Novel Runtime Systems Support for Adaptive Compositional Modeling in PSEs

Srinidhi Varadarajan and Naren Ramakrishnan
Department of Computer Science
Virginia Tech, Blacksburg, VA 24061, USA
Email: srinidhi@cs.vt.edu, naren@cs.vt.edu

Abstract

Problem solving environments (PSEs) have progressed significantly in the past few years. The vision of truly seamless PSEs relies on runtime systems support that is cognizant of the operational issues underlying scientific computations and, at the same time, is flexible enough to accommodate diverse application scenarios. This paper presents a PSE runtime support solution through a novel combination of two computational technologies – *Weaves*, a source-language independent parallel runtime compositional framework that operates through reverse-analysis of compiled object files, and runtime recommender systems that aid in dynamic knowledge-based application composition. Domain-specific adaptivity is exploited through runtime recommendation of code modules and a sophisticated checkpointing framework for transparent deployment. A core set of "adaptivity schemas" are provided as templates for adaptive composition of large-scale scientific computations. Implementation issues, motivating application contexts, and preliminary results are described.

1 Introduction

Problem solving environments (PSEs) are evolving from standalone systems to complex, networked, entities seamlessly integrating geographically disparate resources in a single application. Specific trends contributing toward this evolution are the increasing componentization of scientific codes, emerging system infrastructures such as the Grid [Foster et al., 2001], and the runtime constraints posed by novel computational science applications. To be effective in these emerging environments, PSEs must provide high-level, powerful, computational primitives within the context of the emerging system infrastructures. This requires both an understanding of the architectural assumptions of today's computational systems and an appreciation for how disciplinary scientists do computational science. The focus of this paper is **runtime systems support** that is cognizant of the operational issues underlying PSE infrastructure and is flexible enough to accommodate diverse application scenarios.

An area where runtime systems support holds great promise is in the engineering of adaptivity - adaptivity in terms of algorithm selection, architectural tuning, and exploiting the underlying scientific usage contexts [Foster et al., 2001]. We posit a broad picture of adaptivity here, one which is not restricted to identifying partitioning parameters, modifying data decompositions, or parallel scheduling; instead, adaptivity is proposed at a more logical unit of algorithms and object codes. This viewpoint leads to scientific codes being organized in a model-based framework for adaptive composition, execution, and performance analysis [Adve et al., 2002]. We use the term **compositional modeling** in this paper to collectively refer to all three aspects of **model specification** (how to define the composed elements?), **model execution** (how to execute composed codes?), and **model analysis** (how to use performance information from execution to evaluate and improve models?).

These considerations lead us to identifying two important requirements for runtime systems support in PSEs. First, runtime systems support should enable a transparent transition path for composing and executing legacy codes, without requiring that they be rewritten to achieve this functionality. Second,

runtime adaptivity should allow the dynamic selection, reconfiguration, and execution of code modules, taking into account performance considerations, problem characteristics, and dynamic system infrastructures.

Solution Approach

Our solution approach for runtime systems support is two-pronged: (i) a novel compositional system with checkpointing support for deployment over HPC platforms, and (ii) realizing domain-specific adaptivity through a runtime recommender system. A core set of "adaptivity schemas" constitute a reconfigurable approach to steering and managing large-scale scientific computations.

In this context, a high-level problem specification (e.g., "solve this elliptic PDE with a relative accuracy of 10^{-6} and time less than 600 seconds") is provided to a recommender system that makes an initial recommendation of code modules (e.g., "use a finite-difference discretizer with red-black ordering"). These code modules are communicated to the compositional system as a "configuration", which are then scheduled and executed; as the computation progresses (e.g., the PDE gets discretized and the resulting linear system appears to be ill-conditioned), feedback is provided to the runtime recommender through the checkpointing mechanism, which uses this information to perhaps dynamically insert a preconditioner before the linear solver in the solution loop. The configuration is updated with this selection, and the computation is re-scheduled. This interplay between the compositional system (which supports object-based composition, migration, and checkpointing) and the runtime recommender (which enables dynamic selection of code modules) leads to a novel runtime framework for scientific computations.

In this Paper

Section 2 identifies two core computational technologies that form the basis of our solution for runtime systems support. Section 3 elaborates on how these technologies are integrated to provide novel systems support for PSEs. Section 4 identifies a set of "adaptivity schemas" that can be used as templates for realizing many complex, adaptive, scientific computations. Section 5 presents early results and outlines work in progress. A concluding discussion placing this work in context of emerging trends is provided in Section 6.

2 Core Computational Technologies

Our approach to supporting adaptive compositional modeling centers on two core computational technologies: the *Weaves* parallel compositional framework, and data-driven runtime recommender systems. We discuss them in detail in the context of a real scientific application.

2.1 Motivating Application

Our driver application involves the idea of *collaborating partial differential equation (PDE) solvers* [Drashansky et al., 1999] for solving heterogeneous multi-physics problems. For instance, simulating a gas turbine requires combining models for heat flows (throughout the engine), stresses (in the moving parts), fluid flows (for gases in the combustor), and combustion (in the engine cylinder). Each of these models can be described by an ODE/PDE with various formulations for the geometry, operator, and boundary conditions. The basic idea here is to replace the original multi-physics problem by a set of smaller simulation problems (on simple geometries) that need to be solved *simultaneously* while satisfying a set of interface conditions. The mathematical basis of this idea is the interface relaxation approach to support a network of interacting PDE solvers [McFaddin and Rice, 1992; see Fig. 1].

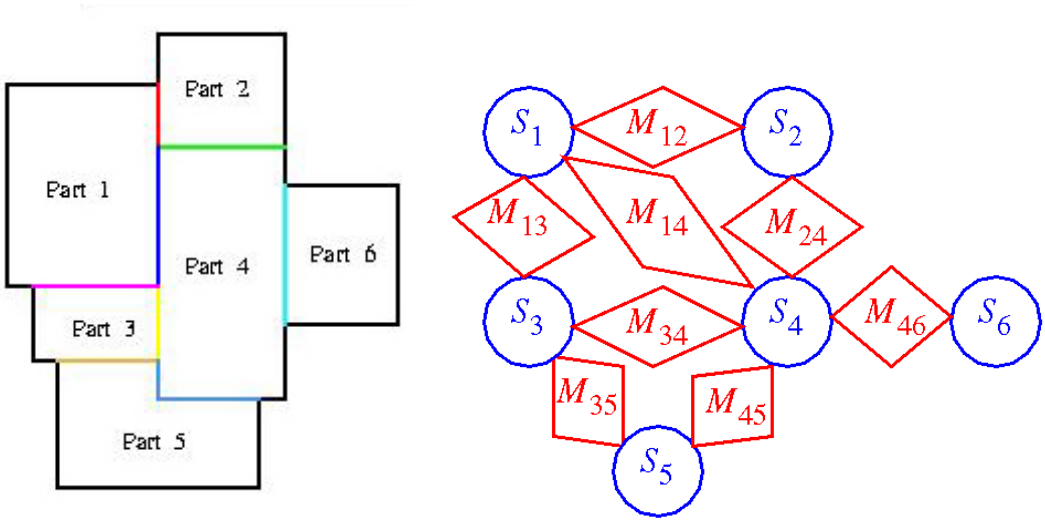

Figure 1: (left) Multi-physics problem with six subdomains with different PDEs. (right) A network of collaborating solvers (S) and mediators (M) to solve the PDE problem. Each mediator is responsible for agreement along one of the interfaces (colored lines).

Mathematical modeling of the multi-physics problem distinguishes between *solvers* and *mediators*. A PDE solver is instantiated for each of the simpler simulation problems and a mediator is instantiated for every interface to facilitate collaboration between the solvers. The mediators are responsible for ensuring that the solutions (obtained from the solvers) match properly at the interfaces. The term "match properly" is defined by the physics - if the interface is where the physics changes - or is defined mathematically (e.g., the solutions should join smoothly at the interface and have continuous derivatives). Distinguishing between solvers and mediators allows us to handle mathematical models naturally and elegantly; further, they can be organized to reflect the hierarchy of the physical structures (in this case, the turbine) underlying the computation.

In large-scale multi-physics simulations, it is not uncommon to have problems requiring collaboration between hundreds of solvers; one solver is assigned to each subdomain and the mediators issue instructions on appropriate boundary condition settings to their adjacent solvers. Once such a "network" of solvers and mediators is configured, it is scheduled for computation. After every iteration, the mediators might proceed to "adjust" the boundary condition settings to ensure a better matching of solutions or, if the change of boundary conditions is smaller than the tolerance, might report convergence. The essence of the collaborating solvers application can be studied from both the application composition and knowledge-based recommendation perspectives.

Application Composition Perspective: Suppose we have several instances of a solver task S_i running in parallel. These solvers need not be the same, e.g., in Figure 1, suppose S_1, S_2, S_3 and S_4 are instances of one solver (e.g., a standard finite difference method with direct Gaussian elimination) , S_5 and S_6 are instances of another solver (e.g., one that uses the GMRES iterative method and a suitable preconditioner), and so on. These solvers are contributing to a shared state maintained by the mediator task M_{ij}. The challenge is to implement codes exhibiting these characteristics - independent tasks, organized at multiple levels of parallelism, sharing state amongst themselves at different levels - and to do so *transparently*. Our emphasis hence is on a compositional framework – *Weaves* - that can transparently support arbitrary state sharing for scientific computations. Weaves extends the threads (which share all global state) and processes models (which separate all global state) to provide the generalized framework that allows selective sharing of global state.

Recommender Systems Perspective: The composite PDE solvers application also helps motivate the need for adaptivity in scientific computations; there are, literally, hundreds of well-defined software modules for supporting various aspects of the simulation process. There are multiple alternatives for numerical methods (iterative or direct solvers), numerical models (standard finite differences, collocation with cubic elements, Galerkin with linear elements, rectangular grids, triangular meshes), and various physical model assumptions and simplifications (e.g., cylindrical symmetry, steady state, rigid body mechanics, full 3D time-dependent physics). In addition, there are a variety of interface-relaxation methods [Rice et al., 1999] that can be implemented by a mediator. Performance information gathered at runtime can be fruitfully used to steer the dynamic selection of a suitable software module, which must then be linked in at runtime, executed, and possibly used to close the loop, to guide future compositions.

2.2 The Weaves Parallel Compositional Framework

Weaves is a source-language independent parallel framework for object-based composition of *unmodified* scientific codes. Weaves works through reverse-compiler analysis; by analyzing compiled ELF object files, Weaves enables the vast repository of legacy scientific libraries to be seamlessly used in a object-based compositional framework, *without requiring that these codes be written in an object oriented language*. Formally, Weaves

- provides a source language independent framework based on object code analysis
- provides transparent checkpointing/recovery support. Object code analysis automatically determines the state that needs to be checkpointed and/or restored *without user intervention*.
- provides support for performance data gathering via code instrumentation.
- supports notions of both *spatial* and *temporal* adaptivity (defined later), a critical element of runtime compositional modeling
- supports runtime migration of fine-grain code modules. As opposed to process migration, Weaves allows the **migration of parts of a composed application**.

As a compositional framework, perhaps the most important feature of Weaves is the modeling perspective it brings to bear on scientific computations; this enables scientific codes to be viewed in the context of a framework that integrates execution, simulation, and modeling of large-scale applications.

2.2.1 Defining the Weaves Framework

The major components of the Weaves programming framework are:

- **Module**: A module is any object file or collection of object files defined by the user. Modules have:
  - A **data context**, which is the global state of the module scoped within the object files of the module, and
  - A **code context,** which is the code contained within the object files that constitute the module. The code context may have multiple entry point and exit point functions.
- **Bead**: A bead is an instantiation of a module. Multiple instantiations of a module have independent data contexts, but share the same code context.
- **Weave**: A weave is a collection of data contexts belonging to beads of different modules. The definition of a weave forms the core of the Weaves framework. Traditionally, a process has a single name space mapped to a single address space. Weaves allow users to define multiple namespaces within a single address space, with user-defined control over the creation of a namespace.
- **String**: A string is a thread of execution that operates within a single weave. Similar to the threads model, multiple strings may execute within a single weave. However, a single string cannot operate under multiple weaves. Intuitively, a string operates within a single namespace.

Allowing a string to operate under multiple namespaces would violate the single valued nature of atomic variables.

- **Tapestry**: A tapestry is a set of weaves, which describes the structure of the composed application. The physical manifestation of a tapestry is typically a single process.

The above definitions have equivalents in object-oriented programming. A module is similar to a class and a bead - which is an instantiation of a module – is similar to an object. Tapestries are somewhat similar to object hierarchies. The major exception is that interaction between beads within a tapestry involves runtime binding. We chose to use our own terminology to avoid overloading the semantics of well-known OOP terms and also avert the implication that the framework requires the use of an OOP language.

Strings are similar to threads in that they can be dynamically instantiated and can also share the same copy of code. However, unlike threads, strings do not share global state. Each string has its own copy of global state. The main goal here is to avoid inadvertent sharing of state between unrelated instantiations of an algorithm, without having to modify the algorithm (ref. the collaborating solvers application).

Since strings are an intra-process mechanism, we will illustrate their operation by comparing and contrasting them to threads. A thread's state consists of (i) an instruction pointer (**IP**), (ii) a stack pointer and (iii) copy of CPU registers. Each thread within a process has its own stack frame that maintains local variables and a series of activation records that describes the execution path traversed by the thread. When a thread is created, the thread library creates a new stack frame and starts execution at the first instruction of the function specified by the thread instantiation call. When the thread scheduler needs to switch between threads, it saves the current IP, current stack frame, and the values in the CPU registers, switches to the state of the next thread, and starts execution at the IP contained in the thread state.

Strings involve an extension to the operation of threads. Similar to threads, each string has its own stack frame, which maintains local state. In addition, each string also has a copy of the global variables in an area called the ***weave context frame***, the start of which is pointed to by a ***weave context frame pointer***. A weave context defines the namespace of a string. This includes the global variables of all the beads traversed by a string. Note that some of the beads in a string may be shared between strings.

A string's state consists of (i) an instruction pointer, (ii) a stack frame pointer, (iii) copy of CPU registers, and (iv) a weave context frame pointer. When a string is created, the string creation call creates a stack frame and a weave context frame (if necessary) and copies the current state of the global variables into the weave context frame. The string creation call also associates a numerical identifier with the newly created string. Since creating a string involves copying its global variables, the string creation cost depends on the storage size of the global variables resulting in a higher creation cost than threads. We justify this cost by noting that it is a one time cost paid at program startup. Also, well-written applications are generally frugal in their use of global state, which mitigates the impact of the copy operation.

Similar to a thread scheduler, the string scheduler starts execution of the new string at the first instruction of a user-specified function. When the string library needs to switch between strings, it saves the current IP, current stack frame pointer, the values in the CPU registers, and the current weave context frame pointer, switches to the state of the next string and starts execution at the IP contained in the string state. The inter-string context switch cost is identical to threads.

Selective sharing of state in our framework operates at the level of individual beads. We illustrate the operation of selective sharing with the example shown in Figure 2 (also repeated in the Figure 5 below). The tapestry defines 4 weaves <Solver S_1, Mediator M_{12}>, <Solver S_2, Mediator M_{12}>, <Solver S_3, Mediator M_{34}> and <Solver S_4, Mediator M_{34}>, and 4 strings, with each string operating within a single

weave. At run time, context switching between the strings automatically switches the namespace associated with the string, preserving the sharing specified in the tapestry.

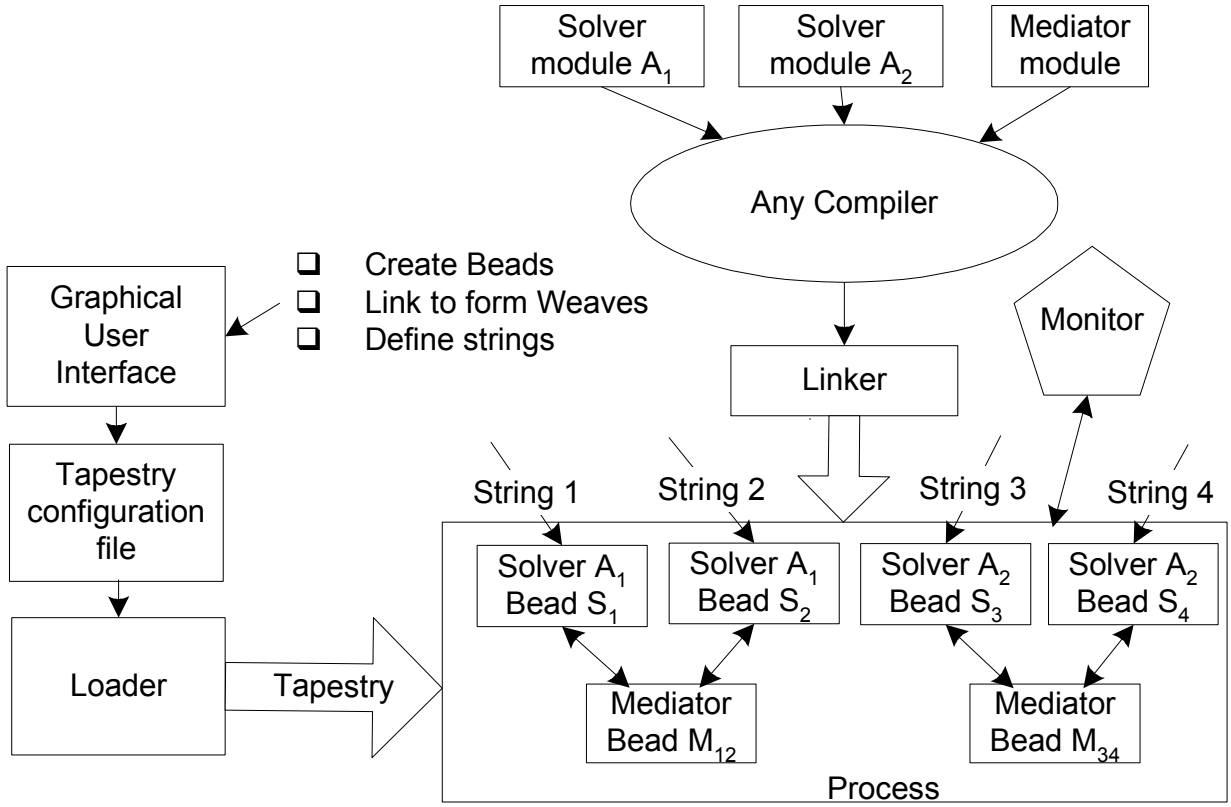

Figure 2: Interaction between the various components of the Weaves framework.

Figure 2 depicts the design process in the Weaves framework. The design process involves two entities: a *programmer* who implements the modules and a *composer*, who uses a graphical user interface to instantiate beads and define the various weaves and strings. The result of the GUI composition is a tapestry configuration file, which is used to load and execute the composed application. Each composed application also has a module called a *monitor* that is automatically linked with the composed application. In the process model, utilities like *ps* (in UNIX) can be used to query the run time of the process. The monitor provides a much more powerful IPC (Inter Process Communication) interface to such functionality. Utilities can query the monitor to determine the current tapestry, beads, strings, and weaves within a composed application.

2.2.2 Runtime Reconfigurability

The tapestry generated by the GUI is not necessarily a static composition. The Weaves programming framework allows applications to rewire themselves on the fly in response to dynamic conditions. Two forms of dynamic application composition are supported in the framework. In the first form, if the requisite modules are already linked into the original tapestry, Weave-aware applications can modify their structure by creating new beads, defining weaves, and instantiating strings at run-time. For non-Weave aware applications, the interface exposed by the monitor can be used to modify the tapestry of a composed application. These modifications may be manually made by a user at the command line or can be automatically generated by an external *resource monitoring* agent.

In the second form of dynamic composition, new code modules can be inserted into a running application through a modified dynamic library interface. In this mode of operation, the dynamically inserted code is analyzed at run-time. Dynamically inserted modules can be used in the same manner as statically inserted modules. This interface provides the full capabilities of Weaves, including arbitrary namespaces and compositional capabilities, in a run-time compositional framework. We will exploit this capability to investigate runtime algorithm selection and composition (see Section 3.1).

2.2.3 Tuple Spaces

The notion of selective state sharing in the Weaves programming framework presents a very powerful mechanism for defining namespaces. Since the definition of a weave permits any set of beads to define a namespace, any composition that can be represented by a connected graph (or a set of independent graphs) can be realized by this framework. From an application's perspective, the definition and operation of distinct namespaces is transparent. This mechanism presents a powerful compositional framework for any procedural code. The Weaves framework also supports the notion of *shared* tuple spaces, not elaborated here for space reasons.

2.2.4 Automatic Checkpointing and Recovery

A primary goal of the Weaves framework is to support adaptive applications that can rewire themselves dynamically in response to changing conditions. Our view of adaptivity encompasses optimistic algorithms that try to take the best execution path given a set of available options. However, the path chosen may not always be right, requiring applications to rollback to a known correct state. As discussed in the introduction, typical HPC applications also require checkpointing and recovery.

Traditionally, state checkpointing and restoration has been relegated to individual applications. This significantly adds to the complexity and maintainability of such codes. Furthermore, event driven codes add an additional layer of complexity. Since the path of execution through an event driven application is not known *a priori*, checkpointing and restoring such applications present significant challenges. The contribution of Weaves here is that it provides a *transparent support* framework that can checkpoint and recover state, *without application support*. The details are beyond the scope of this paper but essentially, note that a Weaves string maintains its global variables in the weave context frame and local variables and call invocation history in the stack frame. This compartmentalizes static state into two well defined regions. We can save the contents of the stack and weave context frames, effectively saving static state. A more involved solution tracks dynamic memory allocation during runtime. By employing a lightweight version of *copy-on-write* semantics, we support an efficient checkpointing mechanism.

2.3 Runtime Recommender Systems

Recommender systems [Ramakrishnan et al., 1998] provide facilities for automatic knowledge-based selection of solution components in PSEs. They help make selections of algorithms and code modules by taking into account both problem characteristics and performance considerations. Recommender systems involve the empirical evaluation algorithms on realistic, often parameterized, test problems, and interpreting and generalizing the results to guide selection of appropriate mathematical software. They are the preferred method of analysis in applications where domain knowledge is imperfect and for which our understanding of the factors influencing algorithm applicability is incomplete. For instance, when solving linear systems associated with finite-difference discretization of elliptic PDEs, there is little mathematical theory to guide a choice between, say, a direct solver and an iterative Krylov solver plus preconditioner. A recommender systems approach is to parameterize a suitable family of problems, and mine a database of thousands of PDE "solves" to gain insight into the likely relative performance of these two approaches (e.g., see Figure 6). Parameter sweep templates [Casanova and Berman, 2002] are thus an important tool for designing recommender systems.

In a traditional design of a recommender system [Ramakrishnan and Ribbens, 2000; Houstis et al., 2000], a database of test problems and algorithms is organized, and performance data is accumulated for the given problem population. This database of performance data is then mined (generalized) to arrive at high-level rules that can form the basis for a recommendation (for future problems). A variety of data mining algorithms are appropriate here (e.g., attribute-value generalizations, inductive logic programming; see [Houstis et al., 2000]). The MyPythia portal [Houstis et al., 2002] provides many interfaces to these algorithms, for both the recommender system builder and the recommender system user. In this system, the data collection phase is distinct from the generalization aspect (we refer to these as "offline" recommender systems); in other applications, data collection occurs in conjunction with data mining [Ramakrishnan et al., 2002], so that it can be "steered" to more accurately sample desired regions of the recommendation space.

In a PSE setting, recommender systems are important aids to application composition, by making dynamic selections of components (we refer to these as "runtime" recommender systems). The importance of a runtime recommender is easily seen in applications such as the collaborating PDE solvers, where selections need to be made of a discretizer, preconditioner, and linear system solver (in that order). Information needed to make a preconditioning recommendation or linear solver recommendation is not available *until after* the PDE has been discretized, hence such recommendations have to happen at runtime, using dynamic information. Specifically, a runtime recommender monitors a computational process, detects state-changes, and makes selections of solution components dynamically, thus aiding knowledge-based application composition at runtime. Designing a runtime recommender is thus more involved than an offline recommender because the database of problems and algorithm executions is not readily available and needs to be captured "on the fly".

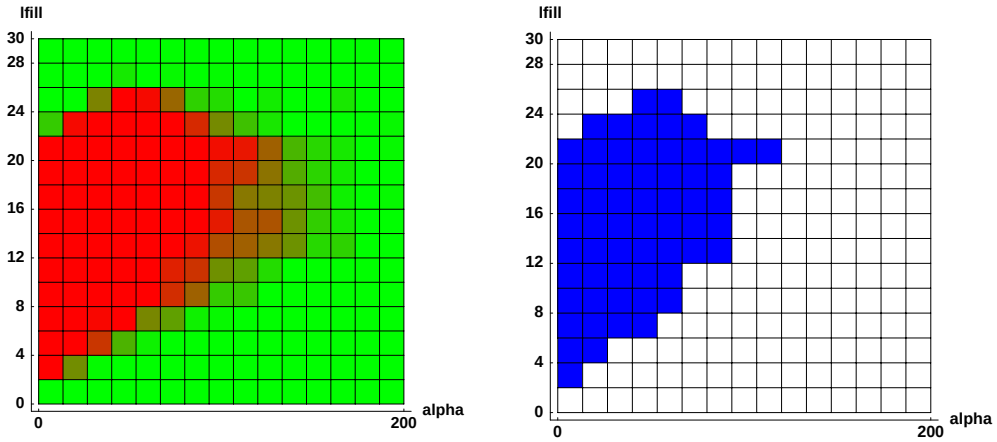

Figure 3: (left) Mining and visualizing recommendation spaces for selecting between a GMRES iterative solver (red) and a direct Gaussian elimination solver (green) to solve an elliptic PDE. **α** is a parameter controlling the singularity in the PDE problem (and hence, the ill-conditioning of the corresponding linear system) and ***lfill*** controls the pre-conditioning in the iterative solver. (right) A mined recommendation space with 90% confidence, showing the region where the GMRES solver is preferred. As **α** grows larger, it is seen that the ***lfill*** parameter must fall within a narrower range for the iterative solver to be preferred, until eventually the direct solver becomes the preferred choice. For more details, please see [Ramakrishnan and Ribbens, 2000].

2.3.1 Strategies for Runtime Recommendation

The primary problem faced by a runtime recommender is to observe a computational process (as it unfolds), make recommendations along the way, with the added complexity that feedback (about recommendations) is not immediate, and will arrive several timesteps (typically unknown) later. This is a problem reminiscent of *reinforcement learning* [Sutton and Barto, 1998], well studied in the control

systems and AI literature. Note that the task here is more ambitious than mere parameter tuning or building expert systems. The key issue is to tradeoff the cost of exploring the environment in the short-term with an accuracy improvement in the long-term. A runtime recommender systems thus grapples with a constant dilemma: should it choose a solver that it knows has worked before (**exploitation**) or should it "try" a different solver to see if it might lead to a performance improvement (**exploration**)?

Our approach to this problem is to model the scientific application as a non-deterministic, stationary system (the transition probabilities between states are assumed to be constant to ensure convergence of the learning algorithms). This network does not need to be handcrafted, but can be constructed online by the recommender system. For example, in the PDE application, "states" correspond to physical stages of the computational process and are represented by features such as singularity, current algorithm, order of the method, and performance criteria (set by the user). The "actions" correspond to choices made by the recommender, such as "use the ILU preconditioner", "switch from iterative to direct method", "decrease the current order". The goal now is to learn the *utility* of taking certain actions in various states. These utility estimations are summarized in the form of a control policy that chooses the action with the highest utility. On each step of the interaction, the recommender receives as input some indication of the current state (such as problem features) and it generates a recommendation as output. This recommendation changes the state of the system (e.g., at the end of the first stage of PDE solution, information about linear system characteristics becomes available), and the value of this state transition is communicated back to the recommender as reinforcement; which then chooses recommendations that will tend to increase the long-run sum of values of the reinforcement signal. Once again, there are a variety of learning algorithms for iterative improvement of generalizations.

The recommender begins in a mode that favors exploration over exploitation. These runs are typically scheduled during idle cycles on our computational cluster. Over time, the recommender encounters enough problems from its database and has explored enough alternatives that it can make an informed judgment about solution alternatives. At this point, its mode of operation becomes primary exploitative with only a small percentage of exploitation (to ensure that the learned utility values are current). Such an approach has been validated for selecting quadrature routines from a space of over 120 algorithms [Ramakrishnan et al., 2002] and for synthesizing type-insensitive codes for ODEs with both stiff and non-stiff regions (see Figure 4 for a policy mined by inductive logic programming). The functionality provided by a runtime recommender can be thought of as automatic determination of control policies to realize adaptivity in scientific codes. Runtime recommendation is traditionally concerned with code executions but can also be employed to assess model-based simulations and make selections of system configurations, as studied in the adaptive control formulation of [Adve et al., 2002].

```
qvalue(1) :-
    state(beginning), algorithm(none),
    action(choose-non-stiff).

qvalue(1) :-
    state(near-stiff), algorithm(non-stiff),
    estimate < threshold, sv < 10,
    action(switch-to-stiff).
```

Figure 4: A partially induced control policy mined by a runtime recommender for the task of solving ODEs with both stiff and non-stiff regions. From the beginning state, the recommender always prefers a non-stiff method (an Adams-Moulton method), but when its estimates improve its assessment of the evolution of solution components, it switches to a stiff method (in this case, an implicit A-stable formula). This approach should be contrasted to

classical differential equation software such as GEAR, LSODE, and DIFSUB where such adaptivity is realized through static decision-making code coupled with the ODE solver.

3 Systems Support for Adaptive Compositional Modeling

To summarize the two core technologies: runtime recommender systems allow the dynamic selection and composition of code modules and *Weaves* provides runtime systems support to realize such compositions. In this section, we further bring out the synergy between these technologies.

From the viewpoint of the Weaves design process, a runtime recommender system acts as the composer, dynamically determining the code modules and their instantiations. Specifically, the runtime recommender supplies the tapestry configuration file that specifies the composition graph. From the viewpoint of the recommender system, Weaves acts as the "end-effector" and provides systems support for checkpointing algorithm executions and modifying code modules in response to changing problem or platform characteristics.

3.1 Temporal and Spatial Adaptivity

Consider how the interaction between Weaves and runtime recommender systems would work for the task of adaptive numerical quadrature. Let us start with the collection of 120 quadrature algorithms described in [Ramakrishnan et al., 2002]. For a given numerical integration problem, the performance goal is to recommend a suitable quadrature routine such that the number of function evaluations is minimized. A recommender system (GAUSS) with this functionality is also described in [Ramakrishnan et al., 2002]. GAUSS can make suitable selections of algorithms from the quadrature library, monitor their execution, and change its recommendation if its earlier selection failed or otherwise did not satisfy the performance constraints. GAUSS is more than a polyalgorithm comprising of the 120 algorithms (where the decision procedures for selection are hardwired); it has the ability to use runtime information about algorithm performance dynamically as it becomes available. It functions by organizing a database of parameterized test problems and algorithm executions and uses an online mechanism to continually generalize from archived performance data.

This mode of operation in GAUSS can be viewed as a form of *temporal adaptivity*. The Weaves framework provides two notions of temporal adaptivity. In the first form – called *pessimistic temporal adaptivity* – the recommender system can dynamically select an algorithm already compiled into the executing application (using a form of an if-then-else construct) but doesn't have the ability to "retract" its recommendation. For this form of adaptivity to be successful, the recommendation space must be limited in its choice of algorithms to *only* those that produce correct results. Pessimistic temporal adaptivity is thus most suited for the exploitation mode of a recommender system. In the second form – called *optimistic temporal adaptivity* – the recommender system can dynamically choose from *any* algorithm that suits the purpose, including those that *may not* produce correct results all the time. Optimistic temporal adaptivity is ideal for the exploration mode of a recommender system.

Optimistic temporal adaptivity requires runtime systems support for checkpointing and recovery, since we need to (i) recover from failed instantiations of algorithms, and (ii) ensure that the recovery process doesn't result in the recommender system following the "failed" path again. Note that (ii) is a rather insidious issue. A perfect checkpoint/recovery mechanism will restore the recommender system state to just before its selection of the failed algorithm, which will result in the recommender system following the "failed" execution path repeatedly. What is needed is a checkpointing and recovery system that can provide a tuple (any set of variables defined in the namespace of the executing algorithm) view of the future, where the tuple presents intermediate results. In addition to pruning the search process of the recommender system, the tuple may be used to augment the features gathered by the recommender, helping it make a more informed decision.

Optimistic temporal adaptivity is a very powerful mechanism for supporting runtime recommendation and composition. The tuple view of the future provides not just algorithm state (in the form of variables comprising the tuple), but also the entire function invocation history prior to the failure (which includes the entire sequence of algorithm recommendations exercised). For applications such as adaptive quadrature, which are based on a divide-and-conquer strategy of repeated problem decompositions and algorithm recommendations, this feature is particularly important.

Weaves supports a further form of adaptivity called *spatial adaptivity*, where even the space of algorithms to be selected for composition is not known until runtime. For instance, consider a molecular electronics simulation, where a sequence of thousands of linear systems have to be solved to compute the I-V profile of a single device. The complete realization of such a simulation may span weeks to months. During execution new solvers may become available - especially in dynamic settings - which may offer better performance characteristics. What we need is a mechanism to transparently substitute the solver compiled into the executing binary with a new solver "on-the-fly" – dynamic function replacement. This notion is similar to dynamic classloaders in Java™.[1] The Weaves framework provides strong support for spatial adaptivity, including across multiple non-OOP source languages.

4 Adaptivity Schemas

In working with concerted groups of scientists and engineers, we have encountered a number of recurring "schemas" capturing how compositional scientific codes should be configured for adaptive execution. This section outlines these schemas and identifies application contexts where they are relevant.

Before we begin, it is pertinent to mention that two common modes of high-level problem solving – viz. **parameter sweeps** [Casanova and Berman, 2003], **algorithmic bombardment** [Barret et al., 1996] – are easily supported using the Weaves framework. Parameter sweeps embody rich opportunities for state sharing and overloading of function invocations, and Weaves enables such sweeps to be conducted within an economy of processes. Offline recommender systems rely on the ability to conduct multi-dimensional parameter sweeps effectively and economically. Algorithmic bombardment is a speculative strategy by which multiple algorithms or solution approaches are assigned to a given problem (simultaneously), some of which may not run to completion and/or may be terminated when they are deemed redundant. Simplistically, algorithmic bombardment can be implemented efficiently through spatial adaptivity. From a simulation perspective, however, the end-goals of such bombardment can be achieved more elegantly through the notions of optimistic temporal and spatial adaptivity. Such a system will not be required to recover from any failures or revisit an earlier stage in the computation.

The list of adaptivity schemas below (see Table 1) is merely meant to be indicative of the power of our runtime systems framework and the coverage is not intended to be exhaustive.

Table 1: Adaptivity schemas currently supported in our research.

Adaptivity Schema	**Example Application Context**
Staged Composition	Compositional PDE Solver Selection
Adaptation of Problem Decompositions	Numerical Quadrature, Adaptive Sorting
Coordinated Problem Solving	Interface Relaxation Algorithms
Algorithm Switching	ODEs, Number Factoring

[1] Java implements dynamic classloaders through its VM. Compiled OOP languages such as C++ can do late binding of function calls at runtime, but the target of the call has to be compiled into the executing binary. Weaves supports source-language independent late binding, including cases where the target of the call is dynamically loaded and linked.

Control Systems	Deriving Controllers for Algorithm Speedups
Active Mining of Recommendation Spaces	Qualitative Assessment for Matrix Computations
Graphs of Models	Multi-paradigm Performance Profiling

4.1 Staged Composition

Staged composition addresses the *sequential* selection and execution of code modules in scientific computations. It is important in problem domains that are characterized by *partial observability*. In this schema, code fragments from a library are composed at runtime to satisfy various general and domain-specific constraints on their structure. For instance, in the PDEs domain, the code fragments would correspond to choices of discretizer, pre-conditioner, and linear system solver. Since information about application performance characteristics is often acquired during the actual computation, rather than before, staged composition is a necessary feature in many application domains.

A runtime recommender can use a model-based approach to prune the search space of code modules and scale its functionality to large domains. Specifically, the sequence of stages in a composition is captured using a Markov decision process and the utilities of states are directly estimated. Then, given an initial state, the runtime recommender would evaluate the various choices (of algorithm components) and choose the one that leads to the state with the highest utility.

4.2 Adaptation of Problem Decompositions

Many scientific computations are characterized by a recursive divide-and-conquer strategy, with algorithm selection happening at each level of the recursive invocation. Classical examples are adaptive numerical quadrature and adaptive sorting on parallel architectures. With the Weaves framework, the runtime recommender has the capability to backtrack both breadthwise and depthwise in the recursive function invocation history. This means that any form of branch-and-bound algorithm can be easily implemented. Notice that the breadth wise capability arises from the *parallel* compositional nature of the Weaves framework.

To curtail the potential explosive growth in space complexity, the runtime recommender must cleverly choose an intermediate representation that is indicative of the problem characteristics and, at the same time, can be cheaply evaluated when necessary. This is because at each backtrack point, the recommender has to make a judgement of code module and execution path. The choice of the intermediate representation is a domain-specific issue but we can give an indication of what it might look like. In the case of recommending numerical quadrature algorithms, it is of critical interest to assess features of the integrand such as the presence of a singularity, whether it is an end-point singularity, whether the integrand is smooth in the interval, and whether it exhibits an oscillatory behavior of non-specific type. These features are sometimes impossible to determine (e.g., when the integrand is provided only as a software routine). One solution approach is to first model the dynamic selection of quadrature nodes by a general purpose adaptive code such as QAGS [Piessens et al., 1983] and then use the layout of *these* nodes as the actual representation of the function. This requires that we employ optimistic temporal adaptivity in order to be able to successfully backtrack and later follow a suitable integration algorithm.

4.3 Coordinated Problem Solving

The collaborating PDE solvers application described earlier falls in this category. Here, adaptivity is the responsibility of one/some of the weaved code modules themselves (in this case, the mediators), and which coordinates the functioning of other code modules. Note that the structure of the composition – shared elements and multiple flows of control (see Fig. 1) - is naturally prone to single-cycle deadlock. While an implementation may be carefully instrumented to avoid deadlocks, the Weaves framework

enables us to use the natural, underlying, problem representation and rely on runtime systems support for deadlock detection and recovery. The discussion section contains details of this mode of operation.

4.4 Algorithm Switching

Algorithm switching refers to the case where the problem being solved remains the same but the currently executing algorithm has to be replaced with another, dynamically. This facility is critical in solving ODEs with both stiff and non-stiff components, solving certain categories of linear systems, and integer factoring. For instance, the ODEs underlying many biological cell cycle models alternate between being stiff and non-stiff several times over the region of integration. In addition, properties such as stiffness are really a facet of both the ODE and the algorithm used to solve it. Algorithm switching is relevant here because our understanding of the problem improves as the computation proceeds. LSODE [Petzold, 1983] is an example of a real scientific code that embodies an algorithm switching mechanism, but as mentioned earlier the switching procedure is hardwired. It is sometimes "overcautious" to prevent thrashing between the two categories of algorithms. This is because, since stepsize selection is dependent on error estimates, situations involving misleading estimates can cause either a premature termination of methods or a switch to an unstable method. A runtime recommender can more carefully assess the suitability of algorithm switching by taking into account problem characteristics and runtime information, not otherwise available to the basic ODE algorithm.

In other applications, algorithm switching is important because the initial choice of algorithm fails. Here, it is imperative that we are able to use results and byproducts from the first algorithm to "seed" subsequent algorithm recommendations. For instance, in crypto-challenges such as integer factoring [Silverman and Wagstaff, 1993], we might switch to the quadratic sieve algorithm when the elliptic curve method fails.

4.5 Control Systems

An algorithm control system can be modeled with various configurations of the runtime recommender in the problem solving loop. More fundamentally, many classical formulations of control systems can be realized in scientific codes. For instance, a simple form of derivative-based control was used by Hovland and Heath [Hovland and Heath, 1997] to achieve an adaptive control policy for the SOR (Successive Over-Relaxation) algorithm. This is shown to be more powerful than using a fixed one with the optimal value of the over-relaxation parameter.

Similarly, adaptive control formulations are common in solving ODEs and automatic quadrature. In the former, the problem of stepsize selection can be thought of as designing a suitable controller (P, PI, PD, or PID formulations) around the basic numerical approximation. Automatic quadrature algorithms embody control systems because they must inherently assess the suitability of their approximations by deriving error estimates (often using approximations of successive orders).

A runtime recommender system extends such control system formulations into the realm of *actor-critic* models; the *actor* is the recommender that makes selections of solution components and the *critic* captures the improvement in how the recommender is itself assessed. Both the actor and the critic are implemented as learning algorithms. As the critic is learning to exercise better judgement, the actor benefits from the improved assessments, leading to a closed-loop control system.

4.6 Active Mining of Recommendation Spaces

In assessing many recommendation spaces, it is important to selectively sample and actively collect data, for the sole purpose of improving the confidence in the recommendation. For instance, in qualitative assessment of Jordan forms [Ramakrishnan and Bailey-Kellogg, 2002], data points are actively collected

at specific perturbations in order to determine the most probable Jordan form of a matrix. This adaptivity schema iterates between a code execution (for collecting a data point), refining the recommendation (another code execution), and repeating these steps until a desired functional is minimized. This idea is a central ingredient of the US National Science Foundation's recent thrust for Dynamic Data-Driven Application Systems (DDDAS; [Darema, 2002]).

4.7 Graphs of Models

In this final adaptivity schema, adaptivity is itself factored as operations on a graph and the task of runtime recommendation reduces to traversing this graph, to achieve user-specified criteria. For instance, in the performance modeling of the Sweep3D code ([Koch et al., 1992]; a benchmark for discrete-ordinates neutron transport), codes are available for analytical modeling, low-level simulation, and actual system execution [Adve et al., 2000]. Each node in the "graphs of models" corresponds to one model family, and the edges denote conditions and constraints to be satisfied (or achieved) when switching from one model to another. Consider two scenarios of Sweep3D modeling: one might (i) model the machine parameters accurately, taking into account processor components, memory components (buffers etc.) and transport components (interfaces to caches), or (ii) one might replace all machine parameters by picking one of the analytical models. Thus, moving from (i) to (ii) in the models graph might take place under the constraint that over 65% of the parts of the composed application need to be removed. Given end-to-end performance constraints, the runtime recommender then attempts to perform a means-end analysis on the induced graph, leading to a *satisficing model sequence*, that involves both models and the edges connecting them (notice that there may be more than one edge between two model choices). Preliminary results for this application are reported in [Houstis et al., 2002].

5 Early Results

5.1 Weaves: Implementation and Evaluation

The core of the Weaves compositional framework is the abstraction of a weave, which allows an application composer to define arbitrary namespaces over a composed application. To implement the namespace abstraction, we analyze the Executable and Linking Format (**ELF**) object files produced by any compiler. ELF is a public domain file format used to represent both object code as well as the final executable on most UNIX systems. Our current prototype is implemented on the Linux operating system running on Intel x86 architectures. Since the implementation only depends on the ELF file format, it can be easily ported to other operating systems/architectures. Furthermore, we anecdotally note that the features of the ELF file format used by our implementation are common to object file formats. Hence, it should be possible to extend the prototype to support other object file formats as well.

We ran a series of experiments to compare the context switch time under the threads, processes and weaves programming models. In this experiment, we created a baseline application that implements a calibrated delay loop (busy wait). We then implemented threads-, processes-, and weaves- versions of the application. In each of these versions, there are n independent flows of control over the same code, where each flow of control executes a calibrated delay loop, which does $1/n^{th}$ the work of the baseline application. We then measure the total time taken to execute the application under each of these models. Since each of the control flows does $1/n^{th}$ of the work and there are n flows, the total time taken should the same as the baseline calibrated delay loop case, except for an additional context switching cost.

Figure 5 shows the results of the experiment on a single processor AMD Athlon™ workstation running the Linux operating system. The results show the run time for five cases: (a) baseline calibrated delay loop, (b) pthreads threads library, (c) Pth threads library, (d) processes, (e) Weaves over pthreads, and (f) Weaves over Pth. The results clearly show that the weaved implementations are significantly faster than

processes, even in this simple case, where the copy-on-write semantics of the *fork()* call are very effective. Furthermore, the run time of weaved implementation of pthreads is very close to the base run time of pthreads alone. The marginal variation in runtime is due to the slightly higher weave creation cost, which is included in the run time. Also, the pthreads implementation is relatively efficient, since the Linux kernel includes operating system support for it.

However, in the case of Pth, the run time of the weaved implemented is higher than the base Pth case. This increase in runtime is because unlike pthreads, Pth is a user-level library and hence suffers from timer inaccuracies inherent in user-level library implementation.

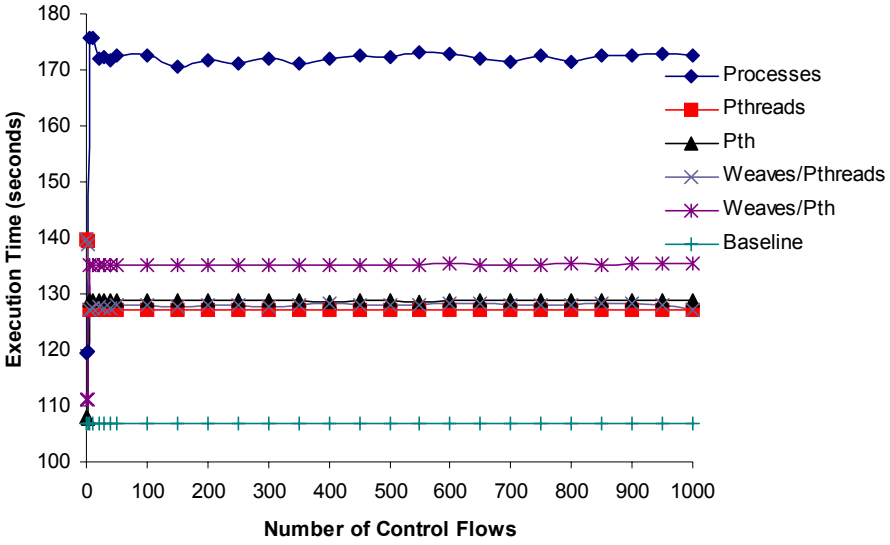

Figure 5: Comparison of inter-flow context switch time in the threads, processes, and weaves programming models. The baseline single process application implements a calibrated delay loop of 107 seconds.

5.2 Experiments in Adaptive Runtime Composition

A number of scientific applications have been or are currently being created using the runtime systems support framework described in this paper. These include:

- A weaved version of the Sweep3D code suitable for performance characterization
- A runtime recommender system for adaptive numerical quadrature
- Compositional PDE solvers for multi-domain, multi-physics problems
- Iterative assessment of spectral portraits of matrices by active mining
- Adaptive ODE algorithm switching for simulating biological cell cycle models
- Dynamic selection of linear system solvers for molecular electronics simulations

Due to space considerations, we describe the basics of programming adaptivity in the context of the numerical quadrature application (embodies the "adaptation of problem decompositions" schema) and the Sweep3D application (embodies the "graphs of models" schema). In the former, we illustrate the operations of the recommender in concert with Weaves, and in the latter, we demonstrate how the Sweep3D application has been weaved and an assessment of its performance characteristics. The reader should keep in mind the larger context in which such a performance model can then be used to drive the characterization of large-scale scientific applications.

5.2.1 Quadrature Application

Our first application centers on providing runtime systems support in the context of the GAUSS quadrature recommendation system [Ramakrishnan et al., 2002]. Consider the numerical integration problem from [Piessens et al., 1983]:

$$\int_0^1 \frac{4^{-\alpha} dx}{(x-\pi/4)^2+16^{-\alpha}}$$

This integrand has a peak of height 4^{α} at $\pi/4$, causing difficulties to different algorithms for different values of α. We begin with a generic main program in FORTRAN that prepares the suitable initializations and invokes algorithm DQAGS, an adaptive integration routine that itself invokes multiple quadrature rules (DQAGSE followed by Gauss-Kronrod routines). It is observed that DQAGS performs well for values of α upto 10. When α is increased beyond this point, DQAGS fails with an error code indicating a divergent integral (IER=5). We now describe how the recommender system and Weaves interact to adaptively rewire the application.

Notice that when the main program is entered, a base checkpoint would be made (just before invoking DQAGS). Within each function invocation inside DQAGS, incremental checkpoints are made which support two important abstractions: each call to the checkpoints returns a handle, or the handles are maintained with a stack abstraction. When an internal routine fails (i.e., with IER=5), two forms of adaptivity take place. First the state of the application is set to just before the last invocation of the failed routine (here, DQAGSE or even DQAGS). The checkpointing system also exposes the tuple-space view of namespace entries from the future that should not be restored (this enables us to "selectively forget the future without repeating it").

More importantly, the runtime recommender is passed the latest activation record of DQAGS as a stack object from which it picks out the needed information (note that in ELF analysis, we have lost type information, hence the stack object only has a collection of data pointers and associated lengths – to retrieve desired objects, type information is rebound by the recommender, enabling it to operate with a namespace abstraction rather than an address space abstraction). The GAUSS recommender uses the IER and NEVAL metrics to suggest to try out algorithm DQAG (i.e., spatial adaptivity). Algorithm DQAG is inserted dynamically into the runtime binary and references to DQAGS are substituted through the PLT (see Section 5.1) with DQAG. In this manner, the recommender directs the flow of the composed application using the primitives supported by Weaves.

Note that DQAG has the same type signature as DQAGS; this need not be the case in general. In order to properly prepare the calls to new procedures, the runtime system provides modifiable "continuation hooks" that are invoked before function calls. These continuation hooks expose the parameters of the original function call to the recommender, which can use domain-specific knowledge to massage/prepare the function invocation to the newly selected routine, to be invoked by the continuation. The return type can also be similarly massaged.

5.2.2 Sweep3D

Sweep3D is an ASCI benchmark for discrete ordinates neutron transport. Available in FORTRAN code in the public domain, it forms the kernel of an ASCI application and involves solving a group of time-independent neutron particle transport equations on a XYZ cartesian cube [Koch et al., 1992]. The code uses a logical discretization of the 3D geometry, taking care to ensure that physical symmetries are not

distorted, angular dependencies are preserved, and derivatives w.r.t. the angular coordinates are maintained.

The main characteristic of Sweep3D is that it uses no global variables. Since the application only relies on local state, multiple instantiations of local state should be enough to create a VM abstraction. This characteristic makes Sweep3D inherently thread-safe, which enables its modeling by either the threads or process models. However, since the application is written in Fortran 77, with dynamic array extensions, modeling with the threads and processes models present interesting implementation problems. *While trying to model the application using POSIX threads, we found that there was substantial global state in the .data section of the ELF executable, a Fortran compiler issue, which essentially made the code-base "thread unsafe"*. Weaving the Sweep3D code-base created independent namespaces, resulting in a thread-safe version.

To support the message passing primitives used by Sweep3D, we created a simple threaded MPI emulator, which implements only the **nine** MPI primitives used by Sweep3D. To ensure correctness, the MPI emulator implementation follows the guidelines set forth in the MPI specification. Our MPI emulator is intended as a test prototype and is neither as comprehensive nor as capable as a complete MPI implementation.

In the weaved implementation, we create *n* distinct virtual machines, each of which executes an independent instantiation of the Sweep3D application. To do this, we create *n* distinct Sweep3D beads and *n* weaves, where each weave has a distinct Sweep3D bead and a shared emulator bead. Each weave also has a single string associated with it. The *n* distinct virtual machines run on a single processor workstation.

We compared the performance of our single processor weaved implementation of Sweep3D against measured values from real runs for up to 150 processors. Measurements for the real runs were made on our 200 processor cluster (1GHz AMD Athlon ™ processors over Myrinet™) *Anantham*. Since the Sweep3D application performs its own timing measurements, we compared the timing numbers (CPU Time) of the weaved version of Sweep3D with the measurements from actual runs. The two input files (50x50x50 and 150x150x150 decompositions) provided in the Sweep3D distribution were used to drive the Sweep3D application.

For upto 150 processors, the timing results from the weaved implementation and the actual runs were consistent to within 0.2%. Furthermore, we tested the weaved version of Sweep3D with over 1000 weaves on a single processor. The variation in the timing results between multiple runs was within 0.2%. This clearly shows that even at high levels of scalability (over 1000 weaves/processor) context switch time does not impact the efficacy of our runtime compositional framework.

6 Discussion

This paper has described a novel runtime compositional system for supporting adaptive scientific computations in PSEs. Weaves serves as a true generalization of the threads and processes models of programming and provides immediate benefits in object-based composition, checkpointing, migrating, and dynamic reconfiguration of scientific applications. Runtime recommender systems encapsulate knowledge about which solution components perform well (and for which situations) and provide intelligent decision support for configuring and managing large-scale computations. Together, they constitute a powerful mode of developing and deploying adaptive scientific applications.

The work presented here has interesting parallels to research in many different areas – we survey a collection of references topically. At a basic level, Weaves's capabilities as a programming model can be

compared to that of distributed OO [Gannon and Grimshaw, 1998], parallel programming primitives [Foster, 1996; Skillicorn and Talia, 1998], agent-based composition [Drashansky et al., 1999], and service-based systems integration [Foster et al., 2002]. The design of the Weaves system bears a strong resemblance to the OO framework of Mentat propounded in [Grimshaw et al., 1996]. However, unlike Mentat, which requires creating code objects in an OO language, Weaves can create an object based framework from code written in any language, allowing the reuse of the vast repository of legacy codes.

Significant research has also been conducted to realize adaptivity in distributed scientific computations are well studied, e.g., in the contexts of performance modeling [Vraalsen et al., 2001; Adve et al., 2002], application tuning [Chang and Karamcheti, 2001], and meta-modeling and control [Ribler et al. 2001; Kennedy et al., 2002]. Many of these applications are focused on selecting system configurations, identifying optimal application parameters, and exploiting opportunities for application scheduling over the Grid. The notion of runtime recommendation presented here applies more broadly to selecting algorithms and code modules, and the knowledge-based framework allows application-specific context about the suitability of algorithms to be exploited. The algorithmic framework used for runtime recommendation (namely, reinforcement learning) is very powerful, and is part of a larger family of strategies for adaptive control of algorithm executions.

As HPC infrastructure (e.g., the Grid) improves and newer applications are explored, we believe the importance of PSE programming primitives will be better appreciated. It will be especially crucial that the programming primitives allow rich forms of adaptivity to be specified and captured without the need for low-level system configuration. There are many recent steps taken in this direction (e.g., the compiler directed frameworks described in [Adve and Sakellariou, 2000]). The central idea here is to encode adaptivity as operations on a suitably defined *task graph*, which serves as an intermediate representation of the dynamic behavior of an application. In addition to operationalizing adaptivity, such a representation allows systematic performance characterization of scientific applications using multiple methodologies [Browne et al., 2000]. In our work, the intermediate representation is the purview of the runtime recommender but dynamic operations of spatial and temporal adaptivity are handled by the checkpointing and composition framework supplied by Weaves. We are currently in the process of defining a language for declaring search primitives (akin to a branch-and-bound operation for optimistic simulation) that can be used as building blocks of adaptivity. The advantage with this formulation is that adaptivity is taking place at the level of code modules and hence can be made as coarse or fine grained as necessary. It also allows for ease of specification by the PSE application composer.

The eventual success of a PSE will lie in "what it lets you get away with." By factoring support for adaptivity in a runtime recommender system and operationalizing parallel composition, checkpointing, and migration using the Weaves framework, the ideas presented here allow us to transparently realize the promise of adaptive PSE applications.

Acknowledgements

This work is supported in part in US National Science Foundation grants EIA-9974956, EIA-9984317 (CAREER), EIA-0103660, and EIA-0133840 (CAREER).

7 References

V.S. Adve, A. Akinsanmi, J.C. Browne, D. Buaklee, G. Deng, V. Lam, T. Morgan, J.R. Rice, G. Rodin, P. Teller, G. Tracy, M.K. Vernon, and S. Wright, Model-Based Control of Adaptive Applications: An Overview, in *Proceedings of the Next Generation Software Workshop, International Parallel and Distributed Processing Symposium (IPDPS'02)*, Fort Lauderdale, FL, Apr 2002.

V.S. Adve, R. Bagrodia, J.C. Browne, E. Deelman, A. Dube, E.N. Houstis, J.R. Rice, R. Sakellariou, D.J. Sundaram-Stukel, P.J. Teller, and M.K. Vernon, POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems, *IEEE Transactions on Software Engineering*, Vol. 26, No. 11, pages 1027-1048, Nov 2000.

V.S. Adve and R. Sakellariou, Application Representations for Multi-Paradigm Performance Modeling of Large-Scale Parallel Scientific Codes, *International Journal of High Performance Computing Applications*, Vol. 14, No. 4, pages 304-316, Winter 2000.

R. Barrett, M. Berry, J. Dongarra, V. Eijkhout, C. Romine, Algorithmic Bombardment for the Iterative Solution of Linear Systems: A Poly-Iterative Approach, *Journal of Computational and Applied Mathematics*, Vol. 74, No. 1-2, pages 91-109, 1996.

J.C. Browne, E. Berger, and A. Dube, Compositional Development of Performance Models in POEMS, *International Journal of High Performance Computing Applications*, Vol. 14, No. 4, pages 283-291, Winter 2000.

H. Casanova and F. Berman, Parameter Sweeps on the Grid with APST, *Concurrency and Computation: Practice and Experience*, Chapter 33 of Grid Computing: Making the Global Infrastructure a Reality (F. Berman, G. Fox, and T. Hey, editors), Wiley, Feb 2003.

F. Chang and V. Karamcheti, A Framework for Automatic Adaptation of Tunable Distributed Applications, *Cluster Computing: The Journal of Networks, Software, and Applications*, Vol. 4, No. 1, pages 49-62, 2001.

F. Darema, Dynamic Data-Driven Application Systems, in *Process Coordination and Ubiquitous Computing*, D.C. Marinescu and C. Lee (editors), CRC Press, 2002.

T.T. Drashansky, E.N. Houstis, N. Ramakrishnan, and J.R. Rice, Networked Agents for Scientific Computing, *Communications of the ACM*, Vol. 42, No. 3, pages 48-54, March 1999.

I. Foster, Compositional Parallel Programming Languages, *ACM Transactions on Programming Languages and Systems*, Vol. 18, No. 4, pages 454-476, July 1996.

I. Foster, C. Kesselman, and S. Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organizations, *International Journal of Supercomputer Applications*, Vol. 15, No. 3, pages 200-222, Fall 2001.

I. Foster, C. Kesselman, J. Nick, and S. Tuecke, Grid Services for Distributed Systems Integration, *IEEE Computer*, Vol. 35, No. 6, pages 37-46, June 2002.

A.S. Grimshaw, J.B. Weissman and W.T. Strayer, Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing, *ACM Transactions on Computer Systems*, Vol 14, No. 2, pages 139-170, May 1996.

E.N. Houstis, A.C. Catlin, J.R. Rice, V.S. Verykios, N. Ramakrishnan, and C.E. Houstis, PYTHIA-II: A Knowledge/Database System for Managing Performance Data and Recommending Scientific Software, *ACM Transactions on Mathematical Software*, Vol. 26, No. 2, pages 227-253, June 2000.

E.N. Houstis, A.C. Catlin, N. Dhanjani, J.R. Rice, N. Ramakrishnan, and V.S. Verykios, MyPYTHIA: A Recommendation Portal for Scientific Software and Services, *Concurrency and Computation: Practice and Experience*, Vol. 14, No. 13-15 (Special Issue on "Grid Computing Environments"), pages 1481-1505, Dec 2002.

P. Hovland and M. Heath, Adaptive SOR: A Case Study in Automatic Differentiation of Algorithm Patameters, *Technical Report ANL/MCS-P672-0697, Argonne National Laboratory*, 1997.

K. Kennedy et al., Toward a Framework for Preparing and Executing Adaptive Grid Programs, in *Proceedings of the Next Generation Software Workshop, International Parallel and Distributed Processing Symposium (IPDPS'02)*, Fort Lauderdale, FL, Apr 2002.

K.R. Koch, R.S. Baker, and R.E. Alcouffe, Solution of the First-Order Form of the 3D Discrete Ordinates Equation on a Massively Parallel Processor, *Transactions of the American Nuclear Society*, Vol. 65, No. 198, 1992.

H.S. McFaddin and J.R. Rice, Collaborating PDE Solvers, *Applied Numerical Mathematics*, Vol. 10, pages 279-295, 1992.

R. Piessens, E. de Doncker-Kapenga, C.W. Uberhuber, and D.K. Kahaner, *Quadpack*, New York, Springer, 1983.

N. Ramakrishnan and C. Bailey-Kellogg, Sampling Strategies for Mining in Data-Scarce Domains, *IEEE/AIP Computing in Science and Engineering,* Vol. 4, No. 4 (Special issue on "Data Mining in Science"), pages 31-43, July/Aug 2002.

N. Ramakrishnan, E.N. Houstis, and J.R. Rice, Recommender Systems for Problem Solving Environments, in *Working Notes of the AAAI-98 Workshop on Recommender Systems*, H. Kautz (editor), pages 91-95, AAAI/MIT Press, 1998.

N. Ramakrishnan and C.J. Ribbens, Mining and Visualizing Recommendation Spaces for Elliptic PDEs with Continuous Attributes, *ACM Transactions on Mathematical Software*, Vol. 26, No. 2, pages 254-273, June 2000.

N. Ramakrishnan, J.R. Rice, and E.N. Houstis, GAUSS: An Online Algorithm Selection System for Numerical Quadrature, *Advances in Engineering Software*, Vol. 33, No. 1, pages 27-36, Jan 2002.

R.L. Ribler, H. Simitci, and D.A. Reed, The AutoPilot Performance-Directed Adaptive Control System, *Future Generation Computer Systems*, Vol. 18, No. 1, pages 175-187, 2001.

J.R. Rice, P. Tsompanopoulou and E.A. Vavalis, Interface Relaxation Methods for Elliptic Partial Differential Equations, *Applied Numerical Mathematics*, Vol. 32, pages 219-245, 1999.

R.D. Silverman and S.S. Wagstaff, A Practical Analysis of the Elliptic Curve Factoring Algorithm, *Mathematics of Computation*, Vol. 61, No. 203, pages 445-462, 1993.

D. Skillicorn and D. Talia, Models and Languages for Parallel Computation, *ACM Computing Surveys*, Vol. 30, No. 2, pages 123-169, June 1998.

R.S. Sutton and A.G. Barto, *Reinforcement Learning*, MIT Press, 1998.

F. Vraalsen, R.A. Aydt, C.L. Mendes, and D.A. Reed, Performance Contracts: Predicting and Monitoring Grid Application Behavior, in *Proceedings of the 2nd International Workshop on Grid Computing*, pages 154-165, Nov 2001.