for a new function with a different parameter signature.

Perhaps the most useful feature of ACC is the ability to remap the entire parameter list of a function to a new function even with a different signature. We make heavy use of this functionality to implement the adaptive use cases presented in this paper.

3.1. ACC for adapting scientific software

Although ACC is primarily a programmer's tool for *factorizing* adaptivity with an existing code base to enable the plug-and-play of different adaptive strategies, we focus on scientific codes mainly because they offer rich adaptation possibilities. In scientific computing, it is quite common that there are multiple algorithmic options for the same problem, which naturally encourages application behavior adaptation depending on different execution contexts. Therefore, adaptation for improving performance or stability has long been used in many scientific applications, and methods to automatically switch between algorithms are critically important. ACC can be used for different adaptation strategies and in particular, it is well-suited to realizing adaptation patterns or *schemas* [30], such as algorithm switching and systems control, that commonly occur in diverse areas of scientific computing. Most of these adaptations rely on monitoring and adjusting state variables or diverting function calls to improve certain aspects of the application's runtime behavior. By using ACC, such capabilities can be

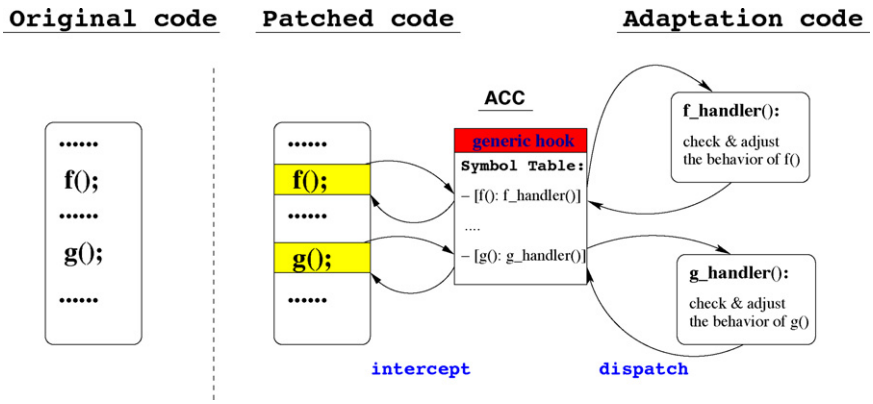


Fig. 2. Function call interception using ACC.

Table 1
Implementation aspects of GenIDLEST adaptation.

	Time step change	Flow model switch	Dynamic tuning
Purpose	Improve stability	Enhance accuracy	Improve performance
Type of scheme	Automatic adjustment	User's dynamic decision	Automatic tuning
States to monitor	CFL number via `mpi_allreduce`	Stream-wise velocity written to a log	Loop execution time
Control point	`mpi_allreduce` inside `calc_cfl`	`calc_cfl` in the loop	`calc_cfl` in the loop
Adaptive logic	Adjust time step to confine CFL number within certain bounds	Switch flow model and activate turbulent data structures	Explore parameter space to determine optimal values
Communication	Not necessary	Broadcast of user's decision	Broadcast of parameter values

easily implemented in a modular fashion that factors them out into a separate code which can be better managed and maintained.

Specifying what needs to be adapted and how differs depending on applications and is the purview of domain-specific information. As such, a fair amount of understanding of the control flow and program structure of a target application at the function level is necessary for the effective development of application-specific adaptation using ACC. However, most scientific computations are structured based on iterative loops and the end of a loop exhibits stable computation states at runtime. In particular, parallel applications typically synchronize concurrent execution at the loop end to construct consistent intermediate results. Hence, parallel communication functions near the computation loop are prime candidates for adaptation control points in using our methods. And in most cases, identifying and specifying these functions as control points tend to be quite intuitive.

A further description of the ACC framework can be found in our previous publication [19].

4. Case study 1: computational fluid dynamics simulations

We have used the ACC framework to implement a variety of adaptation scenarios in the context of real scientific computing codes. In the first case study we target Generalized Incompressible Direct and Large-Eddy Simulations of Turbulence (GenIDLEST) [29] code, a parallel computational fluid dynamics (CFD) simulation program written in Fortran 90 with MPI to solve the three dimensional, time-dependent incompressible Navier–Stokes and energy or temperature equations. In this section, we describe three applications of our fine-grained adaptation approach to GenIDLEST, each of which is designed to enhance the capability of CFD simulations in different aspects such as simulation stability, accuracy, and performance. The implementation aspects of the three adaptation scenarios are summarized in Table 1.

4.1. Synchronous parallel adaptation

Implementing parallel adaptive behavior requires adaptive logic operations to take place synchronously at clearly defined program control points that are shared across all the participating processes. This is important for implementing fine-grained adaptation strategies with SPMD (Single Program, Multiple Data) programs where program behavior needs to change dynamically in response to changes in program state, because asynchronous adaptation in a parallel program can cause race conditions among the processes and make the entire computation invalid. For example, if one process changes a global simulation parameter or algorithm, and continues the computation, before another process makes the corresponding adaptation, the result may be inconsistent.

However, synchronous adaptation can degrade performance if the adaptivity code involves extra global communication and synchronization. To mitigate the potential performance slowdown caused by adaptive global operations, we plug in the adaptivity code at global synchronization points that already exist in the original program, thus placing separate barriers (one from the original code and the other from the new adaptivity code) close together and making the combined overhead smaller. By having the adaptivity operations "piggyback" onto the existing communications that are executed synchronously across the parallel environment, monitoring and adjusting the program states can also be performed synchronously without explicitly using extra global operations. Furthermore, since the end of a computation loop typically exhibits stable system state and consistent intermediate results, by placing adaptation code at the end of a loop, coherent results can be assessed without disturbing the ongoing computation.

4.2. Input CFD problems and execution platforms

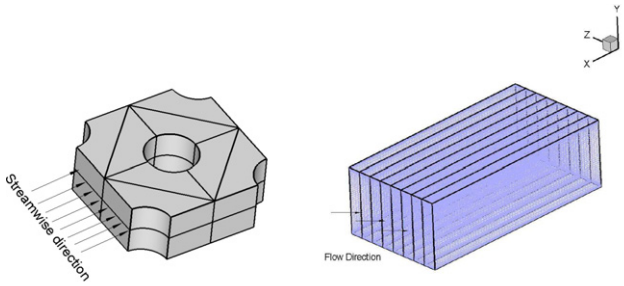
We consider two CFD problems that show distinct physical characteristics to evaluate our adaptation method.

- *Pin fin array*: The geometry of the pin fin array simulation is shown in Fig. 3(a). Extended surfaces or fins have been used extensively to augment the heat/mass transfer from or to a surface primarily by increasing the transfer area and/or increasing the heat/mass transfer coefficient. Reducing the size and weight requirements of equipment necessitates optimal designs of these systems, which in turn requires a detailed understanding of flow and heat transfer characteristics. Most of these fins are in the intermediate slenderness ratio range of 0.5–4 and the typical Reynolds numbers are in the range of 150–4500 for electronic cooling applications. For the GenIDLEST simulation, we divided the geometry into 16 block structures so that the maximum degree of parallelism is 16, where each block is assigned to one MPI process.
- *Turbulent straight channel*: Turbulent flow phenomena are important in various applications and affect the design and use of these applications, where the turbulent channel is one of the canonical geometries used to evaluate the properties of numerical approximations and turbulence models by comparing the results to known measured or computed solutions. Calculations are carried out in turbulent channel flow at a Reynolds number $Re_\tau = 180$ based on channel half-height and wall friction velocity. Grids of $64 \times 64 \times 64$ computational cells were used in the x-, y-, and z-directions, respectively, which are divided in the z-direction into eight $64 \times 64 \times 8$ blocks (Fig. 3(b)).

We use two cluster systems, called Anantham and System G respectively, to perform the GenIDLEST simulations. A summary of architectural features of each of the systems is shown in Table 2.

4.3. Adaptation for stability: automatic adjustment of simulation time step

The stability of the simulation depends on the time step size used. Based on observed Courant-Friedrich-Levi (CFL) numbers one could discern if the simulation is proceeding towards convergence or is becoming unstable. Current practice of running GenIDLEST simulations records intermediate results at the end of a preset number of iterations onto the disk, thereby allowing the user to stop the execution and restart from the last known stable state when the



(a) Pin fin array (b) Straight channel

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

Fig. 3. CFD problems under consideration for GenIDLEST adaptation.

user determines the running simulation is diverging. By plugging in a simple adaptivity module, the enhanced GenIDLEST simulation (requiring no modifications to the original GenIDLEST code) will incrementally adjust the time step value at runtime, allowing the computation to proceed in a stable manner.

4.3.1. Implementation

Fig. 4 shows an overview of our automatic time step adjustment implementation, where the original GenIDLEST code and the adaptation code are combined together at a control point (`stability_cfl_check` in the figure). At the end of every preset number of iterations in the execution flow of GenIDLEST, a local CFL number is calculated by each MPI process, and then the global CFL value is computed in the stability check module using a reduction operation (`mpi_allreduce`) across all the processes. This point is a good candidate for adaptivity code insertion, since by catching and imposing operations at this synchronization point, the newly inserted code can also be executed in synchronization, thereby avoiding dangerous race conditions among the processes. Furthermore, catching the global reduction call also makes it easy to monitor the global CFL number because its value is passed as the second parameter of the function. ACC's parameter accessing APIs can be utilized to access this value.

Two state variables need to be controlled by the adaptation code: one for global CFL number and the other for the time step parameter. As a specific adaptation scheme, we employed a simple multiplicative increase, multiplicative decrease algorithm with upper (`CFL_U_THRESHOLD`) and lower (`CFL_L_THRESHOLD`) threshold values for the CFL number, such that the time step is increased or decreased by a preset factor if the current CFL number becomes out of the bounds defined by the thresholds. Importantly, the entire adaptive logic operations are performed synchronously at the call sites of `mpi_allreduce` without involving any extra global operations, thereby achieving efficient parallel program adaptation.

4.3.2. Experimental results

Fig. 5 shows the results of GenIDLEST enhanced with the constructed adaptivity module for the pin fin array simulation, with different initial values of time step ranging from 10^{-3} to 10^{-5}. `CFL_U_THRESHOLD` and `CFL_L_THRESHOLD` were set to 0.5 and 0.25, respectively. The graphs show how the CFL value changes as the time step parameter is controlled by the new module, thereby maintaining the stability of the simulation. Interestingly, it also shows that the time step in all cases converge to somewhere around 1.7×10^{-4}, which might be the optimal value for the model, regardless of different starting values. Therefore, an adaptive logic based on a sophisticated CFD theory might be devised to find the optimal time step for more generalized problems through our composition method.

Table 2
Architectural summary of execution platforms for GenIDLEST adaptation.

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	64 kB	32 kB
L2 cache	2 × 1 MB	4 × 6 MB (shared by 2)
Memory	1 GB	8 GB
Interconnect	100 Mbps Ethernet	40 Gbps InfiniBand
MPI&Compiler	MPICH2 1.0.8 with GNU Compilers 4.2.5	OpenMPI 1.2.8 with Intel Compilers 11.0

4.4. Accuracy adaptation: runtime change of flow models

The predicted heat transfer and flow characteristics depend on the selection of the appropriate flow model. A fundamental distinction is between laminar and turbulent flow models, and simulations of interest often require a switch from one to the other. This problem becomes acute when the Reynolds number is in the transition region between laminar and turbulent flows. Thus it becomes important to change the flow model from laminar to turbulent once instabilities arise in the flow field, for a simulation that is started assuming the flow is laminar. Two Large Eddy Simulation (LES) turbulent models are considered in this study—the Smagorinsky model (SM) and dynamic Smagorinsky model (DSM) [13]. The most commonly used model is the Smagorinsky model, where the eddy viscosity of the subgrid scales is obtained by assuming that the energy production and destruction are in equilibrium. The drawback of this model is that the model coefficient is kept constant, while in reality it should vary within the flow field depending on the local state of turbulence. The dynamic Smagorinsky model computes the model coefficient dynamically, which overcomes the deficiencies of the Smagorinsky model by locally calculating the eddy viscosity coefficient to reflect closely the state of the flow [13]. The advantage of the DSM model is that the need to specify the model coefficient is eliminated, making the model more self-contained, but with an additional computational expense of 10–15%.

4.4.1. Implementation

The simulated flow model in GenIDLEST is set initially by the user through an input specification parameter, namely `i_les`: 0 for laminar, 1 for Smagorinsky, and 2 for Dynamic Smagorinsky model. Hence, the program state needs to be accessed and changed at runtime via this variable. Importantly, the change should be made synchronously across all processes to maintain the consistency of the parallel computation. To this end, we plug in the adaptivity module at the call site of the CFL reduction function as shown in Fig. 4, because it is executed in synchronization across all the MPI processes, providing a safe place for carrying out adaptation operations without modifying the original code and disturbing

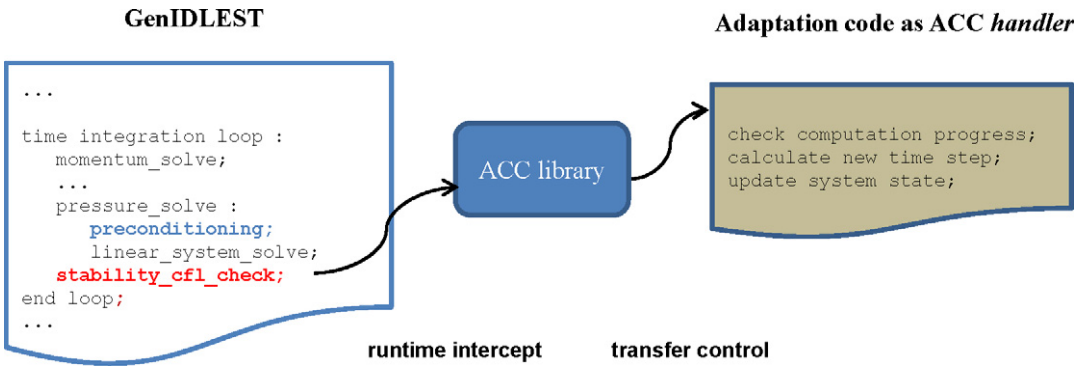


Fig. 4. GenIDLEST execution flow.

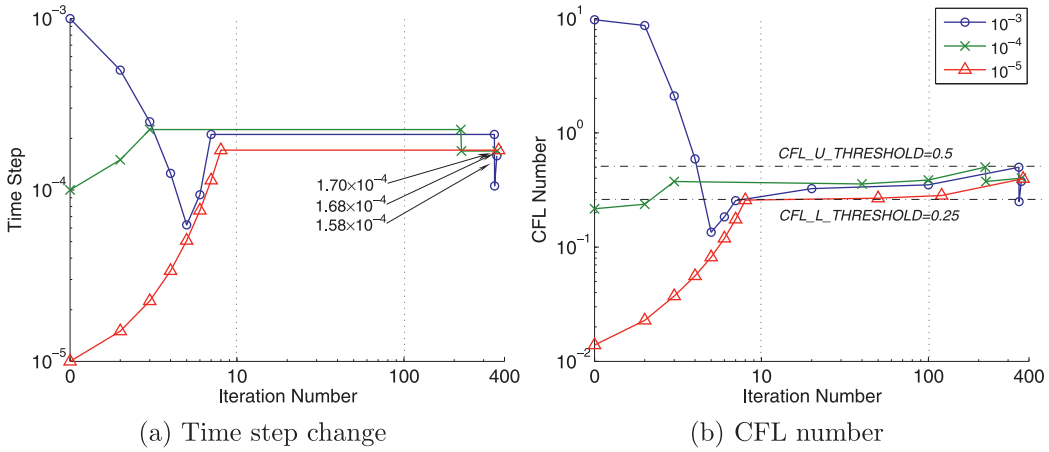


Fig. 5. Automatic adjustment of the time step parameter.

the parallel execution flow already established in the original GenIDLEST. Specifically, the adaptivity code checks if the user wants to change the flow model, for which we make use of Unix signals (e.g., `SIGUSR1`) that can handle immediate, unanticipated user decisions to switch the flow model. These user-sent signals set a flag in the root process, which will pause accordingly with a simple user interface in the next iteration to accept the user's adaptation decisions, which in turn are broadcast to the other processes.

4.4.2. Experimental results

The computational domain consists of an array of cylindrical pins of circular cross section, where a uniform heat flux is applied from the end walls assuming fin efficiency close to 100% for mini-channels. The variation of the velocity in the direction of flow (stream-wise) is plotted in Fig. 6, showing the points in time when the flow models are switched from laminar to SM and then to DSM. The stream-wise velocity initially decreases, as the simulation proceeds towards the solution, which occurs till about 0.6 time units. After this simulation time, we see that the stream-wise velocity tends to vary with time, indicating the development of flow instabilities, and implying that the initial assumption of laminar flow is no longer valid. Thus the model is switched to SM at time 1.0. The drawback with the SM model, as mentioned earlier, is that the model coefficient is set to a constant value, but in reality the coefficient varies with the local state of turbulence, thus it becomes imperative to change the model from SM to DSM. This switch is done after a few hundred iterations (at time 1.4) to make sure that the switch from laminar to turbulent model does not introduce instabilities in the computation.

The switch from laminar to turbulent flow model has a significant effect on the heat transfer. This is shown in Fig. 7(a) and (b), which show the variation of the Nusselt number at the channel walls, which is a measure of heat transfer at that location. The dotted line shows the region of interest, which is at the front of the pin in the line of fluid flow. The laminar flow model does not capture the heat transfer effects at the front of the pin, predicting lower heat transfer rates at the pin front than the turbulent model, thus justifying the model switch from laminar to DSM. This switch shows the capabilities of the adaptive scheme, since to effect the switch without it would have meant stopping the current execution and then restarting the simulation after effecting the required change.

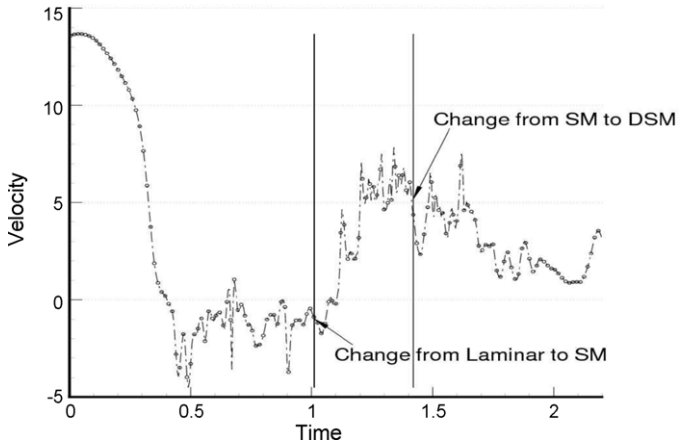
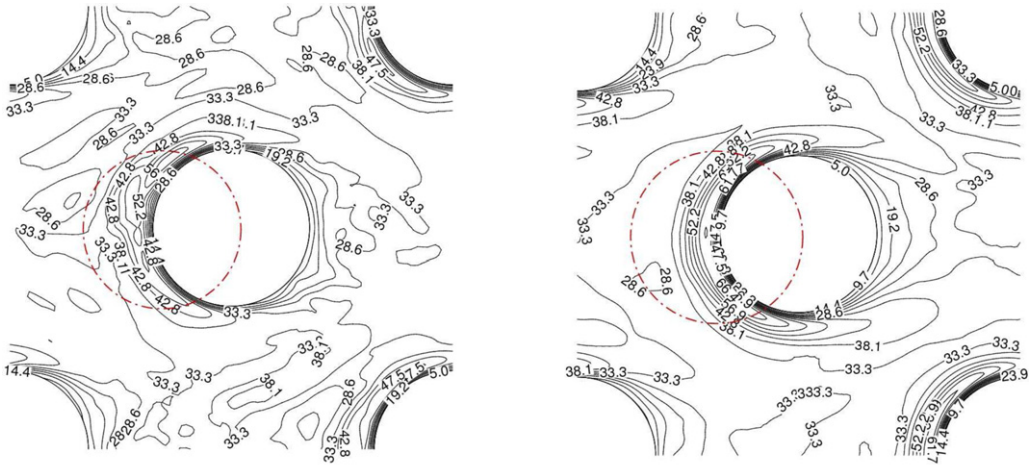


Fig. 6. Variation of stream-wise velocity with flow model change.



(a) Laminar model (b) Turbulent DSM model

Fig. 7. Dynamic flow model change from laminar to turbulent.

4.5. Performance adaptation: dynamic tuning of algorithmic parameters

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 [9,34]. However, most of the automatic tuning efforts are focused on basic computational kernels in numerical computing such as vector and matrix operations, and tuning support for domain-specific algorithmic parameters is lacking, where even a modest effort in parameter tuning can provide significant speedups. 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.

In this section, we apply our modular adaptation approach to implement dynamic tuning of algorithmic parameters of scientific codes. 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.

4.5.1. 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. Tuning implementation modules are plugged into a given existing program through our ACC compositional framework.

4.5.2. Target algorithmic parameters

We aim to tune the preconditioning code in the Krylov method for solving pressure equations, which accounts for a substantial amount of the computation in GenIDLEST simulations. For the preconditioning smoother, GenIDLEST provides many options and parameters to choose from to tweak the performance of the smoothing process such as preconditioning method, inner and outer loop relaxation parameters over decomposed domains, and number of smoothing iterations for subdomains, all of which are application-specific parameters for implementing preconditioners in GenIDLEST.

For this example, we use GenIDLEST's Jacobi preconditioner with the relaxation factor set to 0.95, since, among available choices such as SSOR (Symmetric Successive Over-relaxation) and ILU (Incomplete LU) decomposition, this setting has been found to perform quite well in most GenIDLEST simulations. The two target algorithmic parameters in the preconditioning code considered for our dynamic tuning method are described in the following.

- *Sub-blocks in domain decomposition*. In the data decomposition hierarchy of GenIDLEST simulations, a given whole problem domain is divided into multiple blocks, each of which is then assigned to individual processes in parallel execution. The processor domain is further subdivided into smaller sub-blocks for preconditioning, which groups a set of multiple sub-blocks into a mesh block for coarse level smoothing, as is done in multigrid methods. In the GenIDLEST code, the (ni_blk,nj_blk,nk_blk) parameter set represents the number of sub-blocks in *x*-, *y*-, and *z*-directions, respectively. For example, one of the CFD problems considered in this paper has computational geometry of size $64 \times 64 \times 64$ and it is divided into eight $64 \times 64 \times 8$ blocks, such that each block is assigned to one MPI process. Then, if the parameter is set to (2,b,1), each sub-block will have $32 \times 32 \times 8$ grid cells in the preconditioning process. The cache sub-block is introduced solely for the solution of linear systems. Each time the preconditioner is invoked in the Krylov method, it does most of its work on each of the cache sub-blocks individually. Because of the small size of each sub-block and the repetitive sweeps or iterations done each time a block is visited, not only does this method

provide excellent preconditioning but also increases performance on a single processor by using cache very effectively [33].

- *Inner relaxation sweeps*. The inner relaxation sweep 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 and so, it affects convergence characteristics which could have a detrimental effect on CPU time. The default value is set to 5 in complex flows. In simpler flows, higher values may give better overall CPU time.

Table 3 summarizes the target GenIDLEST parameters considered for dynamic tuning in this application.

CFD simulations with GenIDLEST show distinct performance behavior depending on both the preconditioning parameters used and the execution platforms. For example, Fig. 8 shows execution time measurements on each of the clusters with the CFD problems for 10,000 simulation time steps. Fig. 8(a) shows measurements for the straight channel problem on Anantham, where 2 MPI processes were used on each node. The plots represent 5 different configurations for the sub-block parameter. In the top, elapsed times measured between every 50 time steps are plotted for each of the 5 configurations, and the cumulative sums of elapsed times are plotted in the bottom.[2] Here, we varied the 3-dimensional sub-block parameter only in the x and y directions while keeping nk_blk = 1. Also the values of both ni_blk and nj_blk were set equal. Therefore, for instance, *blk*2 corresponds to (ni_blk,nj_blk,nk_blk) = (2, 2, 1), which specifies the decomposition of the per-process block of (64, 64, 8) grid cells into $(2 \times 2 \times 1 = 4)$ sub-blocks. This results in each sub-block consisting of $(32 \times 32 \times 8)$ grid cells, which take 64 kB for double precision data. Although these 5 configurations are only a small fraction of the entire search space of the sub-block parameter, they can serve as practical examples because they are simple to specify and the distribution is in proportion to powers of 2. The plots show the connection between GenIDLEST performance and the sub-block size—the performance gets better as the number of sub-blocks decreases (i.e., the sub-block size increases) up to a certain point, which happens to be at *blk*4 where the parameter is (4, 4, 1) and the sub-block size is 16 kB for Anantham. Further decrease in the number of sub-blocks results in rapid performance degradation, as shown by the plot for *blk*2. Similarly, Fig. 8(b) shows elapsed time plots (top) and cumulative execution time plots (bottom) measured at each 50 time steps for the pin fin array problem on System G. The best performance is seen at *blk*2 when the parameter is set to (2, 2, 1), which corresponds to the largest sub-block size (32 kB) among the 5 configurations we tested.

With respect to the shapes of the graphs, Fig. 8(a) for the straight channel shows performance curves with an almost linear time variation with intermittent increase in the elapsed time. The linear profile is due to the fact that the calculations have proceeded to become fully developed and the jumps in elapsed time indicate the turbulent nature of the simulation. The performance curves in Fig. 8(b) for the pin fin array show an initial increase followed by a decrease in elapsed time. This observed variation is because the solution is proceeding towards a fully developed condition, leading to this profile. Obviously, the solution behavior of the flow problem depends on the stage of the calculation. From the data in Fig. 8, it can be inferred that given the complexity of the flow problems encountered in engineering applications, predicting solution behavior *a priori* is very difficult. Understanding the performance characteristics of one problem does not necessarily help in predicting the behavior of another problem. This emphasizes the importance of runtime tuning of algorithmic parameters for improving performance on a particular problem and computational platform.

4.5.3. Dynamic tuning implementation

To allow separate development of application-specific tuning code and to compose a dynamically tuned application with a given program, *tuning control points* need to be identified in the original code, such that the execution control is intercepted and transferred to the tuning module. Within the whole program structure, these control points are the places in which the application performance is regularly measured and the considered algorithmic parameters are updated by tuning operations. As in the previous adaptation examples, we choose as the tuning control point the stability check function, `calc_cfl()`, at the end of the time integration loop, which calculates CFL numbers at every preset number of iterations. In this way, tuning operations can be synchronously performed at regular intervals, while mitigating the potential performance slowdown caused from tuning efforts by reusing existing synchronization barriers that occur near the end of the outer (time integration) simulation loop.

4.5.4. 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 step is measured, with the algorithmic parameter of interest set to some initial value. A new value of that parameter is then selected and the same computational step is timed again. Performance of the computation under the two values of the parameter can then be compared, so that the relative improvement or degradation in performance, as a function of the algorithmic parameter, can be estimated. However, this simple scheme requires that the computational step being compared remains the same, i.e., that the only reason for a change in performance from the first measurement point to the second is the change in the parameter. This is often an unrealistic assumption, especially for scientific simulations, where the properties of the underlying problem (e.g., turbulence in CFD simulations) can change in different phases of the computation, resulting in changes in performance behavior even for the same algorithm parameter values.

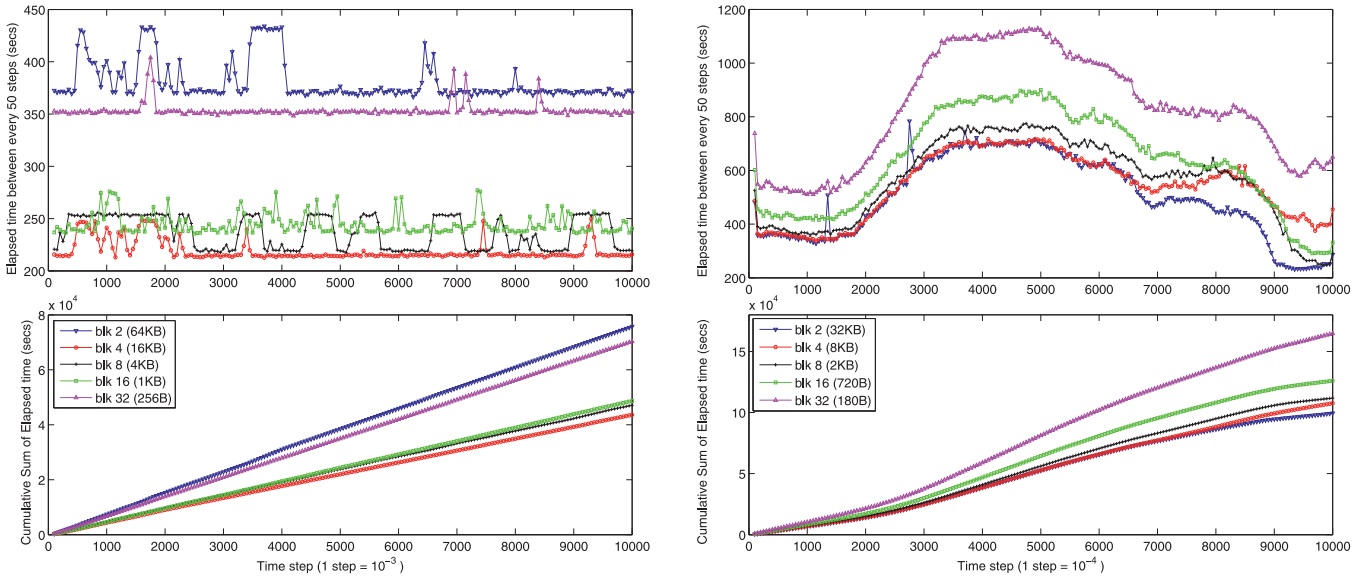
To better handle the case were performance may vary due to a variety of reasons, we estimate the dynamic performance trends by examining the current performance behavior with two execution intervals, so that the slope of the change in performance can be obtained. This gives some indication whether the computation—with the current algorithm parameter value—is becoming easier or more difficult. This slope information is then used in the search heuristic, as in the threshold function in empirical optimization [28], along with the execution time with a new parameter value, to decide whether or not to switch to the new value.

Fig. 9 shows our dynamic tuning procedure. For each algorithmic parameter under tuning consideration, exploring a point in the search space takes four stages: stage 1 initiates measuring performance with the current value, stage 2 completes the first measurement and begins one more examination to measure the slope in performance behavior, stage 3 completes the second measurement and starts measuring with a newly selected parameter value, and stage 4 completes the new measurement and compares performance behavior based on measured history to decide whether the new parameter value is better than the current one. For stages greater than 4 (line 28), the tuning procedure performs no operations. These empty stages are used to control how frequently the adaptation should be performed. For example, with 10 stages

[2] The measurement code was also separately developed and plugged into the GenIDLEST program through the ACC framework without directly touching the original source code.

Table 3
Algorithmic parameters of GenIDLEST for dynamic tuning.

Parameter	Algorithm	Variable name	Dimension
# Sub-blocks	Domain decomposition in preconditioning	`ni_blk,nj_blk,nk_blk`	3D
# Relaxation sweeps	Smoothing	`nswp_in_blk`	1D



(a) Straight channel problem on Anantham (b) Pin fin array problem on System G

Fig. 8. Performance behavior of GenIDLEST with varying sub-block parameter values on different platforms and with different problems.

```
1  Parameters p_1, p_2, ..., p_i, ..., p_k
2  if p_1, ..., p_{i-1} are marked tuned and p_i is not then
3      switch stage do
4          case 1
5              elapsed_time ← 0;
6              start measuring elapsed time with the current value of p_i;
7              break;
8          case 2
9              elapsed1 ← elapsed_time, elapsed_time ← 0;
10             start measuring elapsed time with the current value of p_i;
11             break;
12         case 3
13             elapsed2 ← elapsed_time, elapsed_time ← 0;
14             choose a new value for p_i in the search space;
15             broadcast the new value of p_i from root to all processes;
16             change p_i with the received value;
17             start measuring elapsed time with the new value of p_i;
18             break;
19         case 4
20             elapsed_new ← elapsed_time;
21             compare performance of the new and the old values of p_i;
22             determine either the old or the new value as the current optimum;
23             broadcast the selected optimum of p_i from root to all processes;
24             change p_i with the received value;
25             update the search space of p_i;
26             if the search space of p_i is empty then mark p_i as tuned;
27             break;
28         otherwise break;
29     endsw
30 end
```

Fig. 9. Dynamic parameter tuning procedure.

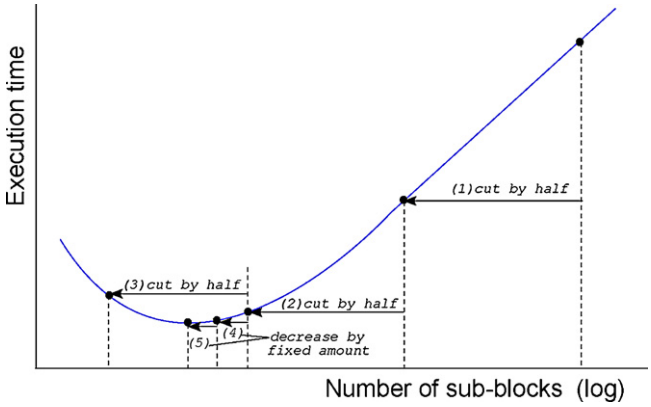


Fig. 10. Coarse search process for GenIDLEST sub-block parameter tuning.

the tuning procedure just breaks and returns to the main simulation code for stages 5–10, essentially pausing the tuning process for several steps of the main simulation. The empty stages are useful for tuning parameters whose changes in value can cause temporary fluctuations in physics simulations (e.g., flow model parameters in CFD simulations [20]), in which some time is required for the simulation to stabilize in order to measure consistent performance behavior.

- *Tuning of the number of sub-blocks parameter*. It is important to balance the trade-off between tuning cost and program performance: spending too much time on an exhaustive search of parameter space for optimal tuning points can adversely affect the overall time-to-solution of the simulation, whereas searching only a small part of the 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 cache sub-block parameter is 3-dimensional, and can grow quite large with only a small increase in problem size, exhaustive search where every combination of (x, y, z) values is examined is not feasible. Therefore, it is critical to plan an effective way of reducing the search space to the extent that the performance benefit from tuning is not significantly degraded. To this end, 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 GenIDLEST performance.

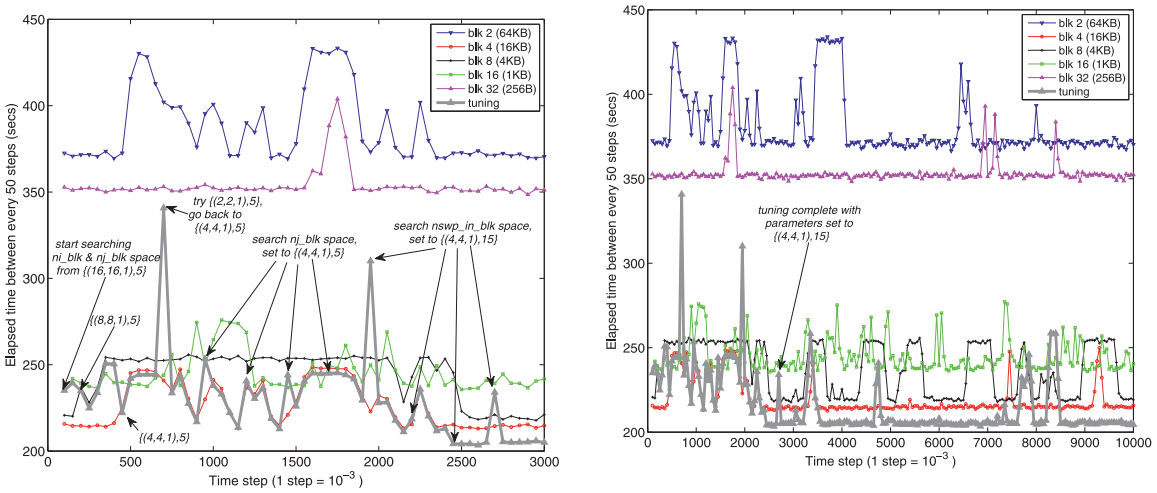
  Secondly, to further reduce the search space of the remaining (ni_blk,nj_blk) pair, we use a two-phase tuning scheme that performs *coarse search* in the first phase and *fine search* in the second phase. In the coarse search phase, the 2-dimensional pair is treated like a 1-dimensional variable by changing both ni_blk and nj_blk together by the same amount. In addition, as illustrated in Fig. 10, the coarse phase uses binary search where both ni_blk and nj_blk are decreased by half in each exploration step that examines a new parameter value for optimality. 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 (the sub-block size becomes bigger) in the coarse phase, starting from a large initial values for the pair (step (1)), which will exhibit an initial period of improved performance in the search. Once a point is reached where performance starts deteriorating (step (3)), 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 (steps (4) and (5)), which tries 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.
- *Tuning of 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, which is directly dependent on the input problem size, the possible number of smoothing iterations on each sub-block is not determined by the problem size; this 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 take 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 $\{1, 5, 10, 15, 20\}$.

When performing dynamic tuning, the tuning order of parameters is important because optimizing operations are performed in conjunction with the computation itself. The order of parameters under tuning directly affects application performance: parameters that are more critical to performance and can show bigger speedups by tuning should be optimized earlier than less critical ones, so that a running computation can benefit earlier from the larger tuning effect, leading to better time-to-solution. Therefore, we first tune the number of sub-blocks parameter and then the number of inner sweeps parameter, since the blocking parameter determines GenIDLEST performance more substantially.

4.5.5. Experimental results

We plugged in the implemented tuning code through ACC and performed GenIDLEST simulations on both Anantham and System G. The GenIDLEST code checks the simulation stability by calling `calc_cfl()` to calculate global CFL values at every 50 time steps, which therefore is the periodic interval the tuning code is executed. Specifically, we set up a 5-stage tuning procedure for exploring a search point, so that the tuning procedure for each searched point takes 250 time steps.

Fig. 11(a) shows the dynamic tuning progress of GenIDLEST for the straight channel problem on Anantham for the first 3000 time steps. The thick gray solid line represents the elapsed time measured at every 50 steps during the tuning process. Data measured for different fixed parameter configurations with the original (unadapted) GenIDLEST program, as previously shown in Fig. 8(a), are also shown 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 the results for other fixed sub-block parameter values in the figure, the tuned results closely follow 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. Fig. 11(a) shows measurements of the elapsed times for the full 10,000 steps, where, after tuning is complete at time step 2700, the tuned simulation shows better performance than other simulations with fixed parameter values.



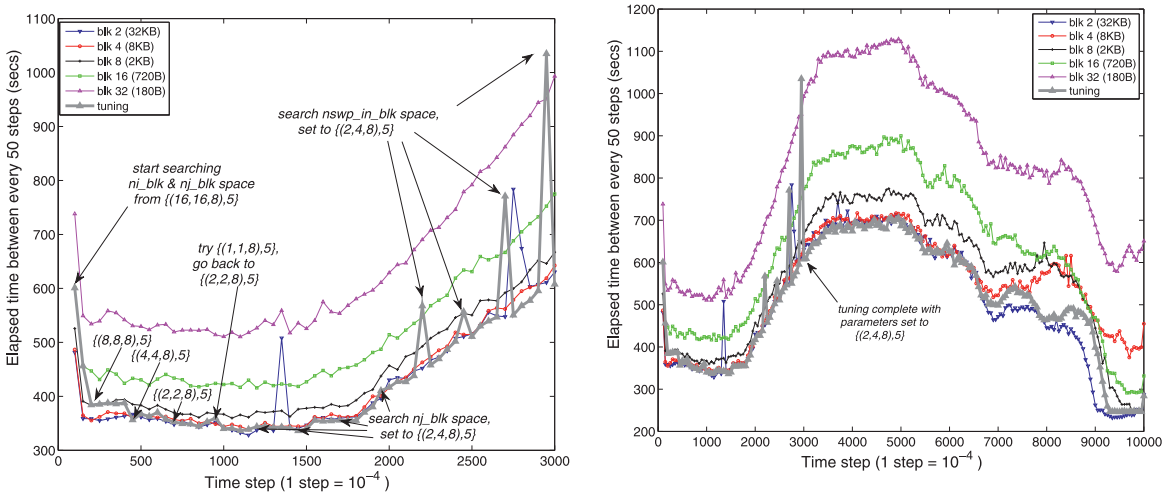
(a) Tuning progress until 3000 time steps (b) Execution profile for 10000 time steps

Fig. 11. Dynamic tuning for the straight channel problem on Anantham.

Fig. 12 shows the dynamic tuning progress of the GenIDLEST simulation for the pin fin array problem on System G, both for the first 3000 time steps (12(a)) and for the full 10,000 time steps (12(b)). Here, tuning of the sub-block parameter completes with the value $(2,4,8)$ at time step 1950, followed by tuning of the inner relaxation sweep parameter that completes with the value 5 at time step 2950. Fig. 12(b) shows that the tuned simulation performs slightly better than *blk2* until near the time step 7000, when it starts to perform a little worse than *blk2* (but still better than the others). This is due to changes in the physics of the simulated problem, which develops in such a way that a parameter configuration that produces good performance initially might not remain optimal through the whole simulation. Our current approach is to minimize overhead by tuning algorithmic parameters only at the beginning of a simulation, and then sticking with those values. Obviously, an enhancement for long-running simulations—where the characteristics of the computation may change over time—would be to redo the parameter tuning at some point, perhaps triggered by an occasional lightweight exploration of other parameter values.

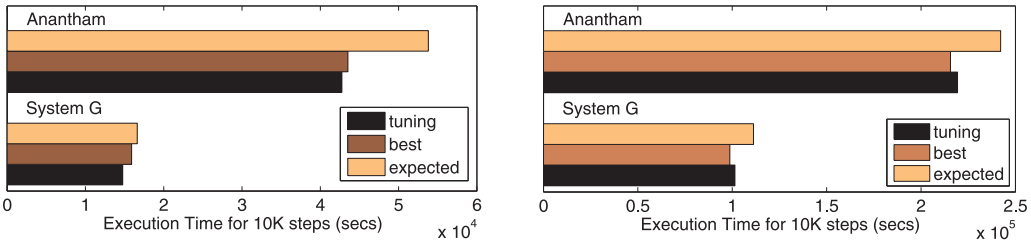
For completeness, we note that the dynamically tuned GenIDLEST simulation for the turbulent straight channel problem on System G completed tuning with the parameters {(ni_blk,nj_blk,nk_blk),nswp_in_blk} set to $\{(2,8,1),10\}$ (compare with $\{(4,4,1),15\}$ on Anantham). The tuned simulation of the pin fin array problem on Anatham completed tuning with {(ni_blk,nj_blk,nk_blk),nswp_in_blk} set to $\{(4,2,8),5\}$ (compare with $\{(2,4,8),5\}$ on System G). This well illustrates the fact that the best choice of algorithmic parameters can depend heavily on the computational platform.

To evaluate the overall improvement in time-to-solution from dynamic parameter tuning, we also measured the performance of GenIDLEST on the same 10,000 time-step simulations, with four different fixed parameter sets (without tuning), corresponding to the configurations *blk2*, *blk4*, *blk8* and *blk16* defined in Section 4.5.2. The default value of nswp_in_blk= 5 is used for these cases. As well as the total execution time for the 10000 steps, elapsed times between every 50 steps were also measured for each simulation. Fig. 13 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; 'best' represents an upper bound on performance, i.e., the hypothetically best time that could be achieved by correctly choosing the best of the four fixed configurations (from *blk2* to *blk16*) at every 50 steps; and the 'expected' time represents an expected



(a) Tuning progress until 3000 time Steps (b) Execution profile for 10000 time steps

Fig. 12. Dynamic tuning for the pin fin array problem on System G.



(a) Straight channel problem (b) Pin fin array problem

Fig. 13. Performance comparison of the dynamically tuned GenIDLEST with the original GenIDLEST.

value for the total execution time, computed by simply averaging the times for the four fixed configurations.

For the straight channel problem, our the dynamically tuned method performed better than 'expected' by 26% and slightly better than the 'best' case by 2% on Anantham; it outperformed both 'expected' (by 13%) and 'best' (by 8%) on System G. The primary reason that the dynamically tuned results are better than what we label as 'best' here is that the number of relaxation sweeps nswp_in_blk is set at 5 for all the fixed parameter cases, while the tuned case is allowed to vary this parameter. For the pin fin problem, on both platforms our method performed better than 'expected' by about 10%, and came within 2% of matching the 'best' performance.

4.6. Adaptation overhead

The runtime overhead of our adaptation method comes from catching the function calls at adaptive control points, which in itself does not involve any global operations that cause communication overhead. The catching overhead is measured at 0.10 μs per call on average on the Anantham cluster, which translates to 140 CPU cycles. Since the catching cost is fixed, the relative overhead depends on the number of interceptions and the entire execution profile of an application. That is, the overhead increases as the number of adaptive control points increases. Still, the catching cost is relatively insignificant if the application spends most of its time on executing other parts of the computation than at control points.

In the adaptive GenIDLEST simulations, control points are intercepted only once at the end of every preset number of iterations of the time integration loop, while most of the computing time is spent inside the loop. As a result, the catching overhead is negligible compared to the whole simulation profile. For example, Fig. 14 shows execution time of the GenIDLEST simulations where the ACC framework is imposed at control points in the time step change and in the flow model change scenario, respectively, but with no adaptation operations. Across the three configurations (4, 8 and 16 processors) the costs for catching 500 calls of `mpi_allreduce` during 500 time steps in the time step change example were measured to be less than 0.7% in all cases compared to the original GenIDLEST simulations (Fig. 14(a)). Similarly, the overhead is less than 0.95% for catching 1000 calls of `calc_cfl` during 1000 time steps in the flow model change example (Fig. 14(b)).

The overhead for the dynamic tuning application includes three components: synchronization, the cost of the dynamic parameter tuning algorithm (Fig. 9), and the extra cost incurred when poor parameter values are used during the exploration of the parameter space. Synchronization overhead is negligible because we do the extra adaptation work at a point where the code already does synchronization. The cost of the tuning algorithm itself is also negligible. The third component of the overhead could be significant if performance is highly sensitive to small changes in parameter values However, for the examples considered here, this overhead was low. We estimated this component of the overhead by totaling the extra time spent doing a 50-timestep interval with a "bad" parameter configurations, i.e., the difference between the time taken when the parameters were poorly chosen and the time taken if we had left the parameters alone. A "good" change in parameters, where performance is improved, is not added into this total. Comparing this total penalty to the total running time of the simulation, the overheads ranged from 0.4% (pin fin array problem on Anantham) to 0.9% (pin fin array problem on System G).

5. Case study 2: cell biology simulations

Adapting an application's runtime behavior requires specifying adaptation schemes, which can be categorized into *precisely defined schemes* and *loosely defined schemes*. Precisely defined schemes include a complete description, *before* application launch, of "where" (adaptation control points), "when" (conditions to trigger adaptation actions), "what" (adaptation targets such as variables or functions), and "how" (operations to execute) to change with regard to runtime program behavior. In contrast, loosely defined schemes consider highly dynamic adaptation scenarios in order to support the user's dynamic and unplanned adaptation decisions. These schemes lack complete adaptation specifications: some part of the adaptations are not specified clearly before application launch and adaptive decisions are deferred until at runtime. Adaptation decisions in such cases can only be made by monitoring the application's dynamic progress, since, until runtime, the user may have only a vague idea of how the program will behave on a given problem instance and computational platform.

Implementing loosely defined schemes requires a high degree of execution control of an application due to the fact that adaptation operations are determined by the user's dynamic decisions. They require a running application to stop at the user's discretion, update its system state with user input, continue its execution followed by additional stops for possible future adaptations. Furthermore, these schemes often require an application to record and restore its state so that its computation can be resumed at an execution point that the user thinks is interesting for performing experiments or simulations. Therefore, enabling loosely defined adaptations in a compositional framework becomes complex. This is in contrast to implementing precisely defined schemes, where adaptation operations are automatically initiated by writing adaptive logic operations in separate modules and by plugging them into an application.

In this section, we apply our adaptation approach to cell biology simulations to implement loosely defined adaptation schemes. Instead of executing automated adaptation operations, the main functionality of the inserted adaptation code is to serve as a gateway to control the execution of an application. By using our approach, domain experts in scientific simulations can initiate dynamic adaptation decisions and perform "what-if" scenarios at runtime, so that they can better mimic what experimentalists do in a physical environment.

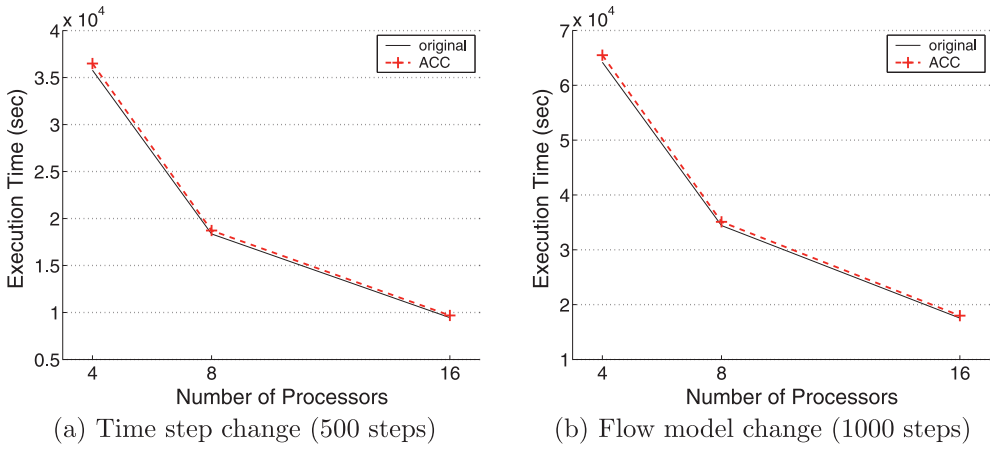


Fig. 14. ACC overhead with GenIDLEST simulations.

5.1. Dynamic adaptation for flexible cell cycle simulations

In search of better models or in exploration of new models, cell cycle modelers change the configuration of a given simulation to examine how the original system is perturbed to evolve into a new state. The course of action using conventional modeling tools that lack dynamic program adaptation functionalities is: (1) stop the simulation and save the state, (2) update the mathematical model (i.e., system of ordinary differential equations (ODEs)) to reflect the perturbation by changing model parameter values, (3) set the initial conditions of the ODEs with the saved state, and (4) restart the simulation. Repeating this procedure whenever configuration changes are desired is cumbersome for cell cycle modelers. Therefore, the ability to adapt simulations at runtime helps facilitate effective cell cycle modeling.

Adapting a simulation at runtime requires a dynamic change in the original simulation model quantified by a set of ODEs. In particular, dynamic changes required for effecting perturbations can be loosely defined. For example, specific reactions to change in a simulated system can remain unspecified before simulation launch and be up to the modeler's runtime decisions.

In the following, we describe the benefits of dynamic adaptation in performing cell cycle simulations.

5.1.1. Effective model parameter estimation

In a mathematical model that describes a biological reaction system, each reaction is modeled as an ODE with parameters that represent reaction rates. Parameter estimation plays a crucial role in modeling biological systems based on real experimental data. Dynamically adapting scientific software through loosely defined schemes allows for increased execution control and interactive feedback to a running computation, so that, based on runtime analysis of computational progress, the user can make adaptive decisions and manipulate ODE parameters immediately through feedback controls without stopping a running simulation. The user then can check to see if the changes in model parameters make the simulation better match experimental data. Thus, support of loosely defined schemes in biological simulations expedites effective model parameter estimation.

5.1.2. Exploration of new evolution pathways

In addition to facilitating effective parameter estimation, adapting cell cycle simulations at runtime allows the user to explore new evolution pathways by manipulating the model parameters of a running simulation. Without having to stop a running computation, the user can dynamically simulate a new *protocol*—a sequence of operations performed in real experiments to develop a certain environment to suppress or catalyze specific biochemical reactions, such as raising the temperature or adding new materials to a system—to investigate how the simulated system evolves from a current condition. Hence, dynamic adaptation capabilities here can effectively facilitate new scientific discoveries.

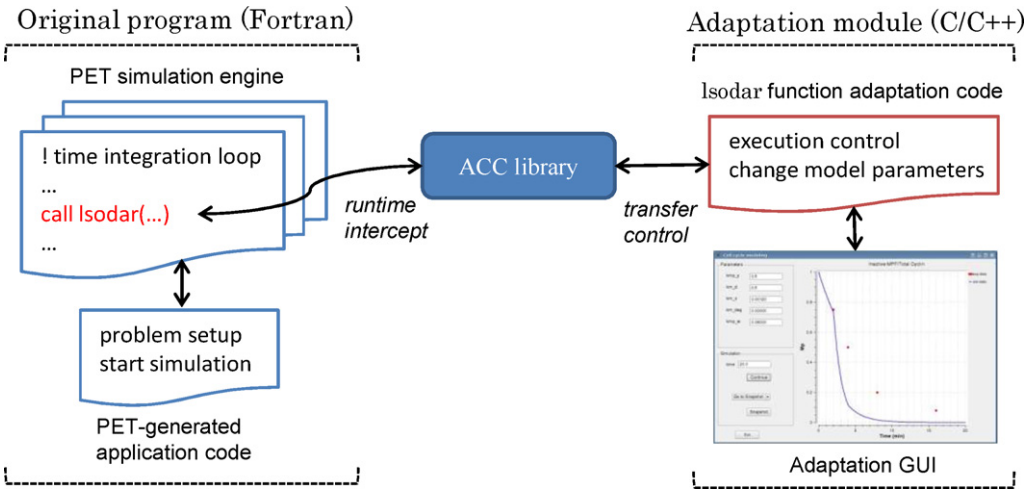
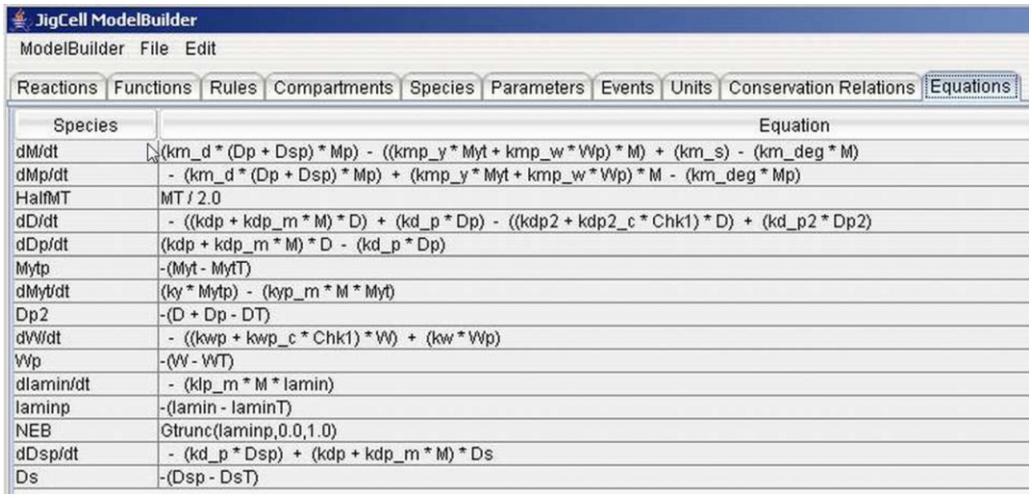


Fig. 15. Dynamic adaptation implementation of PET cell cycle simulations using ACC.



Species	Equation
dM/dt	(km_d * (Dp + Dsp) * Mp) - ((kmp_y * Myt + kmp_w * Wp) * M) + (km_s) - (km_deg * M)
dMp/dt	- (km_d * (Dp + Dsp) * Mp) + (kmp_y * Myt + kmp_w * Wp) * M - (km_deg * Mp)
HalfMT	MT / 2.0
dD/dt	- ((kdp + kdp_m * M) * D) + (kd_p * Dp) - ((kdp2 + kdp2_c * Chk1) * D) + (kd_p2 * Dp2)
dDp/dt	(kdp + kdp_m * M) * D - (kd_p * Dp)
Mytp	-(Myt - MytT)
dMyt/dt	(ky * Mytp) - (kyp_m * M * Myt)
Dp2	-(D + Dp - DT)
dW/dt	- ((kwp + kwp_c * Chk1) * W) + (kw * Wp)
Wp	-(W - WT)
dlamin/dt	- (klp_m * M * lamin)
laminp	-(lamin - laminT)
NEB	Gtrunc(laminp,0.0,1.0)
dDsp/dt	- (kd_p * Dsp) + (kdp + kdp_m * M) * Ds
Ds	-(Dsp - DsT)

Fig. 16. ODE system model for the cell cycles of frog-egg extracts.

5.1.3. Effective simulation through checkpointing and rollback

Most conventional simulation tools are static in the sense that a simulation cannot be rolled back to past states once it is complete. As a result, the user has to finish a simulation and record its results whenever he or she finds the results interesting for future uses, which becomes time-consuming and error-prone as the simulated problem becomes complex and the number of simulations involved gets large. Through the increased execution control offered by dynamic adaptation capabilities, a running computation can be adapted to pause and save its entire state on-the-fly and then resume its execution. By saving and restoring intermediate results and simulation configurations, the user can easily reuse past simulations of interest in future simulations, thus being able to shorten the time-to-solution.

5.2. Implementation

We apply the ACC framework to the PET (Parameter Estimation Toolkit)[3] software of the JigCell project [2,14,31] to implement dynamic adaptations. Specifically, we implement adaptations over the Fortran code generated by PET in the simulation phase, for which PET uses the `lsodar` solver in ODEPACK [16], a collection of Fortran solvers for ODE systems.

5.2.1. Adaptation control

Since the end of a simulation loop typically exhibits stable system state with consistent intermediate results, we specify the entry of the `lsodar` function as an adaptation control point to control the execution of the PET simulation. Thus, the `lsodar` calls in the PET source file are intercepted by the ACC framework. We implement an adaptation scheme in a function `lsodar_handler`, to which the intercepted `lsodar` calls are redirected by ACC.

The `lsodar_handler` function checks the time step at each iteration of the simulation loop and compares it with a stop time specified by the user through an adaptation control GUI, so that the simulation is paused when the stop time is reached. While the application is stopped, the user can analyze the simulation progress, make adaptive decisions, and adapt the simulation by changing the values of the model parameters of interest.

To access the model parameters and variables of a simulated system, we reuse a set of `set/get` helper functions generated by PET, i.e., `get/set_model_param` and `get/set_model_var`.

5.2.2. GUI for runtime adaptation

To better support cell cycle modelers in performing dynamic adaptations, we prototyped a GUI whereby the modeler can control the execution of running simulations and change parameter values of a given model dynamically. We implement the GUI using `Qt`,[4] a cross-platform UI development framework. The GUI runs in a separate thread from the main simulation engine using the `Pthread` APIs [18] in order to control the simulation execution separately from the GUI execution.

Fig. 15 illustrates how we dynamically adapt PET cell cycle simulations through the ACC framework. A simulation program originally consists of the PET simulation engine and a PET-generated simulation driver, each written in Fortran. The adaptation program module includes the `lsodar_handler` code written in C and the Qt GUI in C++. The language-neutral ACC framework combines the simulation program and the adaptation module written in different languages, composing a highly flexible simulator that realizes dynamic, user-driven adaptation schemes.

5.3. Experimental results

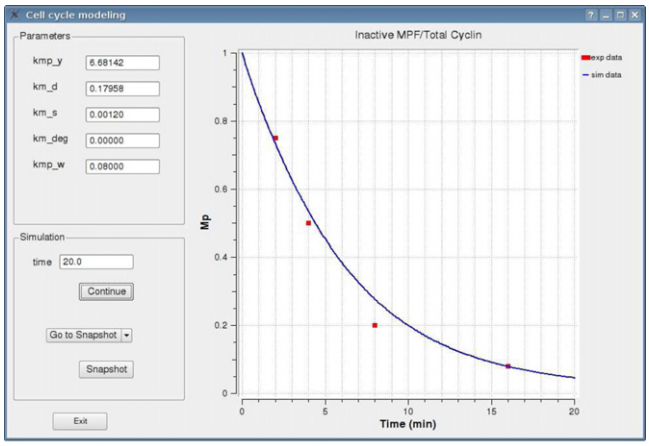
As an example simulation of cell cycle models, we use the frog-egg extracts simulation [37]. Specifically, we apply the adaptation code and the GUI to the PET-generated Fortran code for simulating dephosphorylation of pre-MPF (mitosis-promoting factor) during mitosis, when *Cdc25* is active [24]. The biological system is modeled as a system of 15 ODEs with 28 parameters and 15 variables, as shown in Fig. 16.

Among the reaction equations, we are interested in the second one, which gives the rate of change in the MPF concentration.

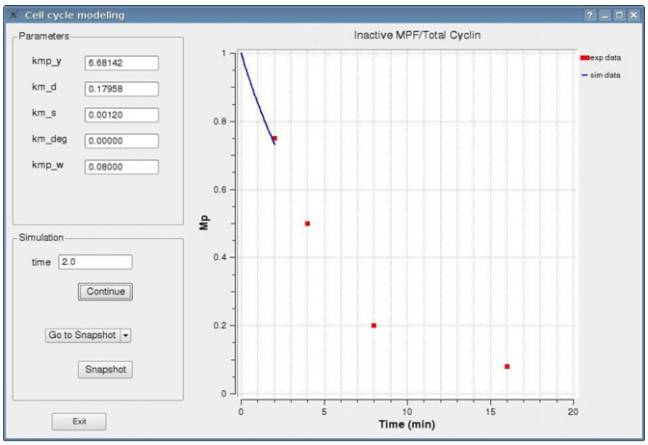
Fig. 17 shows the GUI with the initial configuration for simulating the cell cycle of frog-egg extracts. The 4 red squares in the plotting area represents the actual MPF concentration data obtained in "wet" lab experiments for reference: 0.75 at 2 min, 0.5 at 2, 0.2 at 8, and 0.08 at 16. The upper left `Parameters` pane shows 5 parameters and corresponding input area where the user can change each parameter value during the simulation. The parameter values are initially set using empirically determined "good" values for this specific simulation (i.e., dephosphorylation of pre-MPF during mitosis). The lower left `Simulation` pane embeds the user controls for the simulation. The user can specify the simulation time in the `time` input area and execute the simulation using

[3] http://mpf.biol.vt.edu/pet.

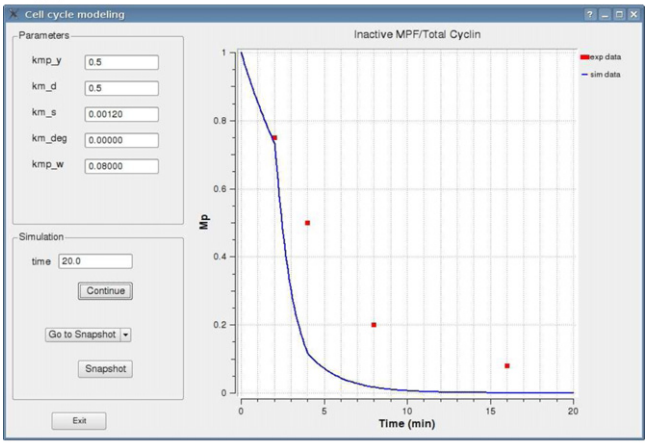
[4] http://qt.nokia.com.



(a) GUI layout and the initial simulation with no adaptation



(a) Simulation until time 2 with initial parameter values



(b) Simulation until time 4 with *kmp_y=0.5* and *km_d=1.0*, and then until time 20 with *kmp_y=0.5* and *km_d=0.5*

Fig. 17. Cell cycle simulation of frog-egg extracts.

Fig. 18. Dynamic parameter change in cell cycle simulations.

the specified parameter values by clicking the `Run` button, which has changed to `Continue` in the figure when the simulation starts execution. Also, the user can save the current simulation state by clicking the `Snapshot` button and restore saved states through the `Go to Snapshot` box.

To show an example use of the GUI, Fig. 17 also plots simulation results until 20 min (blue curve). The simulation uses the initial parameter values, which seem to perform quite effectively because the simulated *Mp* values closely match the real experimental data. We note that this simulation does not involve any adaptations in simulating the model from time 0 to time 20, which is the usual way PET would be used to simulate this model.

5.3.1. Dynamic parameter change

Fig. 18 shows an example adaptive simulation where the model parameter values are dynamically changed by the user. First, the user specifies how long the model should run and what parameter values are to be used, and starts the simulation. Fig. 18(a) plots the simulation results where the simulation end time is set to 2 min and the initial parameter values are used. Then, after observing and analyzing the simulation results, the user decides to change two parameters from their default values: *kmp_y* to 0.5 and *km_d* to 1.0. Wanting to know how the simulated system progresses if the new setting is kept for 2 min, the user sets the simulation time to 4 min and continues the simulation by clicking the `Continue` button. The MPF plot until time 4 in Fig. 18(b) shows how the simulated system evolves with the newly set parameter values, where the MPF concentration drops more rapidly than its previous setting. Finally, after checking the simulation results, the user decides to adjust another parameter, changing *km_d* to 0.5. and simulates the system until 20 min. The MPF plot from time 4 to 20 in the figure shows how the MPF concentration is changed by the new setting of the simulation, where its decrease rate becomes slow after time 4.

As demonstrated by this example, the dynamic adaptation capability implemented in this work allows the user to explore new pathways in evolving biological systems in a flexible way, without requiring stop and restart of an on-going simulation.

5.3.2. Simulation rollback and continuation

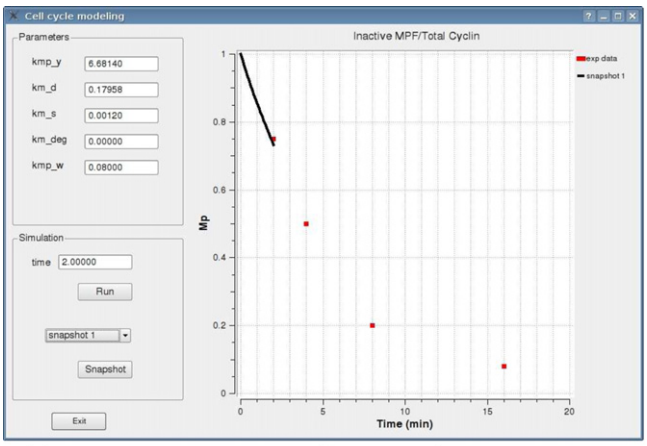
Fig. 19 shows an example of how a simulation can rollback and be continued by our dynamic adaptation framework. First, by clicking the `Snapshot` button, the user saves a simulation state which may be interesting to reuse in future experiments. A saved state contains all the information needed to resume a simulation starting from the saved point, including all the model parameter values and variable values for the duration of simulated evolution. Later, saved states can be restored by choosing a specific state in the `Go to Snapshot` box. For example, Fig. 19(a) shows a restored state (`snapshot 1`) from Fig. 18(a) in the previous parameter change adaptation example. Using the saved results as a new starting point, the user then tweaks two parameters, *kmp_y* and *km_d*, by setting their values to 0.5 and 0.5, respectively, and continues the simulation until 10 min. The simulation results are shown in Fig. 19(b), where the MPF concentration after time 2 until 10 decreases more rapidly by the changes in the reaction rates. At time 10, the user can explore new pathways in evolving the system. As shown in the figure, the user changes *kmp_y* and *km_d* again to 1.0 and 1.0, respectively, and continues the simulation until time 20. Affected by this reaction rate change, the MPF plot in the figure shows more rapid decrease in its concentration after the parameter change.

6. Related work

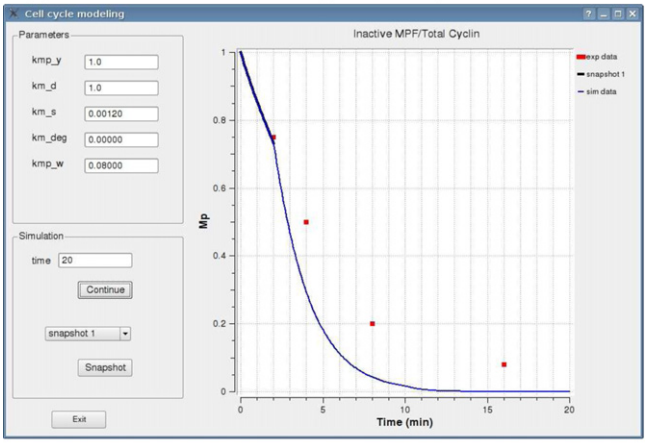
Adaptation support for existing programs has been of much interest at different levels of the software stack in many computer science domains. We describe some of the projects most relevant to our work and contrast them in this section.

6.1. Language and compiler support for adaptation

Program Control Language (PCL) [11] is a small language extension that provides mechanisms that one can use to write separate control code that specifies application-specific adaptation



(a) Restored state at time 2 (Figure 18a) through rollback



(b) Simulation until time 10 with *kmp_y=0.5* and *km_d=0.5* from the restored state, and then until time 20 with *kmp_y=1.0* and *km_d=1.0*

Fig. 19. Runtime simulation rollback and continuation in cell cycle simulations.

strategies at a high level for distributed programs. The PCL framework uses a global representation of the distributed program as a graph of task nodes. Each adaptation primitive of PCL maps to a sequence of graph-changing operations of the target program. Du and Agrawal [10] proposed a Java language extension to help programmers specify adaptation parameters, which exposes the degree of flexibility in the quality of the output, for adaptive program execution. By combining runtime information with a static analysis for relating the execution time to the values of the adaptation parameters, a set of optimal parameter values is determined by initial test runs to achieve the best precision while meeting the specified constraints on execution time. ADAPT [32] is an adaptive optimization system that provides a domain-specific language by which users can describe optimization heuristics to be applied at runtime. Based on user-supplied heuristics (e.g., loop unrolling and specifying machine parameters), the ADAPT compiler generates a runtime system that consists of a modified version of the application, which in turn contains two different execution paths for each code section that is a candidate for optimizations.

These projects share a common theme with our work in that centralized design of adaptation strategies can be specified at a high level for existing programs. However, our work provides more fine-grained control by supporting monitoring and manipulating of state variables internal to a program. In addition, the ACC framework supports language-independent composition, whereby individual code modules written in different languages can be seamlessly combined. Thus, by using our approach, scientific programs written in Fortran that have been traditionally hard to change can be effectively adapted.

6.2. Dynamic instrumentation

Dynamic binary instrumentation tools offer a modular, language-independent way of code modification, so that new adaptation modules can be transparently combined with existing software. Detours [17] is a library for instrumenting arbitrary Win32 binary functions (32-bit API for Windows systems), so that th calls to target functions can be intercepted and detoured to the user-provided detour function, which can replace or extend the target function. For example, if a programmer is writing a program that uses a Win32 function and want to intercept it, he can include the Detours library in the program and use its APIs to intercept the Win32 functions. DynInst [8] offers an API for inserting code to a running program on both Unix and Windows systems. Code modification places are usually the entry or exit (or both) of a function of interest. To modify a program in execution (referred to as mutatee), a separate program (mutator), where the programmer uses the DynInst API to instrument mutatee and perform mutating operations, is executed with the information of mutatee (e.g., the process id of the mutatee). Functions in dynamically loaded modules in an application can be effectively handled since DynInst allows code modification at any time during program execution. Since the accompanying overhead is significant while they perform code instrumentation at program runtime, they are usually developed for sophisticated programs analysis purposes [27] rather than as a tool to realize program behavior adaptation.

6.3. Middleware support for adaptation

In parallel or distributed environments such as the Grid, there is a large body of research work on middleware to support runtime adaptations without recompiling or rerunning of an application. We contrast notable projects to our work in the following.

Buaklee et al. [7] develop an accurate performance model from a detailed analysis of application execution times with varying configuration parameters, such that optimal configuration parameters for the distribution of work can be determined without requiring any user-supplied input. The GrADS (Grid Application Development Software) [22] project proposes a *configurable object program* that encapsulates, in addition to the application code, dynamic adaptation strategies to effectively map and schedule Grid resources, for which resource selection and accurate performance models are provided by the GrADS execution framework. CACTUS-G [1] implements dynamic adaptive techniques for efficient execution of astrophysics simulations in distributed, heterogeneous Grid environments. The focus is on automatically adjusting external operating parameters such as communication message sizes and ghost zone sizes in the grid. The AppLeS project [5] provides a methodology and software environments for on-the-fly adaptive application scheduling in Grid environments. It has been applied to a variety of domains to result in new applications, each of which consists of domain-specific components and custom scheduling superstructure that is controlled by the AppLeS scheduling agent to monitor available resource performance and generate a dynamic schedule on the target Grid platforms.

As their objective is to implement middleware support for adaptation between the application and the underlying execution layer, these efforts focus on resource management towards efficient utilization of the environment, such as load-balancing and scheduling of application tasks, where coarse-grained strategies based on resource constraints or external operating parameters are employed. In contrast, our work implements a parallel adaptation framework that can adjust fine-grained aspects of program state

and behavior by monitoring dynamic progress of the computation itself.

6.4. Computational steering

Computational steering [25,26] enables dynamic adaptation of scientific simulations through interactive user-feedback on changing parameter values of a given simulated system. Computational steering is useful for runtime performance tuning and "what-if" studies because it does not require stop-and-restart of computation all over again. However, sophisticated data visualization is crucial to help guide the user in analyzing the progress of a running simulation, which often takes significant programming effort to implement. Furthermore, implementing efficient middleware or runtime support to instantly update the program states at the computation backend with the user's feedback at the steering fronted is a major challenge in large, distributed systems such as the Grid—a major target execution environment of computational steering applications—where communication between remote sites is a performance bottleneck [35].

In contrast to computational steering that focuses on user-driven parameter change, our work implements software adaptation in general, where program behavior is adapted through changing not only program states but also program modules such as through function call substitution.

7. Conclusion

In this paper, we presented a modular approach to implementing adaptations for existing scientific programs. Our compositional approach addresses adaptation as a separate concern in software development and allows one to separately design and implement adaptation strategies for scientific programs. We use a language-neutral, compositional framework based on function call interception to transparently insert adaptation code at user-specified control points within a program, thus adapting application behavior at runtime. We showcased the applicability of our approach to real-world scientific programs. By using our approach, we enhanced the original capabilities of a parallel CFD simulation program with regard to stability, accuracy, and performance, with negligible adaptation overhead. We also implemented flexible cell cycle simulations whose execution control is dynamically controlled by user-driven decisions, thus facilitating effective examination of biochemical reactions. Overall, the benefits of our adaptation approach include high modularity, fine-grained adaptation control, and language-independence. Our approach is well suited to factoring out new functionality and integrating it with existing scientific programs to implement dynamic application-specific adaptation scenarios.

References

[1] G. Allen, T. Dramlitsch, I. Foster, N.T. Karonis, M. Ripeanu, E. Seidel, B. Toonen, Supporting efficient execution in heterogeneous distributed computing environments with cactus and globus, in: Supercomputing '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM), ACM, New York, NY, USA, 2001, p. 52.

[2] N.A. Allen, C.A. Shaffer, N. Ramakrishnan, M.T. Vass, L.T. Watson, Improving the development process for eukaryotic cell cycle models with a modeling support environment, Simulation 79 (2003) 674–688.

[3] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N.M. Amato, L. Rauchwerger, STAPL: an adaptive generic parallel C++library, in: H.G. Dietz (Ed.), LCPC, Lecture Notes in Computer Science, vol. 2624, Springer, 2001, pp. 193–208.

[4] R. Barrett, M. Berry, J. Dongarra, V. Eijkhout, C. Romine, Algorithmic bombardment for the iterative solution of linear systems: a poly-iterative approach, J. Comput. Appl. Math. 74 (1996) 91–109.

[5] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, D. Zagorodnov, Adaptive computing on the grid using AppLeS, IEEE Trans. Parallel Distrib Syst. 14 (2003) 369–382.

[6] S. Bhowmick, L.C. McInnes, B. Norris, P. Raghavan, The role of multi-method linear solvers in PDE-based simulations, in: V. Kumar, M.L. Gavrilova, C.J.K. Tan, P. L'Ecuyer (Eds.), Computational Science and its Applications – ICCSA, Part I, vol. 2667, Springer, 2003, pp. 828–839.

[7] D. Buaklee, G.F. Tracy, M.K. Vernon, S.J. Wright, Near-optimal adaptive control of a large grid application, in: ICS '02: Proceedings of the 16th International Conference on Supercomputing, ACM, New York, NY, USA, 2002, pp. 315–326.

[8] B. Buck, J.K. Hollingsworth, An API for runtime code patching, Int. J. High Perform. Comput. Appl. 14 (2000) 317–329.

[9] 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.

[10] W. Du, G. Agrawal, Language and compiler support for adaptive applications, in: SC'04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, IEEE Computer Society, Washington, DC, USA, 2004, p. 29.

[11] B. Ensink, J. Stanley, V. Adve, Program control language: a programming language for adaptive distributed applications, J. Parallel Distrib. Comput. 63 (2003) 1082–1104.

[12] V. Estivill-Castro, D. Wood, A survey of adaptive sorting algorithms, ACM Comput. Surv. 24 (1992) 441–476.

[13] M. Germano, U. Piomelli, P. Moin, W.H. Cabot, A dynamic subgrid-scale eddy viscosity model, Phys. Fluids A: Fluid Dyn. 3 (1991) 1760–1765.

[14] L.S. Heath, N. Ramakrishnan, The emerging landscape of bioinformatics software systems, IEEE Comput. 35 (2002) 41–45.

[15] M.A. Heffner, A runtime framework for adaptive compositional modeling, Master's Thesis, Blacksburg, VA, USA, 2004.

[16] A.C. Hindmarsh, ODEPACK, a systematized collection of ODE solvers, in: R.S. Stepleman, et al. (Eds.), IMACS Transactions on Scientific Computation, vol. 1, North-Holland, Amsterdam, 1983, pp. 55–64.

[17] G. Hunt, D. Brubacher, Detours: binary interception of Win32 functions, in: Proceedings of the 3rd USENIX Windows NT Symposium, pp. 135–144.

[18] IEEE and The Open Group, IEEE Standard 1003.1-2001, 2001.

[19] P. Kang, M.A. Heffner, N. Ramakrishnan, C.J. Ribbens, S. Varadarajan, Adaptive Code Collage: a framework to transparently modify scientific codes, IEEE Computing in Science and Engineering 14 (1) (2012) 52–63.

[20] 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.

[21] P. Kang, N.K.C. Selvarasu, N. Ramakrishnan, C.J. Ribbens, D.K. Tafti, S. Varadarajan, Dynamic tuning of algorithmic parameters of parallel scientific codes, in: ICCS '10: Proceedings of the 10th International Conference on Computational Science, pp. 145–153.

[22] K. Kennedy, M. Mazina, J.M. Mellor-Crummey, K.D. Cooper, L. Torczon, F. Berman, A.A. Chien, H. Dail, O. Sievert, D. Angulo, I.T. Foster, R.A. Aydt, D.A. Reed, D. Gannon, S.L. Johnsson, C. Kesselman, J. Dongarra, S.S. Vadhiyar, R. Wolski, Toward a framework for preparing and executing adaptive grid programs, in: IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium, IEEE Computer Society, Washington, DC, USA, 2002, p. 322.

[23] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of the European Conference on Object-Oriented Programming, vol. 1241, Springer-Verlag, Berlin, Heidelberg, and New York, 1997, pp. 220–242.

[24] A. Kumagai, W.G. Dunphy, Control of the Cdc2/cyclin B complex in xenopus egg extracts arrested at a G2/M checkpoint with DNA synthesis inhibitors, Mol. Biol. Cell 6 (1995) 199–213.

[25] 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 (1990) 89–97.

[26] 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.

[27] M. Schulz, D. Ahn, A. Bernat, B.R. de Supinski, S.Y. Ko, G. Lee, B. Rountree, Scalable dynamic binary instrumentation for blue gene/L, SIGARCH Comput. Archit. News 33 (2005) 9–14.

[28] K. Seymour, H. You, J. Dongarra, A comparison of search heuristics for empirical code optimization, in: iWA, 2008: the 3rd International Workshop on Automatic Performance Tuning, 2008, pp. 421–429.

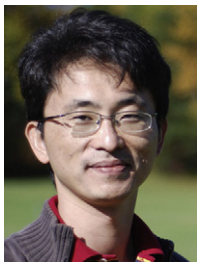
[29] D. Tafti, GenIDLEST – a scalable parallel computational tool for simulating complex turbulent flows, in: Proceedings of the ASME Fluids Engineering Division (FED), vol. 256, ASME-IMECE, 2001, pp. 347–356.

[30] S. Varadarajan, N. Ramakrishnan, Novel runtime systems support for adaptive compositional modeling in PSEs, Future Gener. Comput. Syst. 21 (2005) 878–895.

[31] M.T. Vass, C.A. Shaffer, N. Ramakrishnan, L.T. Watson, J.J. Tyson, The JigCell model builder: a spreadsheet interface for creating biochemical reaction network models, IEEE/ACM Trans. Comput. Biol. Bioinform. 3 (2006) 155–164.

[32] M.J. Voss, R. Eigemann, High-level adaptive program optimization with ADAPT, in: PPoPP '01: Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, ACM, New York, NY, USA, 2001, pp. 93–102.

[33] G. Wang, D.K. Tafti, Performance enhancement on microprocessors with hierarchical memory systems for solving large sparse linear systems, Int. J. High Perform. Comput. Appl. 13 (1999) 63–79.
[34] 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.
[35] H. Wright, R. Crompton, S. Kharche, P. Wenisch, Steering and visualization: enabling technologies for computational science, Future Gener. Comput. Syst. (2008).
[36] H. Yu, D. Zhang, L. Rauchwerger, An adaptive algorithm selection framework, in: PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society, Washington, DC, USA, 2004, pp. 278–289.
[37] J.W. Zwolak, J.J. Tyson, L.T. Watson, Parameter estimation for a mathematical model of the cell cycle in frog eggs, J. Comput. Biol. 12 (2005) 48–63.



Pilsung Kang received a Ph.D. in Computer Science from Virginia Tech. His research interests include computational science, software engineering, parallel programming, and embedded computing. He recently joined Samsung Electronics as a Senior Engineer and now develops embedded software for solid-state drives.

Naresh K.C. Selvarasu joined the Mechanical Engineering PhD program at Virginia Tech in Fall 2007 after completing his masters from Purdue University. He research area is the study of blood flow in coronary arteries, with the object of characterizing the effect of stenting, using Computational Fluid Dynamics, with focus on Fluid–Structure Interaction and Pulsatile Flow.

Naren Ramakrishnan is a professor and the associate head for graduate studies in the Department of Computer Science at Virginia Tech. His research interests are mining scientific data, computational science, and information personalization He received his Ph.D. in computer sciences from Purdue University. He is an ACM Distinguished Scientist and serves on the editorial boards of several journals including IEEE Computer and Data Mining and Knowledge Discovery.

Calvin J. Ribbens is an Associate Professor and Associate Department Head for Undergraduate Studies in the Department of Computer Science at Virginia Tech (Blacksburg, VA, USA). He received a B.S. in Mathematics from Calvin College (1981) and a Ph.D. in Computer Sciences from Purdue University (1986). His research interests include parallel computation, numerical algorithms, mathematical software, and tools and environments for high performance computing.

Dr. Danesh K. Tafti obtained his Ph.D. from the Mechanical Engineering Department at Penn State University in 1989. From 1989 to 1991, he served as a postdoctoral research associate in the Dept. of Mechanical and Industrial Engineering at the University of Illinois at Urbana-Champaign (UIUC). He then joined the National Center for Supercomputing applications (NCSA) at UIUC, where he held positions of Research Scientist, Senior Research Scientist and Associate Director. At NCSA his research focused on developing novel programming paradigms on emerging high performance computing architectures for applications to computational fluid dynamics and turbulent flow simulations. He joined the Mechanical Engineering Department at Virginia Tech in spring 2002 as an Associate Professor where he directs the High Performance Computational Fluid-Thermal Science and Engineering Lab. He was promoted to the rank of Professor in 2008 and currently holds the William S. Cross Professorship in Engineering. His research interests are in high-end, multiscale, multiphysics simulations of single and multiphase systems in the broad areas of propulsion (Solid-rockets, turbines and Micro-Air Vehicles), energy (coal and nuclear systems), and biomedical (cardiovascular) flows. He has over 150 peer reviewed publications to his credit in the areas of fluid mechanics and turbulence, heat transfer, numerical methods and algorithms and high-end computing and has given several invited lectures and keynote presentations at national and international conferences.

Yang Cao received his Ph.D. degree in computer science from the University of California, Santa Barbara in 2003. He is an Assistant Professor in the Computer Science Department at the Virginia Polytechnic Institute and State University. His research focuses on the development of multiscale, multiphysics stochastic modeling and simulation methods and tools that help biologists build, simulate, and analyze complex biological systems. He has published around 40 refereed journal articles.

Srinidhi Varadarajan is the Director of the Center for High-End Computing Systems and an Associate Professor in the Department of Computer Science at Virginia Tech. His research interests are in the area of high-end computing systems, focused more specifically on fault tolerance in large-scale distributed systems, runtime systems, and frameworks for integrated emulation and simulation of computer networks. He received his Ph.D. in Computer Science from Stony Brook University.