Dynamic Algorithm Switching in Parallel Simulations using AOP

PILSUNG KANG
Department of Computer Engineering
Youngsan University
Junam-ro 288, Yangsan, Gyeongnam, 50510, South Korea
E-mail: pilsungk@ysu.ac.kr

We present a modular approach to implementing dynamic algorithm switching for parallel scientific simulations. Our approach leverages modern software engineering techniques to implement fine-grained control of algorithmic behavior in scientific simulations as well as to improve modularity in realizing the algorithm switching functionality onto existing application source code. Through fine-grained control of functional behavior in an application, our approach enables design and implementation of application-specific dynamic algorithm switching scenarios. To ensure modularity, our approach considers dynamic algorithm switching as a separate concern with regard to a given application and encourages separate development and transparent integration of the switching functionality without directly modifying the original application code. By applying and evaluating our approach with a real-world scientific application to switch its simulation algorithms dynamically, we demonstrate the applicability and effectiveness of our approach to constructing efficient parallel simulations.

Keywords: algorithm switching, aspect-oriented programming, parallel programming, program adaptation, scientific computing

1. INTRODUCTION

In scientific simulations on high-performance computing environments, choosing an optimal algorithm for a given problem out of different algorithmic alternatives is the key to accelerating time-to-solution. However, since the characteristic properties of a simulated problem are typically unknown *a priori* and hard to estimate, the conventional course of action for selecting an algorithm has been traditionally relying on complicated heuristics or venturesome speculations based on previous experiences and practices, which may lead to wasted execution time and computing resources due to unsatisfying or sometimes incorrect simulation results.

Adaptive methods and frameworks for automatically selecting optimal algorithms have been proposed [1, 2] to overcome this issue. However, these approaches require explicit programming of algorithm switching operations with given programs, which can be cumbersome and often erroneous for software developers due to the burden of direct modifying application source code. Here, algorithm switching operations denote condition checking for switching decisions and algorithm implementation substitution.

Furthermore, considering that the execution model in the scientific computing domain is becoming increasingly dynamic [3], statically presetting algorithm switching op-

Received ...

erations before simulation launch is ineffective and out-of-place for modern, complex scientific software.

To address these issues, we present a modular, dynamic algorithm switching approach for scientific simulations. Our approach exploits modern software engineering machinery to secure modularity in software development without having to directly modifying the source code. To obtain necessary information about a given problem and to choose the best possible algorithm matching the inherent properties of the problem, we exploit the user's dynamic analysis based on intermediate simulation results during runtime, which is implemented as a part of the dynamic algorithm switching functionality. In order to enable dynamic switching decisions, our algorithm switching implementation pauses the simulation after a preset time to present intermediate state of the simulation, so that the user can examine the state and determine the characteristics of the given simulated system. Once the nature of the problem is analyzed and understood, the user can decide whether to adopt different algorithmic alternatives with the given problem. Our algorithm switching technique is applicable as long as the input and output semantics of a candidate algorithm is same as that of the original one.

There are three major challenges in implementing dynamic algorithm switching.

- **Fine-grained application behavior control**
  In order to change the algorithmic behavior of a running simulation, fine-grained application behavior control is critical. More specifically, program adaptation capability at the level of functions is required because functions are generally considered the most basic element comprising a program module.

- **Modular adaptation of program behavior**
  The algorithm switching functionality with existing application code needs to be implemented such that direct modification of existing code is avoided as much as possible to prevent any possible human errors in restructuring or rewriting a given application. Hence, a modular software development method is needed to extend program behavior with an application-specific adaptation functionality (i.e., algorithm switching functionality), where newly added code is transparently combined with the existing application code base.

- **Applicability of algorithm switching method**
  The methods for implementing the algorithm switching functionality need to be easily adoptable and readily accessible. It is essential that the tools or frameworks used are widely accepted among the programming community, so that the approach is straightforwardly applicable across different computing platforms.

Our approach for dynamic algorithm switching tackles these challenges by means of aspect-oriented programming (AOP). AOP [4, 5] is a programming paradigm which emerged to address the weakness of object-oriented programming (OOP) for *cross-cutting concerns* that cannot be easily decomposed from the rest of the application. AOP enables cross-cutting concerns, called *aspects*, to be centralized in one location, thus encouraging modularity and maintainability in the software design process. Typical aspects in software programs include logging, security, and performance, all of which cannot be easily captured by typical inheritance hierarchy of OOP techniques.

The research work of this paper builds upon our previous efforts [6, 7] where we utilized in-house tools for adapting scientific software. This paper generalizes the approach in scientific programming using popular and modern software engineering techniques provided by AOP to improve modularity and portability in software development.

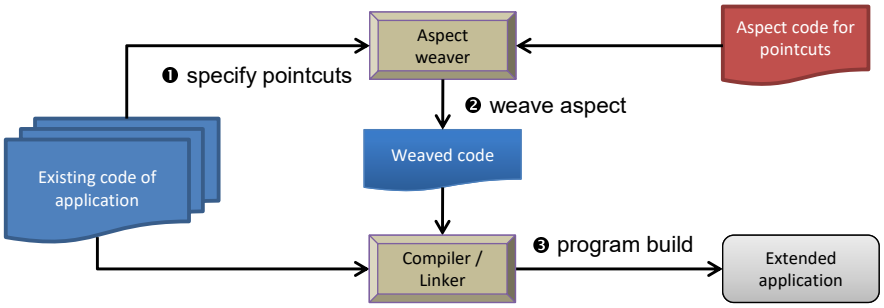

Fig. 1. Application development structure with aspects and weaver in AOP

The remainder of this paper is organized as follows. Section 2 describes the AOP approach for adapting program behavior in the context of parallel simulations. Section 3 introduces scientific simulation algorithms and a dynamic switching scenario as our target application of our approach. Section 4 describes the details of our algorithm switching implementation using the AOP methodology. Section 5 provides experimental results of our approach and evaluates its applicability and effectiveness. Section 6 presents a summary of research related to our work. And finally, Section 7 summarizes our work and concludes the paper.

2. AOP APPROACH FOR PARALLEL PROGRAM ADAPTATION

A major way of implementing a cross-cutting concern in AOP is to use *advice*, which is a piece of code to be performed at intended program execution points specified as a *pointcut*. When a running program reaches a *join point*, the point of execution in a running program, and if the point is at a function or method call, the call is intercepted and the program control is transferred to an associated advice code for execution. This process of instrumenting an advice code into a target program at associated pointcuts is called *advice weaving*, which does not involve directly modifying the original code base. AOP languages offer programming constructs to express pointcuts and advice code, which can be invoked before or after executing a target method. In addition, an advice can entirely replace the original target call. Fig. 1 illustrates a conceptual structure of application development in AOP. A new concern is separately written in aspect code and combined with the original code across pointcuts through the aspect weaver to generate weaved code, which is then compiled to compose an application with extended behavior.

Fig. 2 shows a simple aspect code in AspectC++ [8], a C++ extension of AOP, that illustrates a pointcut and its logging advice code. `Logging` is an aspect class with one advice member function that intercepts the execution of the function named as `checkSystemState` specified with a **`execution`** pointcut expression (line 5). The **`before`** advice is weaved to execute right before at `checkSystemState`, so that the signature of `checkSystemState` is printed out via the AspectC++ API (application programming interface), `JoinPoint::signature` (line 6). Beside augmenting operations at pointcuts, function parameters and return values can be accessed and manipulated through a rich set of AspectC++ APIs. Therefore, AOP is frequently used in modular development of program behavior adaptation at the fine-grained level such as functions, and the appli-

```
/* Logging aspect in AspectC++ for a specific function */
aspect Logging {
public:
  /* advice code to execute before checking system state */
  advice execution(checkSystemState()) : before() {
      std::cout << JoinPoint::signature();
  }
};
```

Fig. 2. Logging aspect implementation in AspectC++

cations include application servers in the enterprise environment [9] and secure software engineering [10].

In addition to the modularity and fine-grained adaptation benefits, using the AOP mechanisms enables to achieve synchronous adaptation with parallel applications. In the SPMD (single program multiple data) programming model used in most parallel applications, all processes executes the same program with different data across multiple connected machines. Specifying a function pointcut in the program and intercepting the program control at the pointcut allows to make concurrent processes execute synchronously because all the processes share the same program. Synchronous adaptation is critical in coordinating parallel processes to preserve stable and consistent program state. Otherwise, race conditions among processes may occur, resulting in inconsistent computations states.

While AOP weaving can be performed statically [11, 12] or dynamically [13, 14, 15, 16], AOP extensions for traditional imperative languages employ static approaches for advice weaving, where source-to-source translation is used to generate standard C/C++ code out of aspects, which can be fed to the usual C/C++ compilers to build a program image. Dynamic weaving is usually offered in modern languages with managed runtime environments such as Java Virtual Machine, and will be discussed in Section 6.

Our approach to implementing dynamic algorithm switching for parallel applications employs AspectC++, a popular C++ AOP extension that generates standard C++ code, so that a wide range of applications on parallel environments can be covered without resorting to custom frameworks or domain-specific languages.

3. SCIENTIFIC SIMULATION OF BIOCHEMICAL SYSTEMS

To show the applicability of our dynamic algorithm switching approach in the context of scientific computing, we choose a parallel stochastic simulation of biochemical reaction systems in the cell biology area. The stochastic simulation algorithm (SSA) is a well-known stochastic method as an exact numerical procedure for simulating well-stirred chemically reacting systems [17, 18]. In this algorithm, the system of N molecular species with population $S_1,...,S_N$ is represented by the state vector $X(t) = (X_1(t),...,X_N(t))$ where $X_i(t)$ denotes the number of species S_i at time t. The species in the system interact through the M reaction channels $R_1,...R_M$ starting from the initial state $X(t_0) = x_0$. Assuming a well-stirred system, the dynamics of each channel R_j is characterized by the state-changing vector $\nu_j = (\nu_{1j},...,\nu_{Nj})$ and a propensity function a_j. The state-changing vector denotes the change in the number of species caused by reactions and specifically, $a_j(x)dt$ is the probability that one R_j reaction event will occur in the next infinitesimal time interval $[t, t+dt)$. In the SSA, the simulation progresses as random samples for j and τ are generated, which causes reaction R_j to fire as the next reaction at $t+\tau$.

In summary, the SSA simulation procedure takes the following steps:

1. Initialization: Initialize the system with the time $t = t_0$ and the state $x = x_0$.
2. Propensity calculation: Calculate the propensity functions a_i for x at t.
3. Reaction selection: Generate the time τ and select the reaction R_j to fire at the next reaction.
4. System update: Update the system state with $t \leftarrow t + \tau$ and $x \leftarrow x + \nu_j$.
5. Termination: Go back to step 2 and repeat if the simulation has not reached the final time, or stop the simulation.

3.1 Slow-scale SSA and Dynamic Algorithm Switching Scenario

The exact SSA algorithm is computationally intensive because it simulates every event specified by the reaction channels of a given system. It does not take the system's characteristic properties into account that may be exploited to boost simulation performance. As an alternative to the exact SSA, the *slow-scale SSA* (ss-SSA) [19, 20] algorithm is an efficient approximation of the exact SSA by utilizing inherent characteristics of a simulated system. In the ss-SSA algorithm, reactions are partitioned into two groups: "fast" and "slow" reactions. Reactions with much larger propensity functions are called fast, and all the other reactions are called slow. In biochemical systems, "fast" reactions do not contribute much to the system's evolution, while "slow" reactions do. By using the fast and slow reaction information, the ss-SSA performs stochastic methods for only slow reactions while the rest of the system is approximated using analytical methods, thereby achieving significant simulation speed-up. Since the ss-SSA is an approximation for the exact SSA, its limitations and implications need to be well understood by the user of our algorithm switching method in the context of stochastic simulation of biochemical reaction systems.

To apply the ss-SSA, the given system needs to be determined if fast reactions exist and such reactions also need to be identified. To demonstrate the effectiveness of our approach, we implement algorithm switching in stochastic simulations from the exact SSA to the ss-SSA at runtime, where the necessary information about a simulated system is dynamically acquired through the user who analyzes the system based on intermediate simulation results and identifies which reactions are slow or fast. In summary, our algorithm switching scheme takes the following procedure.

1. Program launch: Start the simulation with the exact SSA.
2. Prompt user: The execution pauses after a preset time to show intermediate system state for the user to analyze.
3. System identification: The user identifies fast reactions and inputs the information to the running simulation.
4. Algorithm switching: The simulation resumes execution by switching to the ss-SSA algorithm based on the user input.

4. DYNAMIC ALGORITHM SWITCHING IMPLEMENTATION

The target scientific software for our dynamic algorithm switching approach is StochKit [21], a flexible C++ stochastic simulation library on parallel environments. StochKit provides a core set of algorithm implementations including the exact SSA with MPI (Message-Passing Interface) [22] interfaces for large scale Monte Carlo simulations. Implementing application-specific adaptation scenarios in AOP typically involves

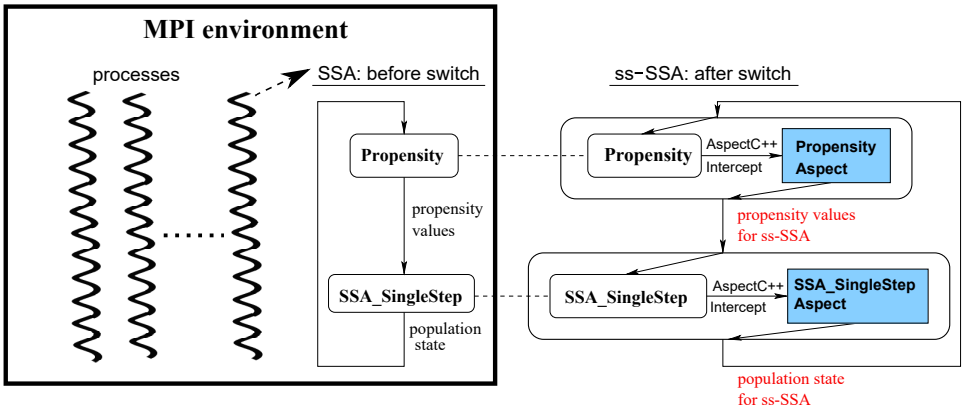

Fig. 3. Switching from Exact SSA to ss-SSA via AspectC++ aspects on a parallel environment

identifying adaptation control points for specifying pointcut expressions and defining adaptation operations. We describe theses two stages for implementing algorithm switching scenarios with StochKit in the following.

Identifying Algorithm Switching Control Points

The exact SSA simulation within the StochKit modules are controlled by the `SSA_SingleStep` function in `SSA.cpp` and the `Propensity` function in `ProblemDefinition.cpp`. `SSA_SingleStep` takes the current simulation time and populations of each species as input parameters and advances the simulation by choosing the next reaction to fire. `Propensity` takes the current system state as input parameter and calculates the propensity values of each species to update the probability for reactions to occur in the next simulation step. Therefore, to control the SSA simulation behavior, we consider these two SSA implementation functions as adaptation targets to switch to the ss-SSA algorithm.

Defining Switching Logic

To switch from the exact SSA to the ss-SSA, the characteristics of the simulated system need to be obtained from the user at runtime. Hence, we intercept the calls to `SSA_SingleStep` by using the AspectC++ **`before`** advice at the **`execution`** pointcut of `SSA_SingleStep`. Here, we stop the program execution after a pre-defined time in the beginning of the simulation and prompt the user with intermediate simulation results, so that the user can determine if the given problem can be solved using the ss-SSA. In addition to `SSA_SingleStep`, the calls to `Propensity` need to be caught to modify the probabilities of reaction channels to fit to the ss-SSA. To entirely replace the original `Propensity` function, we use AspectC++ **`around`** advice at the **`execution`** pointcut of `Propensity`. By substituting the ss-SSA version of propensity values, we effectively suppress the probabilities of fast reaction events, so that only the slow reactions are simulated while fast reactions are calculated by analytical solutions.

Fig. 3 illustrates the overall structure of how the original exact SSA functions in StochKit, `SSA_SingleStep` and `Propensity`, are adapted to realize the intended ss-SSA behavior on parallel environments by the algorithm switching aspects written in AspectC++.

To present the implementation details, Fig. 4 shows the AspectC++ aspect code for

algorithm switching in our application. The simulation stops to accept the user's input at $t = STOPTIME$ of simulating the first sample, switching to the ss-SSA very early in the simulation (line 5). At line 11, to intercept the `SSA_SingleStep` calls, we use the **`before`** advice at the **`execution`** pointcut of `SSA_SingleStep`. At line 13 and 14, we fetch the current system state (i.e., population of each species) and simulation time passed as the first and the second argument of `SSA_SingleStep` using the AspectC++ `tjp->arg()` function parameter access API. If algorithms has not been switched (line 16) and simulation pause time has reached (line 17), the simulation stops for the MPI root process to show the current state of the simulation (line 24) and to prompt the user for information about possible fast reactions (line 25). The information provided by the user is stored in `propensity_zeros` and broadcast across the MPI environment to all the participating processes (line 29 and 31), which will switch to the ss-SSA algorithm for the remaining simulation using the broadcast information. Once the algorithm is switched to the ss-SSA, the aspect code computes an analytical solution for the population of each fast species identified by the user, and updates the system states accordingly (line 36).

In StochKit simulations, the propensity for each reaction depends on the given problem and is specified as a C++ function in a problem definition source file. Similar to the SSA aspect code, our approach implements a simple AspectC++ aspect code for the propensity function, so that the calls to the original propensity function are intercepted to realize the ss-SSA style simulation where the propensity value of each fast reaction, which was previously identified by the user, is set to zero after algorithm switching.

5. EXPERIMENTAL RESULTS AND EVALUATION

This section evaluates experimental results of applying our dynamic algorithm switching implementation to scientific simulations with StochKit using a simple biochemical system. The simulations were performed on a Linux cluster with each compute node comprising 2x Intel Xeon E5-2470v2 10 core @ 2.40Ghz, 96GB Main memory @ 1600MT/s, and 80GB SSD system drive, interconnected with 10Gbit Ethernet.

5.1 Fast Reversible Isomerization

We consider the fast reversible isomerization process with a single slow reaction [20], which can be described as the following equations.

$$S_1 \underset{c_2}{\overset{c_1}{\rightleftarrows}} S_2 \quad \textit{(fast)}, \qquad S_2 \xrightarrow{c_3} S_3 \quad \textit{(slow)},$$

S_i denotes the species in the simulated biochemical system and c_j is the reaction rate constant for the reaction channel R_j of the system. Given the system's state vector $X(t) \equiv (x_1(t), x_2(t), x_3(t))$, with $x_i(t)$ denoting the number of molecules of species S_i at time t, the simulated system's propensity functions and state-changing vectors can be specified as follows.

$$\begin{aligned} a_1(x) &= c_1 x_1, \quad \nu_1 = (-1,+1,0), \\ a_2(x) &= c_2 x_2, \quad \nu_2 = (+1,-1,0), \\ a_3(x) &= c_3 x_2, \quad \nu_3 = (0,-1,+1). \end{aligned}$$

As proved in the slow-scale SSA paper [20], the ss-SSA can be used to approximate the exact SSA for this system such that the analytical solution of the deterministic reaction

```
1  /* AlgoSwitchSSA.ah - algorithm switching aspect for SSA.cpp in StochKit */
2  #include <mpi.h>
3  #include "Vector.h"
4
5  extern double STOPTIME;
6  bool algo_switched = false;
7  int propensity_zeros[NCHANNELS];
8
9  aspect AlgoSwitchSSA {
10   /* intercept the call to SSA_SingleStep using before advice */
11   advice execution("% ...::SSA_SingleStep(...)") : before() {
12     /* fetch arguments for system vector and current simulation time */
13     CSE::Math::Vector *origX = (CSE::Math::Vector*) tjp->arg(0);
14     double *pTime = (double *) tjp->arg(1);
15
16     if (!algo_switched) { // algorithm not switched
17       if (*pTime > STOPTIME) {
18         int rank, size;
19         rank = MPI::COMM_WORLD.Get_rank();  // get current process id
20         size = MPI::COMM_WORLD.Get_size();  // get number of processes
21
22         /* if root process, prompt the user with current system states */
23         if (rank == 0) {
24           showSimulationStates();
25           getFastReactionsFromUser();
26           }
27         }
28         /* broadcast info about fast reactions and algorithm switching */
29         MPI::COMM_WORLD.Bcast(propensity_zeros, NCHANNELS, MPI::INT, 0);
30         algo_switched = true;
31         MPI::COMM_WORLD.Bcast(&algo_switched, 1, MPI::BOOL, 0);
32       }
33     }
34     else {  // algorithm switched
35       CSE::Math::Vector& xx = *origX;
36       updatePopulation(xx);  // calculate and update system states
37     }
38   }
39 };
```

Fig. 4. Aspect code for switching from SSA to ss-SSA in AspectC++

rate equation for fast species can be substituted at each exact SSA step,

$$x_T = x_1 + x_2, \quad x_1 = \frac{c_2 x_T}{c_1 + c_2}, \quad x_2 = x_T - x_1.$$

For the algorithm switching simulation with fast reversible isomerization and its measurements, we use the following parameter values.

$$\begin{aligned} &c_1 = 1, \quad c_2 = 2, \quad c_3 = 5 \times 10^{-5}, \\ &x_1 = 1200, \quad x_2 = 600, \quad x_3 = 0 \quad \text{at } t = 0, \\ &\text{FINALTIME} = 10000, \end{aligned}$$

where FINALTIME denotes the simulation finish time until which the system is set to evolve.

Fig. 5 shows the AspectC++ aspect implementation of propensity function for the ss-SSA switching of the fast reversible isomerization with the given parameter values. The ss-SSA switching implementation was programmed to stop for user's input with $STOPTIME = 1.0$ (line 5). After the user determines the type of each species and inputs the information, the remaining samples are simulated with the ss-SSA implementation. We use the AspectC++ **around** advice at the execution of the `Propensity` function (line

```
1  /* AlgoSwitchPropensity.ah - algorithm switching aspect for ProblemDefinition.cpp */
2  #include "Vector.h"
3
4  const double c1=1, c2=2, c3=5e-5;  // assumed reaction rates for fast isomerization
5  const double STOPTIME = 1.0;
6  extern bool algo_switched;
7  extern int propensity_zeros[NCHANNELS];
8
9  aspect AlgoSwitchPropensity {
10   /* intercept the call to Propensity using around advice */
11   advice execution("% ...::Propensity(...)") : around() {
12     if (algo_changed == false) tjp->proceed(); // call orig function if not switched
13     else {
14       /* fetch first argument for system state vector */
15       CSE::Math::Vector *origX = (CSE::Math::Vector*) tjp->arg(0);
16       /* calculate propensity for slow-scale SSA */
17       CSE::Math::Vector retval = SlowScalePropensity(*origX);
18       tjp->proceed(); // need to call orig function to access return values
19       /* fetch the location of return value and set the propensity */
20       CSE::Math::Vector *result = (Vector*) tjp->result();
21       *result = retval;
22     }
23   }
24
25   /* return slow-scale propensity of fast isomerization */
26   Vector SlowScalePropensity(const Vector& x)
27   {
28     /* use xx1_inf rather to calculate propensity */
29     long xt = (long) (x(0) + x(1));
30     double xx0_inf = xt * ((double)c1/(c1+c2));
31
32     Vector copyX = x;
33     copyX(0) = xx0_inf;
34
35     Vector a(3);
36     a(0) = c1*copyX(0); a(1) = c2*copyX(1); a(2) = c3*copyX(1);
37
38     return a;
39   }
40 };
```

Fig. 5. Aspect code for propensity calculation in switching from SSA to ss-SSA in AspectC++

11) to replace the original propensity function. If the algorithm is not switched yet, we proceed with the original call (line 12). Otherwise, we fetch the current population passed as the first argument of `Propensity` using `tjp->arg()` (line 15) and invoke the ss-SSA version of propensity function, `SlowScalePropensity` separately written at line 26, with the obtained population value (line 17). To manipulate the return value of the propensity function call by substituting the ss-SSA propensity values for the original call, we need to call the original function and locate its return value using AspectC++ `tjp->result()` (line 20). In fact, this is one of the weak points of AspectC++, where the return value of a function is not easily accessed for manipulation and executing the replaced function is required when using the **`around`** advice to replace a function call. Finally, the return value of propensity is set to be the value of the ss-SSA version at line 21.

Table 1 shows the statistical properties of the simulation results using the implemented algorithm switching for evolving 4,096 samples in comparison with the exact SSA simulation. As shown in the table, the differences in the mean and the standard deviation of each species population between the algorithm-switched ss-SSA simulation and the exact SSA simulation are negligible, which demonstrates the effectiveness of our algorithm switching implementation. This is also backed up by the probability density plots of the ss-SSA (orange dotted) in Fig. 6 which appear to closely match the exact

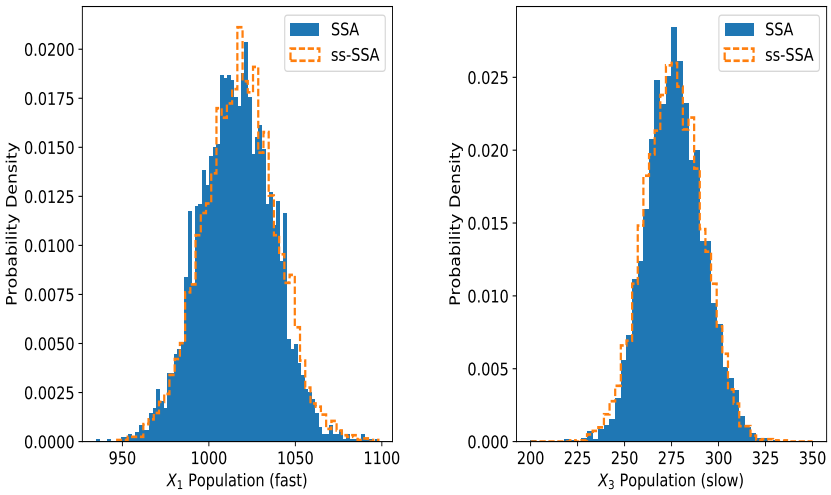

Fig. 6. Probability densities of exact SSA (blue bar) vs. algorithm-switched ss-SSA (orange dotted) for fast reversible isomerization

SSA's (blue bar) in both the fast and slow reactions. Overall, Table 1 and Fig. 6 show that the ss-SSA is an effective approximation algorithm for the exact SSA in the case of fast reversible isomerization.

	$\mathbf{X}_1$ **(fast)**		$\mathbf{X}_2$ **(fast)**		$\mathbf{X}_3$ **(slow)**	
	Mean	Std	Mean	Std	Mean	Std
SSA	1015.88	21.06	508.18	19.11	275.93	15.62
ss-SSA	1018.07	20.96	506.39	18.83	275.53	15.39

Table 1. Statistics of exact SSA vs. algorithm-switched ss-SSA for fast reversible isomerization

Fig. 7 shows the execution time of the algorithm-switched ss-SSA in comparison with the exact SSA simulations. While the simulation results are virtually the same in both cases, the gain in computational efficiency is substantial in the algorithm switching case. The performance gain is from around 10× (256 processes) to almost three orders of magnitude (4 processes). With 4 processes generating 1,024 samples each, for instance, the algorithm switching case took only 3.9 seconds on each process, out of which only 0.25 seconds was consumed by the ss-SSA for generating 1,023 samples and the rest was used by the exact SSA for realizing one sample, getting the user input, and the MPI communication among the processes. In contrast, the exact SSA took almost 1 hour to complete all the 1,024 samples on each process.

The algorithm-switched ss-SSA simulations do not appear to benefit much as the number of processes increases due to the relatively large overhead of MPI communications on parallel environments compared with the very efficient execution of the ss-SSA simulation. Simulating an extremely large number of samples with much greater FINAL-TIME would show performance improvements with the growth in the number of pro-

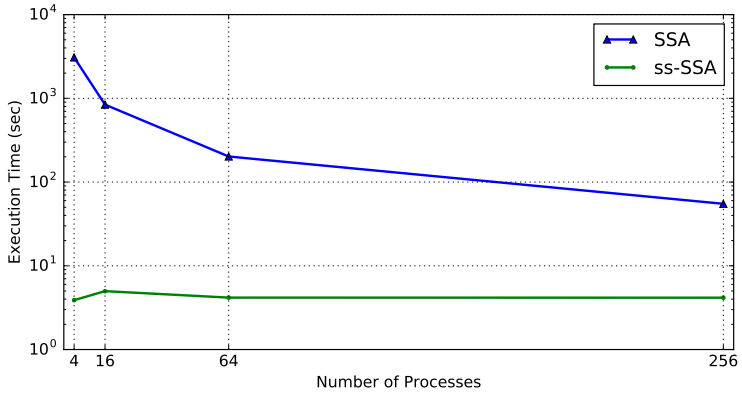

Fig. 7. Execution time of exact SSA vs. algorithm-switched ss-SSA simulations for evolving 4096 samples in fast reversible isomerization

cesses in parallel execution.

6. RELATED WORK

There is a sizable amount of research on implementing dynamic algorithm switching across different levels of the software stack. At the algorithmic level, adaptive methods for automatically selecting optimal algorithms to match a given problem include multi-method adaptive solvers [1], algorithm selection frameworks [2, 23], and AI (artificial intelligence) planning techniques [24]. The distinction against our work here is that the adaptive algorithm selection methods require switching operations to be explicitly programmed within an application before application launch.

Projects at the programming language level include Program Control Language (PCL) [25] for specifying application-specific adaptation strategies for distributed parallel applications, and the ADAPT (Automated De-coupled Adaptive Program Transformation) [26] language for describing runtime optimization heuristics. These projects are similar to our work in that adaptation strategies are described at the language level. However, unlike our approach based on popular programming language, these projects require adopting and learning new domain-specific languages, which can be a non-trivial burden for working with existing scientific software.

Other approaches to algorithm switching include competitive parallel execution (CPE) [27]. CPE is a technique to leverage multi-core or multi-processor systems by introducing and concurrently executing multiple alternatives (called *variants*) for parts of a sequential program, thereby choosing the "best" option for a given problem or execution environment. The runtime system of CPE enables isolated execution between the variants to ensure the behavior of the CPE-enabled program is not distinguishable from that of the original sequential program. The major distinction between CPE and our approach is that CPE targets sequential programs, while our approach focuses on algorithm switching for parallel applications. In addition, CPE requires a specialized API and runtime systems, which is in contrast to our approach based on readily accessible C++ language extension.

There have been a few efforts to apply the AOP paradigm and its software engineering methodology to scientific programming on cluster environments. Most notably, Han et

al. [28] apply AOP mechanisms [29] to implementing MPI extensions for fault tolerance and heterogeneity on top of existing MPI frameworks such that modularity and reusability are secured in the software development perspective. Their work is in contrast to our approach in that they focus on implementing additional functionalities at the middleware level of the parallel software stack, while our approach is centered around implementing algorithm switching in scientific simulations towards realizing program behavior adaptations at the application level.

Most relevant to our approach is the dynamic weaving mechanism offered by AOP languages and frameworks. Dynamic weaving is usually implemented by the languages equipped with managed runtimes such as Java, while AspectC++ used in our approach provides only static weaving based on source-to-source translation. In the next subsection, we describe dynamic aspect weaving of Java AOP in detail.

6.1 Dynamic Aspect Weaving of AOP Frameworks in Java

There are two types of the dynamic weaving mechanism in Java AOP depending on the time of weaving: load-time weaving (LTW) and runtime weaving. In LTW, weaving is performed by instrumenting Java bytecode files at the time of class loading and it has typically been implemented by replacing the default system classloader to perform instrumentation with a substituted classloader [30, 31]. Beside the custom classloader method, AspectJ [32] also offers advanced LTW mechanisms using the Java Virtual Machine Tools Interface (JVMTI) facility, which allows the load-time weaver to intercept the loading of a class.

In contrast to LTW, runtime weaving enables program behavior modification while the application is running. Most approaches exploit the support from the JVM (Java Virtual Machine). For instance, PROSE [13] implements a JVM plug-in for runtime code modification using the JVM debug interface and Wool [33] uses JVM's HotSwap mechanism to reload class files dynamically. Steamloom [34] implements runtime code modification as an extension of the VM and HotWave [35] uses standard AspectJ weaving tools to ensure compatibility with standard JVMs.

While Java AOP's dynamic weaving is attractive thanks to its sophisticated software engineering techniques for realizing program adaptation, Java and its AOP constructs have been scarcely used in parallel scientific applications [36, 37]. This is mainly due to the inevitable performance overhead caused by the VM layer and its managed execution environment.

7. CONCLUSIONS

In this paper, we presented a modular approach to implementing dynamic algorithm switching in parallel simulations. Our approach applies the AOP paradigm where dynamic algorithm switching is considered as a separate concern in scientific software development and uses modern AOP software engineering mechanisms to implement a user-driven algorithm switching functionality onto a real-world scientific simulation application in a modularized fashion. We demonstrated that using our approach can achieve significant performance speedups in parallel scientific simulations by dynamically switching to an efficient algorithm well suited to a given input problem. Performance benefits of using dynamic algorithm switching is substantial compared to conventional methods that blindly stick to a single, computationally intensive algorithm without taking the characteristics properties of the given problem into account over the entire course of the simulation. Overall, we expect that our approach for improved modularity and performance benefits

in developing application-specific parallel programs will be useful as modern scientific software becomes increasingly complex and large-scale.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP; Ministry of Science, ICT & Future Planning) (No. 2017R1C1B2009361). This work was supported by a 2017 research grant from Youngsan University, Republic of Korea.

REFERENCES

1. S. Bhowmick, L. McInnes, B. Norris, and P. Raghavan, "Robust algorithms and software for parallel PDE-based simulations," in *Proceedings of the Advanced Simulation Technologies Conference, ASTC'04, April 18-22, 2004*. Society for Modeling and Simulation International (SCS), 2004.
2. N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger, "A Framework for Adaptive Algorithm Selection in STAPL," in *PPoPP '05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2005, pp. 277–288.
3. M. Anderson, M. Brodowicz, L. Dalessandro, J. DeBuhr, and T. Sterling, *A Dynamic Execution Model Applied to Distributed Collision Detection*. Cham: Springer International Publishing, 2014, pp. 470–477.
4. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the European Conference on Object-Oriented Programming*, Vol. 1241. Berlin, Heidelberg, and New York: Springer-Verlag, 1997, pp. 220–242.
5. G. Kiczales and M. Mezini, "Aspect-Oriented Programming and Modular Reasoning," in *Proceedings of the 27th International Conference on Software Engineering*. ACM, 2005, pp. 49–58.
6. P. Kang, M. A. Heffner, N. Ramakrishnan, C. J. Ribbens, and S. Varadarajan, "Adaptive Code Collage: A Framework to Transparently Modify Scientific Codes," *IEEE Computing in Science and Engineering*, Vol. 14, no. 1, 2012, pp. 52–63.
7. P. Kang, "Modular Implementation of Dynamic Algorithm Switching in Parallel Simulations," *Cluster Computing*, Vol. 15, no. 3, Sep. 2012, pp. 321–332.
8. O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language," in *CRPIT '02: Proceedings of the 40th International Conference on Tools Pacific*. Darlinghurst, Australia: Australian Computer Society, Inc., 2002, pp. 53–60.
9. M. Fleury and F. Reverbel, "The JBoss Extensible Server," in *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*. Rio de Janeiro, Brazil: Springer-Verlag New York, Inc., 2003, pp. 344–373.
10. B. De Win, F. Piessens, W. Joosen, and T. Verhanneman, "On the Importance of the Separation-of-Concerns Principle in Secure Software Engineering," in *Workshop on the Application of Engineering Principles to System Security Design*, 2002, pp. 1–10.
11. E. Hilsdale and J. Hugunin, "Advice Weaving in AspectJ," in *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*. New York, NY, USA: ACM Press, 2004, pp. 26–35.

12. P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. De Moor, D. Sereni, G. Sittampalam, and J. Tibble, "abc: An Extensible AspectJ Compiler," 2006, pp. 293–334.
13. A. Popovici, T. Gross, and G. Alonso, "Dynamic Weaving for Aspect-Oriented Programming," in *AOSD '02: Proceedings of the 1st International Conference on Aspect-Oriented Software Development*. New York, NY, USA: ACM, 2002, pp. 141–147.
14. A. Popovici, G. Alonso, and T. Gross, "Just-in-time Aspects: Efficient Dynamic Weaving for Java," in *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, ser. AOSD '03. New York, NY, USA: ACM, 2003, pp. 100–109.
15. C. Zhang and H.-A. Jacobsen, "TinyC^2: Towards Building a Dynamic Weaving Aspect Language for C," in *Proceedings of the 2nd AOSD Workshop on Foundations of Aspect-Oriented Languages*, Boston, MA, USA, March 2003, pp. 25–34.
16. A. Vasseur, "Dynamic AOP and Runtime Weaving for Java - How Does AspectWerkz Address It?" in *Proceedings of the 2004 Dynamic Aspect Workshop (DAW'04)*, R. E. Filman, M. Haupt, K. Mehner, , and M. Mezini, (eds.), Lancaster, England, March 2004, pp. 135–145.
17. D. T. Gillespie, "A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Reactions," *Journal of Computational Physics*, Vol. 22, no. 4, December 1976, pp. 403–434.
18. D. T. Gillespie, "Exact Stochastic Simulation of Coupled Chemical Reactions," *The Journal of Physical Chemistry*, Vol. 81, no. 25, 1977, pp. 2340–2361.
19. Y. Cao, D. T. Gillespie, and L. R. Petzold, "Accelerated Stochastic Simulation of the Stiff Enzyme-Substrate Reaction," *Journal of Chemical Physics*, Vol. 123, no. 14, October 2005, pp. 144 917.1–144 917.12.
20. Y. Cao, D. T. Gillespie, and L. R. Petzold, "The Slow-Scale Stochastic Simulation Algorithm," *Journal of Chemical Physics*, Vol. 122, no. 1, January 2005, p. 014116.
21. H. Li, Y. Cao, L. R. Petzold, and D. T. Gillespie, "Algorithms and Software for Stochastic Simulation of Biochemical Reacting Systems," *Biotechnology Progress*, Vol. 24, 2008, pp. 56–61.
22. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA: MIT Press, 1999.
23. H. Yu, D. Zhang, and L. Rauchwerger, "An Adaptive Algorithm Selection Framework," in *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 278–289.
24. T. A. Johnson and R. Eigenmann, "Context-Sensitive Domain-Independent Algorithm Composition and Selection," *SIGPLAN Not.*, Vol. 41, no. 6, 2006, pp. 181–192.
25. B. Ensink, J. Stanley, and V. Adve, "Program Control Language: A Programming Language for Adaptive Distributed Applications," *J. Parallel Distrib. Comput.*, Vol. 63, no. 11, 2003, pp. 1082–1104.
26. M. J. Voss and R. Eigenmann, "High-level Adaptive Program Optimization with ADAPT," in *PPoPP '01: Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*. New York, NY, USA: ACM, 2001, pp. 93–102.
27. O. Trachsel and T. R. Gross, "Variant-based Competitive Parallel Execution of Sequential Programs," in *Proceedings of the 7th ACM International Conference on Computing Frontiers*. New York, NY, USA: ACM, 2010, pp. 197–206.

28. H. Han, H. Jung, and H. Y. Yeom, "Aspect-Oriented Development of Cluster Computing Software," *Cluster Computing*, Vol. 14, no. 4, 2011, pp. 357–375.
29. A. Zaidman, S. Demeyer, B. Adams, K. De Schutter, G. Hoffman, and B. De Ruyck, "Regaining Lost Knowledge Through Dynamic Analysis and Aspect Orientation," in *Proceedings of the Conference on Software Maintenance and Reengineering*, ser. CSMR '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 91–102.
30. Ralph Keller and Urs Hölzle, "Binary Component Adaptation," in *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings*, 1998, pp. 307–329.
31. G. A. Cohen, J. S. Chase, and D. L. Kaminsky, "Automatic Program Transformation with JOIE," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1998, pp. 14–14.
32. AspectJ, http://www.eclipse.org/aspectj [accessed on May 1, 2017].
33. S. Chiba, Y. Sato, and M. Tatsubori, "Using HotSwap for Implementing Dynamic AOP Systems," in *1st Workshop on Advancing the State-of-the-Art in Run-time Inspection*, 2003, p. 1.
34. C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann, "Virtual Machine Support for Dynamic Join Points," in *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*. Lancaster, UK: ACM, 2004, pp. 83–92.
35. A. Villazón, W. Binder, D. Ansaloni, and P. Moret, "Advanced Runtime Adaptation for Java," *ACM SIGPLAN Notices*, Vol. 45, no. 2, 2010, pp. 85–94.
36. B. Harbulot and J. R. Gurd, "Using AspectJ to Separate Concerns in Parallel Scientific Java Code," in *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*. Lancaster, UK: ACM, 2004, pp. 122–131.
37. D. K. Kim, E. Tilevich, and C. J. Ribbens, "Dynamic Software Updates for Parallel High-Performance Applications," *Concurrency and Computation: Practice and Experience*, Vol. 23, no. 4, 2011, pp. 415–434.

Pilsung Kang is an Assistant Professor in the Department of Computer Engineering at Youngsan University, South Korea. His research interests include computational science, parallel systems, high-performance software, and quantum computing. Kang has a PhD in computer science from Virginia Tech.