Program Control Language: A Programming Language for Adaptive Distributed Applications ⋆

Brian Ensink *, Joel Stanley, Vikram Adve

University of Illinois at Urbana-Champaign,
Computer Science Department, 1304 W. Springfield Avenue, Urbana IL 61801.

Abstract

Many distributed applications must meet stringent performance requirements even when the performance characteristics of the underlying systems and networks vary significantly at runtime. Runtime adaptation can be used to tolerate such changes, but sophisticated adaptive distributed programs can be extremely challenging to design, implement and debug. This paper proposes a language called Program Control Language (PCL) that provides a novel means of specifying adaptations in distributed applications. PCL is based on an abstract, global representation of a distributed program (a static task graph), which enables a programmer to *reason about* and to *describe* a wide range of application-specific adaptation strategies at a high level, using a few key mechanisms. PCL provides simple high-level syntax for local and remote adaptation operations, local and remote performance monitoring (and aggregation), and for performing adaptations synchronously or asynchronously with respect to the execution of the application process initiating the adaptation. The global task graph representation enables remote performance metrics and adaptation operations to be specified in simple global terms by any process, and the compiler and runtime system automatically perform the communication and synchronization required for the remote operations. The paper describes the conceptual adaptation framework, the PCL language, and our implementation of the PCL compiler and runtime system. The paper uses three adaptative applications examples to illustrate the capabilities and benefits of PCL, and to show experimentally that the performance overheads of using PCL for implementing an adaptive application are negligible.

Keywords: adaptive, computational grid, distributed, parallel computing, performance metric, program control language, runtime system, task graph

1 Introduction

Many distributed applications on a Computational Grid [1] have to run on widely shared systems and networks, and the performance characteristics of these systems can change significantly during program execution. Such applications have to be able to meet stringent performance requirements in the presence of such dynamic changes in runtime conditions. Some examples of application classes that face such a challenge include high performance parallel applications running on a Computational Grid [1], distributed multimedia applications that must maintain some formal or informal quality-of-service metric (e.g., frame rate for a distributed video server), applications for mobile hand-held devices operating under tight and variable power and bandwidth constraints, and perhaps distributed Web servers with variable loads and varying access patterns.

A common strategy used to achieve performance goals under changing operating conditions is to use *runtime adaptation*: making *runtime* changes to the behavior of the application or underlying libraries in response to changing operating conditions. Such "adaptive" changes may be internal changes in parameter settings, algorithms (or components of algorithms), data representations, or load-balancing strategies; or external changes to program configuration such as changing servers, or task migration.

There is extensive research on supporting adaptive applications at the level of the operating system [2–4], network[5–8], and especially runtime systems or middleware [9–16,2]. These systems provide valuable runtime services including performance monitoring, resource allocation, resource reservation, and domain-specific configuration options. Even with such support, however, adaptive strategies can be very complex to design, specify, and debug within the application or appliction class. Some key challenges include the increased complexity of the applications, the need for developing and implementing application-specific or domain-specific performance estimation techniques, the need for coordinating the adaptation of multiple processes in a distributed program, and the lack of support for analyzing correctness properties of the adaptations. More generally, runtime systems do not by themselves provide a framework with which to reason about the effect of adaptation operations on program behavior.

The broad goal of our work is to provide programming language and compiler support for adaptive distributed applications, in order to simplify the development

⋆ This work was sponsored by the NSF Next Generation Software program under contract number EIA-9975024. It was also supported in part by an NSF CAREER award number EIA-0093426, by the NSF Operating Systems and Compilers program under grant number CCR-9988482, by DARPA/ITO under contract number N66001-97-C-8533, and by an IBM SUR grant to the University of Illinois.
* Corresponding author.
Email addresses: ensink@cs.uiuc.edu (Brian Ensink), jstanley@cs.uiuc.edu (Joel Stanley), vadve@cs.uiuc.edu (Vikram Adve).

of such applications. Towards this end, we have developed a programming model for adaptation and a programming language based on that model. The language, Program Control Language or PCL, consists of a small set of extensions to a base language (such as C, C++, Java, or Fortran) that allow the programmer to write high-level code to "control" the behavior of a target program at runtime. Together, the programming model and the language provide a number of potential benefits in designing and implementing adaptive distributed programs:

- *An abstract framework for reasoning about and describing adaptation strategies*: The framework we propose is based on a Static Task Graph (STG), which provides a general, abstract representation of a distributed computation, including the local computations (tasks), control flow, and the distributed structure [17–19]. The framework provides a small number of task graph manipulation mechanisms that can be used to describe a wide range of adaptation strategies. (This is important because adaptation techniques vary widely from one application to another, and are often based on highly application-specific or domain-specific information [20,21,16,2].)

- *A global view of the distributed computation, with global specification of adaptation operations across multiple processes*: The STG provides a single logical view of the entire distributed computation, including the tasks and control flow of all processes. This allows "distributed adaptation", i.e., one that requires adaptive changes within multiple processes, to be specified by a single process simply as operations on this task graph, without any explicit remote operations. The compiler and runtime system automatically perform the remote communication required to perform these adaptations on each relevant process.

- *Specification of performance metrics and events*: The PCL language provides syntax to identify the metrics used to guide adaptation (with associated data gathering functions), and events defined in terms of those metrics for triggering adaptation asynchronously when desired. As with adaptation operations, any process can refer to performance metrics for itself or other processes in global terms using the task graph and process identifies. Metrics for remote processes and aggregate statistics across processes are automatically retrieved by the compiler using operations from the runtime system.

- *Specification of coordination criteria for "distributed" adaptations*: While individual adaptation operations on remote processes can be performed automatically by the compiler and runtime system, an adaptation that involves adaptive changes within multiple processes must be carefully coordinated to ensure correctness. In recent work (outside the scope of this paper), we have shown that the programmer can specify coordination criteria as directives in the code expressed in simple global terms using the task graph [22]. Using these criteria, the compiler and runtime system can automatically perform the communication and synchronization required to coordinate the operations that must be performed by different processes. These criteria also express when local adaptation operations can or cannot happen with respect to the computations (tasks) of a single process. Together, these coordination criteria can significantly simplify the programming ef-

fort required to design and implement a complex distributed adaptation involving multiple processes.

Together, PCL provides a global programming model for specifying adaptation, including the performance metrics and events that trigger adaptation, the program modifications that implement the adaptation, and correctness criteria for coordinating adaptation in the context of the target computation. While some of these capabilities could be provided directly by runtime systems or middleware, some of the most fundamental ones cannot. In particular, language and compiler support are essential for providing a global abstraction of the distributed programming (the task graph) and supporting operations on the task graph. This abstraction is fundamental to the first, second and fourth capabilities in the list presented above.

It is also important to note that many of our adaptation mechanisms (including all those in our example codes) only require identifying a few relevant tasks affected directly by adaptations, and do not require extracting or describing other parts of the task graph. The Static Task Graph provides the conceptual model for reasoning about and specifying adaptation, but is needed only in very limited ways by the compiler and generated code. This makes it very straightforward to use PCL for existing applications and development environments, while still benefitting from the abstract adaptation model provided by the language.

We have built a compiler for PCL using C as the base language, using the LLVM infrastructure [23] developed in our group at Illinois. We have also built a PCL runtime system that is used by the compiler generated code. The PCL compiler and runtime system support local and remote adaptation operations specified via the STG, and local and remote performance metrics. (Events are not yet supported by the system.) Furthermore, the task graph adaptation mechanisms are implemented entirely via static code generation (i.e., by transforming the application source code), and do not require any runtime representation of task graphs or runtime code generation. This greatly reduces the complexity and of the runtime system.

A key aspect of PCL is an asynchronous strategy for gathering remote performance metrics and performing remote adaptation operations. A major potential performance problem can occur for adaptations that use performance metrics and adaptation operations on remote processes on wide-area networks: every adaptation attempt may require one or more round-trip messages to gather performance data, plus additional round-trip messages if a decision is taken to adapt. The PCL language provides an "asynchronous invocation" directive for adaptation, and such invocations are then executed in a parallel thread within the runtime system without blocking the invoking user-level thread. This has two potential benefits: it can greatly reduce the blocking time for adaptation operations involving remote processes, and it ensures that the remote communication operations for adaptation do not interfere with the communication operations or parallelism of the target application.

We present three example applications that were made adaptive using PCL, and we discuss in qualitative terms the potential benefits of using PCL for these ap-

pliactions. These applications are a simple parallel fractal code, a parallel version (MPIPOV [24]) of a sophisticated ray tracing code called Povray [25], and an iterative PDE solver with a adaptation strategy applicable to many domain decomposition codes [21]. We present several experiments that use these codes to evaluate the potential costs of the PCL mechanisms in terms of the runtime overhead for remote monitoring and adaptation. (Note that it is explicitly *not our goal* to evaluate the potential benefit of specific adaptation strategies since that has little to do with the use of PCL and depends almost entirely on properties of the specific applications and the adaptation strategies used.)

Our use of PCL for these examples shows that it is feasible to write simple control code to gather remote metrics and to perform local and distributed adaptation operations, and to do so while keeping the control code separate from the target application code. Our experiments with the PCL versions of these codes show that the runtime overhead introduced by the PCL runtime system (including performance monitoring and adaptation operations) is negligible, amounting to less than 1-2% of execution time. Furthermore, the asynchronous strategy for retrieving remote metrics and performing adaptations greatly outperforms a more straightforward synchronous strategy for two out of three of these codes. Overall, our results show that with the asynchronous strategy, there is no noticeable performance penalty to using PCL for monitoring and controlling adaptation.

The next section of the paper motivates and describes the task graph framework for adaptation. The following sections describe the PCL language, the PCL compiler and runtime system, and our experimental results. We then discuss related work. Finally, we conclude with a summary and a brief description of our plans for further work on PCL.

2 A Conceptual Framework for Adaptation

Informally, the term "adaptation" in this work refers to any program-specified change in behavior at runtime in response to external operating conditions or external input data. We do not include internal adaptations that occur within many algorithms in response to internally computed results, e.g., changing mesh resolutions in an Adaptive Mesh Refinement code. In practice, the former kind of adaptation usually involves occasional but relatively gross changes, whereas the latter are fine-grain decisions that can happen continuously during execution.

As noted in the Introduction, we have been able to develop a framework with a small number of program manipulation mechanisms that can be used to implement a wide range of adaptation strategies. The framework is based on Static Task Graphs, which can be thought of as a generalization of a control flow graph (CFG) to include information about distributed structure. By (conceptually) modifying a program's static task graph, the behavior of a program can be changed *at runtime* in well-specified ways. Furthermore, these changes can be programmed at a high level,

leaving many implementation details to a compiler and runtime system.

The adaptation *framework* we propose is potentially useful outside the context of the existing PCL language and compiler, because it can provide a useful abstraction for designing adaptation strategies and reasoning about their impact on program behavior. Therefore, we first describe the framework and then separately describe its realization in PCL.

2.1 Static Task Graphs

A *task* is a sequence of instructions in a program containing no internal parallelism and no internal synchronization operations, such that control always enters the task at the first instruction and leaves at the last one. Logically, one or more instances of a task are created and executed at runtime. The constraints on a task imply that each task instance must be executed by a single thread to completion (in the absence of errors or external interrupts), and any precedence relationship between a pair of task instances arises only at task boundaries. There are three types of tasks, namely, computation, control-flow and communication tasks, representing instructions executed for ordinary computations, for a control-flow branch (e.g., an IF statement or loop header), or for a communication operation respectively. A task may contain internal control-flow (e.g., an entire loop nest), which is useful because coarse-grain tasks are often sufficient for adaptation strategies in practice. Thus, a single task may include multiple basic blocks of the control flow graph of the program.

A *static task graph* (STG) is a graph in which each node represents a task and each edge represents a precedence relationship between a pair of tasks. The precedence may represent either ordinary control-flow or a synchronization operation in the program. Note that data transfer between threads (e.g., for a message) is not explicitly captured in the graph, but is implicitly represented via communication tasks (this choice is not essential to our work). (A variant of the STG could include data transfer edges or combine synchronization edges with explicit data transfer information, but this choice is not essential to our work.) Note that a task graph can be thought as a control-flow graph (CFG) plus synchronization operations, but it will typically use tasks that represent entire subgraphs of a traditional CFG (i.e., more coarse-grain than individual basic-blocks). Figures 1, 2 and 3 show the static task graphs for a distributed video tracking code, MPIPOV and ATR, which are discussed further below.

The STG is an implicit property of the target distributed program, analogous to a CFG for a sequential program. Just as with the CFG, different STGs for a program are possible where each graph contains tasks with different granularities of computation.

We assume that the target distributed program is executed by a collection of processes with unique identifiers, $0 \leq i$, each with a local address space and a local copy

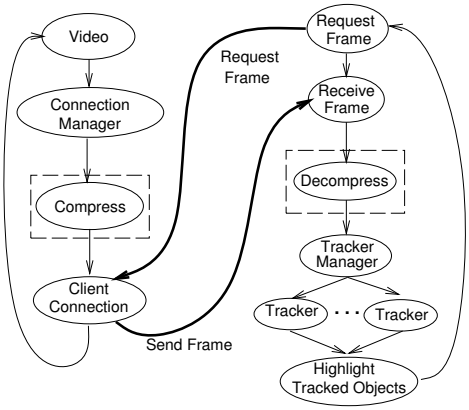

Fig. 1. Static task graph for distributed video tracking (server on left; client on right).

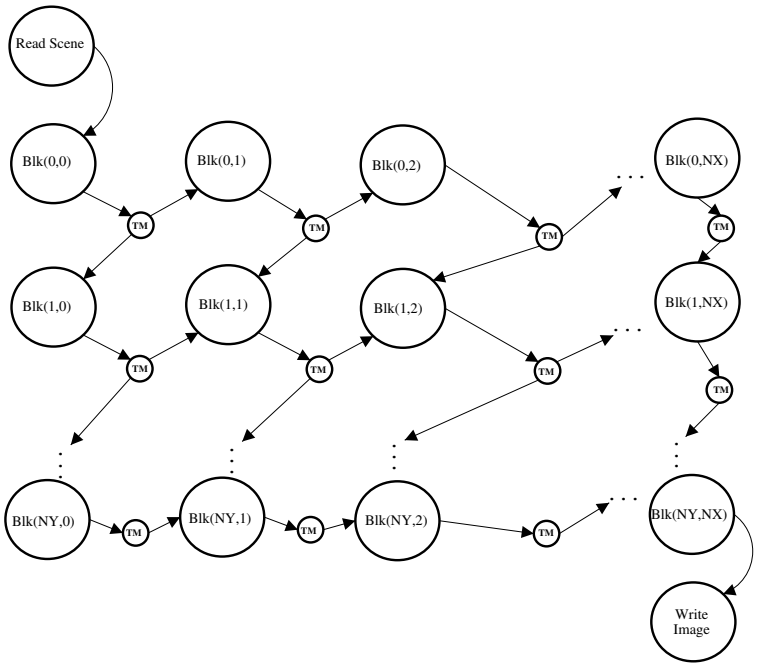

Fig. 2. Static task graph for MPIPOV

of the program text and data. (The process identifiers are purely a convenience; the processes may be dynamically created and destroyed.) A process may use multiple threads that share code and data. In practice, we create one instance of the PCL runtime system for each process.

The STG by itself does not fully specify program behavior. A second fundamental aspect of program behavior is the *task scheduling algorithm* used to allocate task instances to processes at runtime (and to threads within those processes in a multi-threaded program). The scheduling algorithm is a separate property from the STG, and is again implicitly defined by the program. A simple example of a dynamic scheduling algorithm is round-robin allocation from a centralized task queue; vari-

ants of this policy are used by many master-worker programs. The PCL language and compiler do not consider the scheduling policy and in particular do not provide any mechanisms to modify the policy. (Many adaptations that involve changes in task scheduling or allocation can nevertheless be specified in PCL as changes to program parameters.)

Note that the STG is a single "global" description of a distributed program because it describes the behavior of the complete program. Many adaptation strategies may need to apply different adaptation operations to different processes. Nevertheless, these conceptually modify the global task graph, with appropriate conditional branches that produce different behavior at different clients. For example, in the Fractal program presented in Section 5.1.1 an adaptation may decide that some subset of workers must send their completed data compressed to the master. This requires executing a compression task at particular workers in order to compress and send completed data messages, and a decompression task at a master for those workers. Conceptually, this compression task is inserted in the global STG, and it is guarded by conditional code that decides which workers need to execute that task. This conceptual view is important so that all processes logically operate on a single common STG representation of the distributed program, regardless of how adaptation may have modified the local behavior of the different processes.

2.2 Adaptation Operations in the Framework

The adaptation framework we propose consists of several primitive adaptation operations, and distributed coordination support for those operations (currently being developed). The adaptation operations are as follows (the use of the *pid* parameter for all the operations is explained below):

AddTask$(T_{new}, T_{pred}, T_{succ}, pid)$: Insert task T_{new} and add two new control-flow edges: $T_{pred} \longrightarrow T_{new}$ and $T_{new} \longrightarrow T_{succ}$. Remove the previous control-flow edge $T_{pred} \longrightarrow T_{succ}$ (it is illegal for such an edge not to exist).

RemoveTask$(T, T_{pred}, T_{succ}, pid)$: Remove task T and its incident edges from the STG, and add a new control-flow edge $T_{pred} \longrightarrow T_{succ}$.

ReplaceTask$(T, T_{new}, T_{pred}, T_{succ}, pid)$: Equivalent to *RemoveTask*(T, T_{pred}, T_{succ}) followed by *AddTask*$(T_{new}, T_{pred}, T_{succ})$, but guaranteed to happen atomically with respect to other operations on the STG.

AddEdge$(E_{type}, T_{pred}, T_{succ}, pid)$, $E_{type} \in \{ControlFlow, Sync\}$: Add a control-flow edge or a synchronization edge respectively from $T_{pred} \longrightarrow T_{succ}$.

RemoveEdge$(E_{type}, T_{pred}, T_{succ}, pid)$, $E_{type} \in \{ControlFlow, Sync\}$: Remove the $T_{pred} \longrightarrow T_{succ}$ of type control-flow or synchronization respectively. There must

be exactly one such edge.

ChangeParam($L_{val}, R_{val}[, T], pid$): Execute the assignment $L_{val} = R_{val}$ at the entry point to task T. T is optional and if it is omitted, the assignment is performed immediately (once) at the point at which the *ChangeParam* is invoked. In either case, L_{val} and R_{val} must be syntactically valid expressions at the point where the assignment is executed, and R_{val} must have no side-effects.

If $pid = -1$ is specified on an adaptation operation, that adaptation is performed globally, i.e., for the common static task graph of the program. As noted above, however, some adaptation operations may need to be applied only to specified processes, and broadcasting the adaptation to all processes in the distributed program would be quite inefficient. By specifying a value $pid \geq 0$, the programmer can ensure that the adaptation operation will only be applied to the specified process. Nevertheless, the conceptual semantics of the operation are as if it had been performed on the single common STG, using conditional branches to test the *pid* as necessary. The compiler and runtime system together ensure that these global semantics are preserved, even as adaptation operations are performed locally.

Adding a task to the STG effectively inserts a new computation at a specified location in the program text. Deleting a task removes a computation from a specified location; the T_{pred} and T_{succ} parameters ensure that the appropriate control-flow between a predecessor and successor are restored. (If there are multiple predecessors or successors, separate *AddEdge* operations may have to be used to add additional control-flow or synchronization operations). Adding or deleting an edge from the STG will typically occur in tandem with changes to the tasks in the STG, e.g., for adding a new message operation to the program. These operations may also be useful by themselves, e.g., for replacing a conditional branch with an unconditional one or for eliminating a synchronization operation.

The *ChangeParam* operation is intended to modify parameters that govern the behavior of one or more tasks. In practice, this will usually be used for values that are not modified within the task itself (although we do not impose this restriction). An example is the parameter B governing the IF task in ATR (Figure 3), which will affect the maximum number of concurrent iterations outstanding. Two points about this example are worth noting. First, it is sufficient to execute the assignment once in this case, and this behavior is sufficient in all the examples we have encountered. Executing the assignment repeatedly at task entry should only be needed if L_{val} or R_{val} need to vary in different instances of some task T. Second, the same effect could have been achieved by replacing the IF task with a new IF task that uses a different embedded value of B. It is more intuitive, however, to think about the adaptation as changing a parameter, and can be implemented more efficiently.

With this small set of operations, we have been able to express adaptive behaviors for several different applications from very different domains. We illustrate this here using three example applications: adaptive video tracking, ATR, and MPIPOV. Section 5 describes additional applications that are captured by our framework.

The video tracking application [16] is a client-server application representative of distributed multimedia applications that must preserve QoS targets in a dynamic environment. In this code, a server captures a live video stream off a camera (or disk for experimental purposes). The user specifies objects to be tracked and the client applies multiple tracking algorithms to track and display the location of the objects. The QoS metric the application aims to preserve is tracking precision, that is, the distance from the center of the tracking region and the center of the object being tracked. Figure 1 shows the static task graph of the video tracking application. This code adapts in two ways to preserve tracking precision under changing conditions [16]. When CPU loads at the client change, it can change the set of tracking algorithms used to achieve different speed/precision tradeoffs. When available bandwidth between the server and client varies, the client can request switch between raw video and compressed motion JPEG. The former adaptation can be implemented in the framework using *ReplaceTask*, *AddTask* or *RemoveTask* operations. For the latter adaptation, the code can switch from raw to compressed video by using *AddTask* to insert Compress and Decompress tasks at the server and client respectively. Switching back to raw video can be done using *RemoveTask*.

MPIPOV [24] is a master-worker ray tracing code that renders an image in parallel by decomposing it into an array of blocks. Figure 2 shows the static task graph (in an unrolled form), where each task labelled `Blk(y,x)` in the figure corresponds to the computations for rendering one block of the image. Tasks labelled `Read, Write, and TM (Task Manager)` are executed by the master. As blocks are rendered, con-

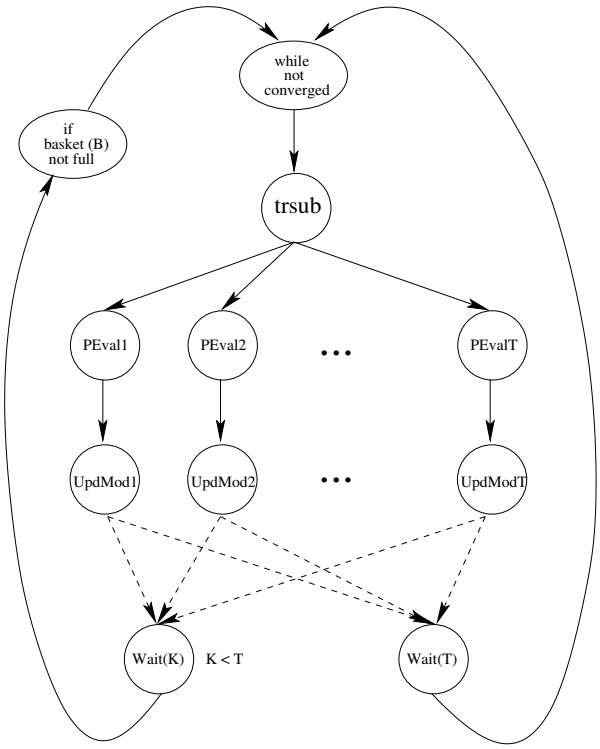

Fig. 3. Static task graph representation of asynchronous parallel stochastic optimization (ATR).

trol passes to a TM node, which then inserts one or more dependent blocks to a ready queue for workers to retrieve. A slow worker can hurt concurrency by limiting the number of blocks available to be assigned. The adaptive version of the code adjusts the task size assigned to each worker by changing the number of image blocks assigned to a worker in a single task. This adaptation can be implemented in the framework by using a *ChangeParameter* command to change an entry in a table that specifies the number of blocks for each worker.

The ATR [26] code is a very long running master/worker code that performs an asynchronous parallel stochastic optimization. Each iteration of this code begins by solving a (potentially large) linear program on the master, shown as task `trsub`. The master then creates T tasks labelled `PEval`i, which are enqueued in a task queue and retrieved and executed by a set of workers. After a task `PEval`i completes, its results are returned to the master, which then executes task `UpdMod`i, which updates an approximation (a "model") of the solution. The program can execute up to B iterations concurrently, where B is called the basket size. When $K < T$ tasks of an iteration are complete, the next iteration can be activated (i.e., its tasks inserted into the task queue) if the basket is not full, i.e., if less than B iterations are active. A new iteration is also activated when all T tasks complete. This code adapts to changing count of available processors by increasing or decreasing available parallelism, in two ways: (1) changing the number of concurrent iterations (i.e., the degree of asynchrony), and (2) changing the number of subtasks within each iteration. The first adaptation simply requires changing the basket size parameter, B, used by the IF task at the left of the figure. The second can be implemented by changing the parameter, T, used by the `trsub` task to govern the number of `PEval` and `UpdMod` tasks created at runtime.

2.3 Coordinating Distributed Adaptations

Coordination is important for distributed applications to control access to shared resources and data. Similarly, a single logical adaptation may require individual changes on multiple processes, and coordinating these adaptive changes is also important to maintain consistency and correctness before and after an adaptation takes place.

Consider the following example adaptations all of which require coordination at some level. For example, in the distributed video tracking example of Figure 1, the communication channel may contain several frames and partial frames in transit over the network. Two kinds of coordination requirements occur for the video compression adaptation. First, the client and server should switch formats prior to the same frame. Second, neither switch should take place while an compress-send task or receive-decompress task is partially complete. More generally, an adaptation may be illegal while a particular communication operation is partially complete (e.g., operations that affect the size and contents of the message). An adaptation that affects two different processes may have to be performed before corresponding instances of tasks have completed execution (e.g., when two neighboring processes decide to

change the size of ghostzones used in a parallel simulation, both must change them at logically corresponding points in the computation [21]).

The static task graph provides a global view of a distributed or parallel application, and this view is available to *each process* in the program. This global view allows coordination requirements to be expressed using simple high-level rules (instead of explicit communication and synchronization operations). These rules can be specified by any process, and yet can refer to the tasks of all the different processes within the program. The key benefit will be that the *programmer need not write any explicit communication or synchronization* to coordinate a distributed adaptation operation.

We are currently expanding the framework above with such rules (and corresponding language syntax) that the programmer can use to specify the coordination requirements, and compiler and runtime support to implement them automatically and efficiently [22]. These rules are outside the scope of this paper, but briefly work as follows. We call a connected subgraph of the STG a *region*. A single region may include tasks from multiple processes. Logically, each region has a *progress counter* on each process that tracks the number of times that process has entered the region (via any incoming task graph edge). The progress counters essentially measure logical time. A coordination rule specifies the relationship that must hold between progress counters of different processes (for the same or different regions). For example, in the task graph of Figure 1, the *Client Connection* and *Receive Frame* tasks together form a single region. The coordination requirement would specify that the frame compression adaptations (i.e., inserting compress/decompress tasks or removing them) can only be performed when the local region counters for this region are equal on both processes, and that it should be performed at region entry. This ensures the two coordination requirements mentioned above are satisfied (the client and server should switch formats prior to the same frame, and should not switch while a compress-send task or receive-decompress task is partially complete). When the user-written adapt logic issues an adaptation on one process, the runtime systems on all affected processes communicate with each other and use the progress counters of the affected regions to safely schedule the adaptation for a future time on each process. The adaptation is then executed locally on each process when the region's progress counter advances to the scheduled time. The rules, compiler support, and runtime algorithm are described in [22].

3 Program Control Language

We propose a small language extension that we call Program Control Language (PCL), based on the adaptation framework discussed in the previous section. In addition to the mechanisms defined by the adaptation framework, PCL includes constructs for distributed performance monitoring and for triggering adaptations based on performance conditions.

Our current definition of PCL does not include two aspects of the framework: edge adaptations and coordination criteria. We have deferred these for different reasons. We have not found edge adaptations to be necessary in any of the examples we have worked with so far. The coordination criteria described as part of the framework is the focus of our current work. A tentative definition (syntax and semantics) for these criteria and preliminary experience with the compiler implementation are described elsewhere [22].

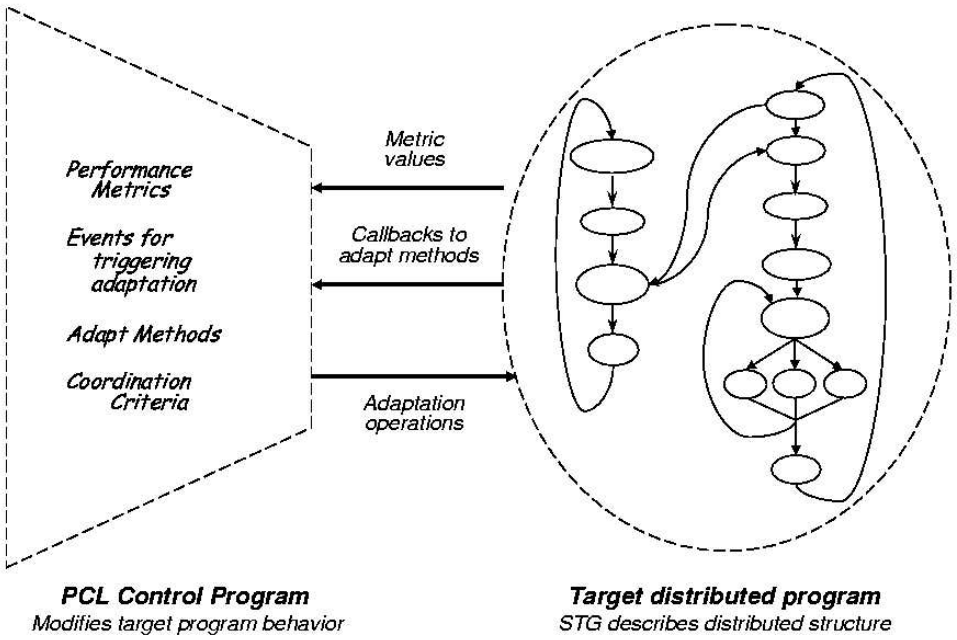

Fig. 4. Overview of the structure of an adaptive application in PCL

Figure 4 shows an abstract view of the logical structure for an adaptive code written with PCL. Fundamentally, a programmer using PCL for runtime adaptation would write logically separate "control code" that monitors and controls the behavior of a "target application" at runtime, using the static task graph as an abstract description of the target code.

The changes necessary within the target code are designed to be as non-intrusive as possible. Adding a legal adaptation strategy requires inserting PCL operations (syntactically, just function calls) into the target program at each of the following locations: (a) each program site where instrumentation must be inserted for application-level performance monitoring; (b) each adaptation site where any tasks must be inserted, removed or replaced; and (c) (optionally) at each point where the adaptation decision must be taken. All of the actually performance monitoring code, decision logic, and adaptation operations themselves are logically part of the control code and are kept separate from the target program.

3.1 Overview of PCL

Syntactically, PCL is a small extension to an existing language such as C, C++, Java, or Fortran. The semantics of PCL constructs are almost entirely language-independent, except where code fragments declaring new names or manipulating parameters of the target program have to be included in the PCL code. Our examples and experiments in this paper focus on using PCL with C programs.

The grammar shown in Figure 5 describes PCL. All the new constructs defined by PCL are structured as ordinary function calls to special functions, making the syntax familiar to programmers and simple to implement in an existing compiler. There are five primary features in PCL: *Task operations, ControlParameters, Metrics, Events* and *AdaptMethods*. These features are illustrated in the appendices which show the PCL code for a simple parallel fractal application and MPIPOV, a freely available MPI extension for a popular raytracing engine (the applications and the adaptations they use are described in more detail in Section 5). We will refer to the grammar and the example in the discussion below. The implementation of the operations is described in Section 4.

The actual adaptation logic is implemented by one or more `AdaptMethod`s. An adapt method can be called synchronously by manually inserting a call to the method at a particular point in the target application, or may be invoked asynchronously as event handlers in response to runtime events. The adapt method uses PCL operations on the static task graph to perform adaptation, as described below. The example in Appendix A declares the adapt method `Adapt()`, and it uses `ChangeParameter`, `RemoveTask` and `AddTask` operations to perform adaptation.

The `ControlParameters` construct exposes a set of variables in the base application which the adapt method may modify, as explained in Section 3.3 below.

Finally, several of the PCL constructs in Figure 5 (e.g., `AddTask`) take an optional argument `PID`. If this is absent, the PCL operation takes place on the calling process. If `PID` is present, it indicates the operation should take place on the specified remote process. The value of `PID` must be the process identifier assigned by the PCL runtime library to the remote process (called RTL identifier). RTL identifiers are further discussed in Section 4.1.

3.2 Identifying Tasks and the STG

Using PCL does *not* require the programmer to extract, describe, or even to know the Static Task Graph for a program. Since all adaptation operations in the framework (except edge adaptations which are not currently supported in PCL) operate on individual tasks, the programmer must only identify the relevant tasks within the target program, and sites where tasks may be inserted.

PCL allows any function to be considered a *task* and the name of the function also serves as the name of the task. In principle, any single-entry segment of code (e.g., a simple block scope in C identified by a label) containing no process creation or destruction could also be used as a task, but our current implementation does not support that. Special statement labels, called *adapt sites* are used to indicate points in the static task graph where adapt operations are allowed. Tasks may only be inserted and removed at an adapt site. The adapt site declaration also contains the actual arguments that will be passed to all tasks invoked at that site.

3.3 Task Graph Operations

Rules 8–10 in the grammar of Figure 5 show the syntax of the `AddTask`, `RemoveTask`, and `ReplaceTask` operations in PCL. The task to be added is specified by the function name. Recall that the necessary arguments are specified in the declaration of the *adapt site*. The compiler can easily check that the constructed call is valid at the specified adapt site. The RemoveTask operation only requires the adapt site and task name. An invalid RemoveTask operation (specifying a non-existent task at the site) triggers a runtime error.

An *adapt site* is a label identifying a location in the base application. Several `AddTask` operations may add tasks to the same site. PCL does not guarantee the order of execution of the tasks at a site, i.e., a site is represented as an unordered set of tasks. (If precise ordering semantics are required between the tasks, then multiple sites should be used.) When a task is added at a site, implicit control flow edges are created in the STG from and to the code before and after the site, corresponding to the insertion of the task into the program at the site (but without a guarantee of order between multiple tasks, as noted above). This simplifies the syntax of the `AddTask` operation in PCL, because the T_{pred} and T_{succ} parameters defined in Section 2 do not have be specified: they are implicitly defined by the choice of the adapt *site*.

The examples in the appendices show the use of `AddTask` and `RemoveTask` operations. For example, in Appendix **A**, the `AddTask` operation near the end of the Fractal example inserts a task representing a call to the function `CompressTask` at the statement labelled L12 within the file `Worker.c`. The parameters to the call are specified in the `pcl_AdaptSite` directive for this site near the top of the figure.

The `ControlParameters` list declares those variables of the target code that can be modified by the adaptor logic, i.e., can appear in the R_{val} expression of a `ChangeParameter` adaptation operation. The examples in Appendices A and B show a scalar control parameter (`MaxIterations`) and an array parameter (`TaskSizeForWorker[]`) respectively. The behavior of the code is undefined if the adapt method modifies any other variables of the target code, and the compiler can statically check this condition. Thus, this declaration helps expose the expected side effects of adaptation operations, and enables simple static checking of those side-effects.

(1) *control-parameter* ::= **pcl_ControlParameter**(*name*, *ptr*);

(2) *metric* ::= **pcl_Metric**(*name*, *site*, *function*, *param-list*);

(3) *read-metric* ::= **pcl_ReadMetric**(*name*[, *PID*]);

(4) *adaptop* ::= *addtask*
(5) | *removetask*
(6) | *replacetask*
(7) | *changeparameter*

(8) *addtask* ::= **pcl_AddTask**(*adapt-site*, *task*[, *PID*]);

(9) *removetask* ::= **pcl_RemoveTask**(*adapt-site*, *task*[, *PID*]);

(10) *replacetask* ::= **pcl_ReplaceTask**(*adapt-site*, *old-task*, *new-task*[, *PID*]);

(11) *adapt-site* ::= **pcl_AdaptSite**(*name*, *arg1*, ..., *argN*);

(12) *changeparameter* ::= **pcl_ChangeParameter**(*name*, *ptr*, *size*[, *PID*]);

(13) *adapt-method* ::= **pcl_AdaptMethod**(*function-name*);

(14) *async-call* ::= **pcl_AsyncCall**(*method*, *arg1*, ..., *argN*);

(15) *event* ::= **Event** *name* (*eventpolicy*, *function*);

(16) *eventpolicy* ::= **EP_Changed** (*name*)
(17) | **EP_Threshold** (*name*, *threshold-value*)
(18) | **EP_Percent** (*name*, *percent-value*)
(19) | **EP_Value** (*name*, *constant-value*)

Fig. 5. **PCL Grammar. New keywords are in bold font.**

3.4 Metrics

PCL provides the `Metric` keyword to declare performance metrics. A PCL metric is a named value that can be used by the adaptor logic to make adaptation decisions. It can measure or compute arbitrary aspects of program performance such as CPU utilization or frames per second of a video stream client. The metric declaration also guides the compiler in inserting performance instrumentation into the target program.

As shown in rule 2 of the PCL grammar, a metric consists of a name and one or more pairs of the following: a labeled site in the application and the instrumentation code to use at the site. The instrumentation code supplied by the user should be in the form of a function which performs the desired measurement at the specified site, such as incrementing a counter, computing a frame rate or taking a timestamp. The function must return the value of the computed metric (the type can otherwise be arbitrary). Multiple instrumentation functions can be given for the same labeled site.

The PCL runtime library provides several basic instrumentation primitives for metrics such as elapsed wall clock time over an interval, and CPU load average. These

library functions enable a variety of standard performance information to be collected and used within the adaptive code simply by declaring a metric for each one. They can also be used to construct user-defined measurement functions for more application-specific metrics.

The operation `pcl_ReadMetric` provides a simple mechanism to retrieve the value of a metric either locally or remotely from a specified process. The adaptation code can use metric values either directly within the logic of an adapt method, or use the metric (by name) in declaring events that can be used to trigger adaptation asynchronously as described in the next Section. The Fractal example in Appendix A illustrates the use of metrics. It declares a Metric named `elapsedTime`, using two standard functions `StartTimeStamp` and `EndTimeStamp` inserted at specified points in the code for a worker. Within the `Adapt` function, the call to `pcl_ReadMetric` then retrieves the value of metric elapsed time for a specified process *pid*.

It is important to note that there is no guarantee about the delay between the time that `pcl_ReadMetric()` is called and the time the value is actually sampled by the PCL runtime at process *pid*. In fact, as explained in Section 4, the entire `Adapt()` method is performed in a separate thread so that the remote operations to retrieve metrics and perform adaptations do not interfere with the execution of the target program.

3.5 Events

A PCL `Event` allows the user to invoke adaptation operations in response to particular events that occur at runtime. An event is triggered by some type of change in the value of a metric. The runtime system then dispatches the appropriate event handler, which is simply an adapt method that can choose whether or not to perform an adaptation in response to the event.

An event declaration specifies an event handler and an event policy, as shown by rules 16-19 of the grammar. The event handler must be a function that accepts one argument: an Event object. The event policy indicates the metric to monitor and the conditions under which the event is raised. When an event is raised the given handler function is invoked and passed an Event object with information about the raised event.

PCL supports four kinds of event policies:

`EP_Changed` The metric value changes by any amount

`EP_Threshold` The metric value crosses a constant threshold

`EP_Percent` The metric value changes by some percentage

`EP_Value` The metric value changes by a constant amount

The event handling procedure can use the Event parameter to optionally demultiplex multiple events, allowing the user to handle multiple events with the same adapt method.

An event may be triggered only when the value of a metric is recomputed. The `Metric` construct above exposes to the compiler the instrumentation code for the metric which is the only code allowed to assign a value to the metric. Thus, simply by specifying Metrics and Events, the compiler and runtime system can do instrumentation and monitoring of metrics and dispatch events automatically, freeing the programmer from a significant amount of implementation effort.

4 The PCL Compiler and Runtime System

The PCL compiler uses source-to-source transformations to compile the application and PCL code into code in the base language, which can then be compiled using standard system compilers. The PCL runtime is written in C++ and uses CORBA [27] for communication and Remote Method Invocation (RMI). These choices ensure that the PCL compiler and runtime are widely portable and would not be significantly more difficult to deploy and use than a runtime library that supports adaptation. The PCL compiler has been written using the LLVM compiler infrastructure [23] developed in our group at the University of Illinois. A key benefit derived from LLVM is that the PCL compiler is completely language-independent, and should be directly applicable to programs written in a different language if an LLVM front-end for the language is available (such a front-end for C++ is nearing completion).

The details of the runtime library are described in Section 4.1. Section 4.2 describes the important aspects of the PCL compiler. The key tasks the compiler and runtime system currently perform are code generation at adaptation sites to support adaptation, supporting distributed adaptation, and supporting distributed metrics and control parameters. These issues are discussed below. Events and correctness criteria are not yet fully supported by the system, and are briefly discussed in the section on future work.

4.1 The PCL Runtime System

PCL uses a sophisticated portable system to support distributed performance monitoring and adaptation. The runtime library uses a multithreaded implementation to hide the latency of remote metric accesses and adaptation operations. The library provides support for managing addition and removal of tasks at adapt sites, for providing access to both local and remote metrics, and for changing remote control parameters and performing other local and remote adaptation operations. Runtime support for local and remote operations are described in the next two subsections,

followed by specific runtime support for metrics.

4.1.1 Runtime Support for Local Adaptations

Recall from section 3 that an adapt site is a location where tasks are inserted and deleted. At any point in time, an adapt site represents an unordered set of zero or more tasks. This set is initially empty. The RTL maintains a task table for each site as an array of function pointers representing the set of tasks to be executed at that site. The RTL provides an interface for adding, removing, and executing the functions in that set, which are used by the compiler to implement `pcl_AddTask`, `pcl_RemoveTask` and `pcl_ReplaceTask` operations.

The RTL provides a registry mechanism that acts as a "reflection registry" to record the addresses of specific variable and function names in the base application address space, over which the RTL mechanism must exert control. The RTL uses this registry mechanism to implement both PCL metrics and `ChangeParameter` operations.

The evaluation of the PCL `pcl_Metric` directive results in creating an entry for the metric name in the metric registry of the local RTL instance. Whenever a metric query is received from a remote process, the address in the table is used to look up the metric value and return it to the originating site.

A similar mechanism is used for the `pcl_ChangeParameter` operation. For example, suppose a user wishes to designate some parameter p as a `ControlParameter`. The PCL compiler generates the code that registers the address of p with the RTL instance i (the instance associated with p's address space). Later, `ChangeParameter` directives may be issued to i and the modification of p is effected. The requisite actions on the part of the compiler are discussed below.

4.1.2 Runtime Support for Remote Adaptations

Each runtime library is continuously listening and ready to receive incoming PCL operations. Remote operations are handled by the runtime library by first forwarding the operation to the remote runtime library, where they are executed as if they were local operations.

The runtime system is complex because some adaptations may require distributed operations to be executed. One instance of the runtime library (RTL) is created for each process of the target code. Each RTL instance must have a unique identifier to name only that instance. The user code can call a routine to set the ID, for example, using MPI-assigned ranks in an MPI program. Alternatively the RTL can automatically assign IDs. In this case the runtime library provides routines for the user code to discover the ID of the runtime library of another process.

The user must pass a valid RTL ID to PCL operations that are to be performed on a remote process (this is the optional `PID` parameter to the PCL operations in

Figure 5). When a particular RTL instance receives a request from the base application to perform a remote operation on a specified process, it uses a routing table to redirect the request to the correct RTL instance. The remote RTL instance then executes the request as if it had originated locally, and the result of the operation (if any) is passed back to the execution unit that initiated the remote operation.

A program thread that invokes a PCL runtime operation is blocked until the operation completes. Blocking the user thread for the entire latency of a series of remote operations can be extremely expensive. Instead, PCL provides the `pcl_AsyncCall` operation which the programmer can use to specify that the adapt method should be invoked "asynchronously" in order to hide the latency of remote operations. Each RTL instance uses a second thread called a support thread. An asynchronous call to an `pcl_AdaptMethod` in the main application code is replaced by a call to a wrapper function that simply enqueues the function call and returns. The support thread dequeues and executes these adapt function invocations in FIFO order. Only the support thread is blocked during these operations, allowing the user thread to continue executing. The same mechanism can also be used to implement asynchronous invocations of the adapt method via the `pcl_Event` mechanism, but this is not currently supported.

The PCL runtime system also uses a sophisticated scheduling strategy for coordinating adaptation operations on different processes, both relative to each other and relative to the execution state of the user threads. These operations implement the abstract coordination criteria specified by the programmer. As with the language rules, preliminary experience with the runtime implementation is described in [22].

4.1.3 Runtime Support for Metrics

Each metric must be registered with the runtime library at execution time. This is done by inserting a call to a metric registration function (passing it a pointer to the metric) at the program entry point. Currently, we only support metrics of type `double`. The runtime library uses the pointer to the metric to service remote requests for the value of the metric as described in section 4.1.

As a convenience to the PCL user, we provide a library of robust, commonly-used metrics. Most of the library metrics can be either local or remote, and are qualified by an optional parameter (a RTL identifier) which specifies that the metric needs to be queried on a remote host, although some library metrics are implicitly remote, and thus require the RTL identifier parameter. Some examples of frequently-used metrics are elapsed time, CPU load averages, network bandwidth and latency to/from a remote host, and "rate metrics" (i.e., simple counter metrics that are implicitly divided by a sampled elapsed time value).

A programmer can also write additional system or application-level metrics, and use them easily within a PCL program. The `pcl_Metric` language mechanism makes no distinction between user-defined metrics and the set of metrics provided in the RTL. New metrics can also use existing ones simply by invoking the metric implementation

functions directly (e.g., this can be used to compute a "metric" that combines an application parameter and an ordinary performance attribute)

The primary concrete benefit of the current `pcl_Metrics` mechanism (compared with direct use of a runtime library) is to simplify retrieving remote metric values, and to minimize the overheads of doing so by using asynchronous calls to adapt methods. The mechanisms for retrieving remote metric values can then be used directly for these user-defined metrics.

For the longer term, we are exploring how the notion of performance properties of a program can be formalized at the language level so that a program can refer to and use it's own performance attributes directly. This would enable compiler support for analyzing complex low-level metrics (e.g., pipeline performance) or for automatically predicting common program metrics (e.g., cache misses). The potential benefit is a semi-automatic performance estimation strategy that combines application-knowledge and compiler knowledge for more effective performance measurement and semi-automatic performance prediction. We tentatively refer to this capability as "performance programming."

4.2 Compiler Support

The PCL compiler's role is a straightforward translation of PCL language constructs to runtime library mechanisms. It is simple enough to implement as a source-to-source compiler or directly within the front end of an existing native compiler. Our implementation within the LLVM framework operates as a source-to-C compiler, and is described here.

4.2.1 Code Generation at Adaptation Sites

The PCL compiler first builds a list of adapt sites by analyzing the `AddTask`, `RemoveTask`, and `ReplaceTask` operations which appear in the program. These tasks will be recorded in the set of function pointers maintained by the runtime library for each adapt site. For efficiency each adapt site is assigned a unique identifier which the runtime library and compiler use to identify the site.

Each adapt site is then transformed as follows. The compiler inserts a call to the runtime library at the adapt site to obtain a reference to the task set. Finally, the compiler generates code to iterate over the table of function pointers for that site and invoke each task currently in the set. Each task is passed the same list of expressions as actual parameters, as specified in the adapt site declaration. This strategy ensures that it is possible to check that the actual parameters are legal expressions at the adapt site and match the required arguments to the functions for each task at the site. If tasks with different parameters need to be invoked at the same point (or if a particular order has to be enforced), the user can create multiple adapt sites at that point using different labels.

The compiler replaces each `pcl_AddTask` and `pcl_RemoveTask` operation (which are used within a `pcl_AdaptMethod` in the PCL control code) with an appropriate call to a runtime mechanism to add or remove a task from the given site. The compiler treats a `ReplaceTask` operation as a `RemoveTask` operation immediately followed by an `AddTask` operation. A remote `ReplaceTask` operation is also currently performed as two remote operations, but will be optimized to use a single remote transaction in the future.

A completely different runtime strategy for PCL is to maintain an actual task graph representation at runtime, and use that to drive the execution of the program. This would essentially be a form of control reflection, as opposed to the reflection of data or types provided in languages like Java. [1] (A close analogy for a sequential program would be to preserve the control flow graph and execute the code by emulating the graph.) In fact, some graphical programming languages like CODE and HENCE do maintain a runtime graph that drives program execution [28]. (Those languages were aimed at parallel programming, did not support any runtime changes to the graph.) While this is a powerful approach for implementing adaptation, the runtime overhead of this approach could be quite high, and would probably have to be supplemented with runtime code generation to reduce the overhead to an acceptable level. So far, we have found the simple mechanisms for adding and removing tasks surprisingly powerful, and has allowed us to rely exclusively on static code generation for implementing the PCL language.

4.2.2 Metrics

A PCL metric declaration indicates one or more program locations and exactly one function to invoke for each location. The compiler automatically inserts the metric function invocations at the given locations. The compiler could inline suitably lightweight metric functions for greater efficiency. Alternatively, the compiler could treat each metric function as a task and each invocation point as an adapt site. If the latter approach was taken, the control logic could dynamically switch the measuring code on and off as needed, via the standard `AddTask` and `RemoveTask` operations. This flexibility could be particularly useful for programs that adapt only periodically or do so in a very coarse-grained manner.

4.2.3 Control Parameters

It is quite common for an application-level adaptation to be governed by modifying a value of an application variable at runtime. Declaring such a parameter as a PCL control parameter exposes this fact to the compiler and indirectly to the runtime system allowing a control parameter to be changed remotely (or locally) by another process. It also ensures that all adaptation operations are implemented uniformly via PCL mechanisms rather than having some be implicit within the application or control code.

[1] This distinction was pointed out by Jon Howell at Microsoft Research.

For each control parameter the compiler generates a callback function that will assign a new value to the parameter. The new value is passed to the function as an actual argument. The callback function is automatically registered with the runtime library by adding a call to the registration function at the program entry point. Finally, each `pcl_ChangeParameter` operation is simply replaced by a call to this callback function.

4.2.4 Adapt Methods

The compiler support for invoking adapt methods (any function declared `pcl_AdaptMethod` keyword) is straightforward. First, the programmer can call this function directly just like an ordinary function to initiate an adaptation decision (which may or may not actually perform any adaptation) — no special compiler support is required. If the programmer invokes the function using the `pcl_AsyncCall` mechanism, the compiler generates a wrapper function for the method so that the runtime library's support thread can invoke it and pass only a single argument.

To define the wrapper function, the compiler generates a structure type that can hold all the function arguments. The structure is packed before calling the adapt method wrapper function, then unpacked and used to invoke the adapt method. A pointer to the structure is passed to the wrapper function as the only argument. The structure itself is simply heap allocated, and the compiler adds a call to free the structure at the exit point of the adapt method. (Alternatively, a small buffer of preallocated structures can be created for more efficient execution but this has not proved necessary so far.)

5 Experimental Results

In order to evaluate the PCL language design and implementation, we have tried to address the following key questions experimentally:

(1) What are the benefits of using PCL for implementing an adaptation in a distributed program? We only discuss this question subjectively and based on our own experience to date.

(2) Do the runtime operations implementating high-level PCL mechanisms have significant overhead, i.e., is there a performance penalty to using PCL for adaptation?

(3) Does the asynchronous execution of adapt methods (particularly those involving remote operations) provide a significant performance benefit over direct synchronous execution?

We emphasize again, as noted in the Introduction, that our goal is not to evaluate the efficacy of particular adaptation strategies for the applications we studied but rather

to examine the benefits and overheads of PCL for implementing these adaptations.

5.1 Applications

We used three applications for our experiments: Fractal, POVray, and a PDE solver. These applications and our experience with using PCL for them are described below. We were unable to use ATR because it is written in C++ and the PCL compiler is currently limited to C.

5.1.1 Fractal

The fractal program is a parallel master-worker application written with MPI that computes mathematical fractals such as the well known Mandelbrot set. The master assigns a fixed slice of the image to each available worker which will compute that part of the fractal and return the data to the master. The master displays the partial results and assigns another portion of the image to the worker. The worker performs an iterative computation to determine whether each point in the complex plane is a member of the set. A parameter called `MaxIterations` sets an upper limit on this computational work, with the tradeoff that a a lower value decreases the amount of work but can produce a higher number of false positives. Each worker has a copy of this parameter.

We used PCL to add two adaptations to the fractal program. Appendix A shows the entire PCL code for this example. Both adaptations are performed by the master. The first changes the value of `MaxIterations` for individual workers in order to perform load balancing when CPU loads at the workers vary significantly. We declared this variable as a control parameter and the `pcl_ChangeParameter` operation allows the master to adapt this parameter for each worker *without writing any explicit communication.* The adaptive code uses a PCL metric to measure the average elapsed time for each individual worker (also without any communication), and averages these across workers. If the average time for a single worker is significantly above or below the average across workers then `MaxIterations` is decreased or increased respectively for that worker.

The second adaptation turns compression of data on and off at each before sending results to the master, in order to adjust to potentially variable network bandwidths. Compression can be reduce message latencies, but at a cost in CPU time on both worker and master for compression and decompression (the CPU cost of the former is much higher). The adaptation is implemented by adding or removing a compression task on an individual worker; again, this is controlled by the master *without writing any explicit communication.* The master first reads the CPU load average for a worker and adds or removes the compression task on the worker based on whether this load is high or low. A more sophisticated strategy is for the master to measure the network bandwidth to determine if compression would actually be beneficial. The master simply uses the size of incoming messages to determine whether the

data is compressed.

The master periodically executes the adapt method ("`Adapt()''` after a fixed number of image slices have been computed. It invokes this function asynchronously from within its main loop via a `pcl_AsyncCall` (not shown). This function gathers performance information from all of the workers, makes the decisions, and if necessary initiates the adaptations described above.

PCL provides three tangible benefits for this code. First, it allows the adaptation operations to be specified in high level terms using the `AddTask` and `RemoveTask` directives, instead of embedding these choices into the application using conditional branches. Second, it allows the application to be instrumented for measuring worker task times simply by declaring the `elapsedTime` Metric, without having to insert instrumentation code within the target application. Third, it allows both the remote adaptation operations to be specified and the remote metric values to be accessed easily within the master, with no explicit communication. (A distributed performance library could retrieve remote metric values for a predefined set of metrics, but it would be difficult to do for arbitrary user-defined metrics without language support.) Finally, PCL also provides some intangible benefits, by separating the adaptation code cleanly from the base application, exposing the interfaces between them (via the `ControlParameters` and adaptation operations), and exposing the metrics used to guide adaptation.

5.1.2 POV-Ray

POV-Ray is a widely used, publicly available raytracing application [25]. A parallel MPI version (MPIPOV) of POV-Ray exists [24], and it uses a straightforward master-worker application model to assign some number of fixed-size blocks of image regions to workers for rendering. POV-Ray performs supersampling to eliminate aliasing artifacts, and this induces a partial order on the rendering of blocks where each block depends on its left and upper neighbors. The master enforces this partial order when giving out new blocks to workers. In this version, each worker was assigned a fixed, constant number of blocks at a time.

In order to adapt to varying CPU loads at the workers, we created an adaptive version of MPIPOV that varies the number of blocks assigned to each worker using a a simple load-balancing strategy. Each worker computes a sliding-window average of its task completion times for N successive tasks. (This is important in order to smooth out variations in the computational cost of each task, which depends on the features of its part of the image.) The master periodically queries each worker for the current value of this sliding-window average, and averages these values across all workers. The master then compares the average from each worker to the global average and uses that ratio to scale the number of blocks assigned to each worker. Thus, slow workers are penalized by being assigned less work, while workers who complete their assigned tasks rapidly are given heavier workloads.

The PCL code we used for adaptation in MPIPOV is shown in Appendix B. Two

PCL mechanisms are employed to realize the above adaptation strategy: a remote metric query that retrieves the average completion time for a given worker, and a local `ChangeParameter` operation that modifies the number of image blocks assigned to a given worker.

PCL provides two tangible benefits for this code, similar to the corresponding benefits for Fractal: simple declaration of a metric to measure the moving averages of task times, and simple, direct access to remote metric values. (No remote adaptation operations are needed for this code.) It also provides the same intangible benefits as for Fractal.

5.1.3 PDE Solver

Many scientific computing and simulation problems require solving large partial differential equations. We have implemented a simple PDE solver with adaptive ghostzones, similar to [21]. We used a one dimensional decomposition to simplify the parallelization of the problem, but this does not reduce the complexities of distributed coordination.

Ghostzones are used in domain-decomposition-based PDE solvers to hide network latencies by replacing many smaller messages by fewer but larger messages. If there are G ghostzones on the boundary then each pair of adjacent processes will have to exchange G rows of data every G iterations. This has lower communication cost than exchanging one row of data every iteration, but requires $G-1$ rows of computations to be duplicated on bot neighbors. The ghostzones adaptation changes the value of G at each interprocess boundary to adapt to changing bandwidth conditions. Changing G requires distributed coordination among the two neighboring processes because each process must know the number of ghostzones over its borders in order to know when to exchange ghostzone rows.

We initially wrote a non-adaptive parallel PDE solver with MPI which used a statically fixed number of ghostzones for all boundaries. The ghostsize adaptation required the introduction of two new tasks (using `pcl_AddTask`) that reallocate and update the PDE solver's internal data structures when the number of ghostzones is changed. Because each of these tasks should only be executed once when the ghostsize changes, each task removes itself from the adapt site by executing a purely local non-composite `pcl_RemoveTask`, which removes it from the adapt site so that it is not invoked the next time the region is entered.

The adaptation logic in the PDE solver uses only local metrics to determine when to increase or decrease the number of ghostzones. The PDE solver measures the time to transfer ghostzone rows between each neighbor and the total time to compute each iteration. If the transfer time is too large or small compared to the actual work time then ghostsize is changed.

5.2 Overheads of Using PCL

In order to evaluate the performance cost of using PCL as a basis for adaptive applications, we run the adaptive PCL versions of the applications so that they perform all runtime operations for adaptation (the basic PCL runtime threads, performance monitoring, and retrieving local and remote performance metrics) but do not actually perform any adaptations.

More specifically, we compare the performance of three different versions of each application. The first version is the original application itself, using no PCL and performing no adaptations. In the second version, the adapt method does not execute any remote metric requests or adaptation operations (i.e., the only overhead is creating and running the PCL runtime threads). In the third version, the `Adapt()` method executes the loop that requests the elapsed time metric from each worker but does not do any actual adaptation operations. The total number of processors was varied between 2 and 16 for these experiments.

Figures 6 and 7 show the execution times for the three versions of Fractal and MPIPOV respectively. The figures show that there is essentially negligible additional overhead both for version 2 (PCL with no remote monitoring), and even for version 3 (PCL with remote monitoring). (The following subsection explains why the remote monitoring is so low.) These results indicate that applications can use PCL without penalizing performance, even if there is no benefit to adaptation at all. (Figure 7 has only two plots because the adaptation code performs no remote operations other than remote metric value requests.)

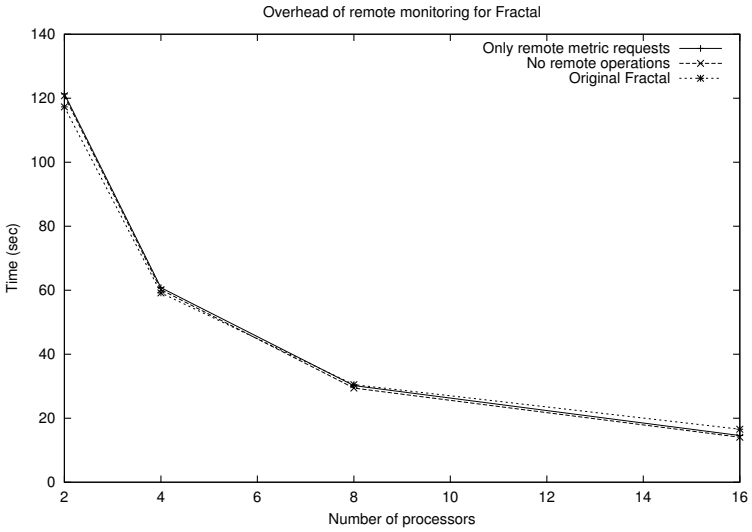

Fig. 6. Overhead of remote monitoring for Fractal

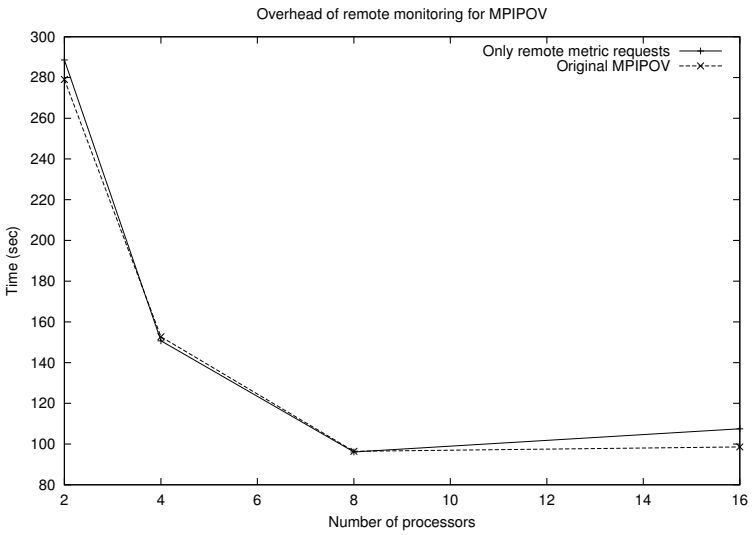

Fig. 7. Overhead of remote monitoring for MPIPOV

5.3 Effect of Synchronous versus Asynchronous Adaptation

The next goal of our experiments is to evaluate the importance and effectiveness of our strategy of executing adapt methods asynchronously instead of synchronously, using a separate thread that runs concurrently with the original application thread.

In the case of Fractal, the different in performance between the synchronous and asynchronous strategies can be strongly affected by the presence or absence of remote adaptation operations. We therefore consider a best-case and a worst-case scenario from the viewpoint of the synchronous version. The worst-case version of the adapt method for Fractal gathers performance metrics from each worker then performs one change parameter, immediately followed by one remove task, immediately followed by one add task (this is an upper bound on the amount of work done by any invocation of the adapt method). The best case version only gets remote metric values and does no other remote operations. In contrast, MPIPOV uses purely local adaptation operations having negligible runtime overheads, so this distinction is not important. In the PDE solver, the original code with no adaptations is itself equivalent to the "best-case" version since the adaptive version does no remote metric operations.

Figure 8 shows the execution time for synchronous and asynchronous invocations of the best-case and worst-case adapt methods for Fractal. The synchronous version does not scale well as it performs worse for 16 processors than 8. The long delays while the adapt method executes create a bottleneck in the master and thus severely reduce concurrency. [2] The asynchronous version does not suffer from this problem

[2] It is important to note that these bottlenecks are a characteristic of the adaptation algorithm and its performance monitoring requirements, and not of the PCL implementation.

and shows scalability comparable to the original application.

The best case adapt method does significantly fewer remote operations than the worst-case. However, the synchronous version still exhibits poor scalability, and the execution time for 16 processors is no better than for 8. As in the worst-case, the asynchronous version is very close in performance to the original application (and also to the asynchronous worst-case version).

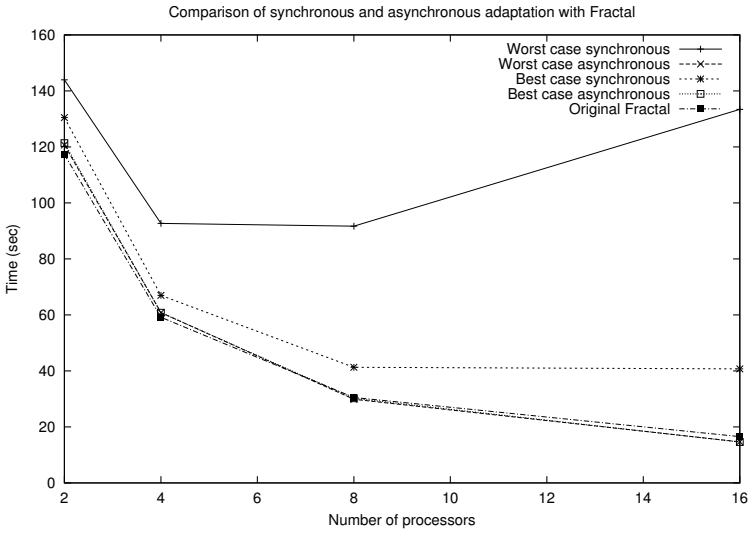

Fig. 8. Comparison of synchronous and asynchronous adaptation for the Fractal application

Figure 9 shows the execution time for the three versions of the MPIPOV program. As in Fractal, synchronous adaptation severely limits the concurrency of the workers which directly limits scalability. Asynchronous adaptation achieves performance very close to that of the original application.

Figure 10 shows the execution time for the non-adaptive PDE solver, and asynchronous and synchronous versions of the adaptive solver. The adaptation logic does not invoke any operations to read remote metric operations because only local metrics are used. Remote operations are used to execute the adaptation. All three versions of the application perform roughly the same, with the asynchronous version performing slightly better than the synchronous version.

6 Related Work

Although there has been a wide range of work on middleware, libraries, and runtime systems to support adaptive distributed applications, there is very little previous work we know of that specifically aims to provide programming language and compiler support for such applications. For this reason, we believe the PCL project is

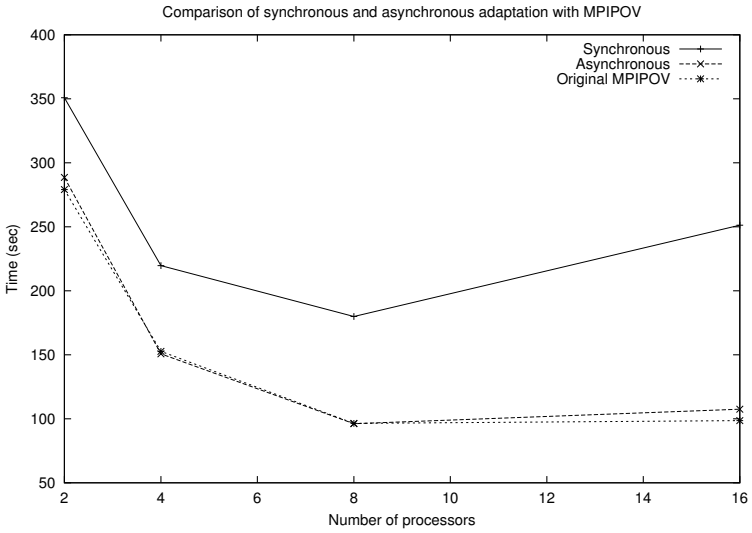

Fig. 9. Comparison of synchronous and asynchronous adaptation for MPIPOV

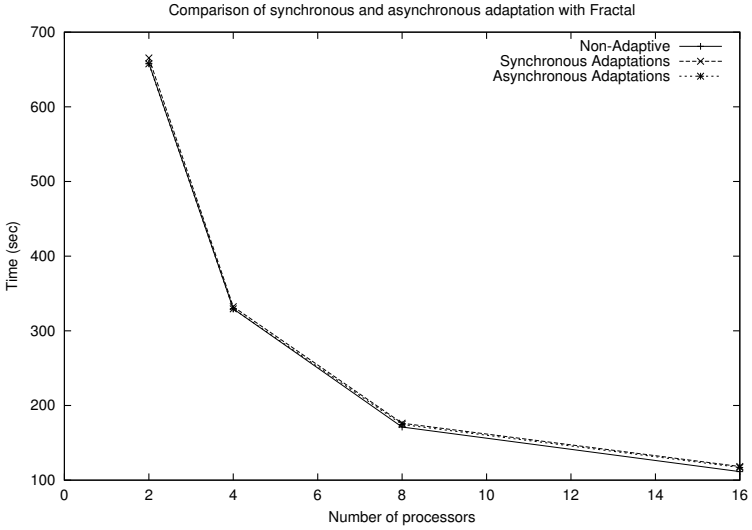

Fig. 10. Comparison of synchronous and asynchronous adaptation for the PDE solver

a relatively new strategy for supporting such applications compared with the non-language-based approaches taken before. Nevertheless, we do expect that that the most practical long-term approach for building sophisticated adaptive applications is to base them on reusable domain-specific libraries or middleware frameworks, and to use PCL for implementing the domain-specific adaptation policies and services within those reusable layers. We note examples of such uses below.

The only other language or compiler effort we know of that is specifically aimed at supporting adaptive applications (such as those targeted in our work) is the Grid Application Development Software project (GrADS) [14], which is developing

a broad software platform for Grid applications. They have proposed a compiler-based strategy for encapsulating an application as a configurable object program that can be reconfigured and reoptimized at runtime. (They have also developed a wide range of runtime systems and techniques, described later below.) They have not proposed specific language mechanisms or higher-level programming models to use in their applications. We believe that the PCL language and adaptation framework could be a very effective programming model for developing applications on top of such an environment. For example, it would expose extensive information about adaptation behavior to the compiler (the configurable object generator), to dynamic compilers, and to the runtime system (e.g., it might enable the automatic generation of performance contracts [29]).

An alternative strategy for simplifying adaptive applications is to provide libraries tailored for specific classes of applications. The AppLES project has developed template libraries (AMWAT [30] and APST [31]) for master-worker and parameter sweep codes. The Condor project has developed a class library called MW for master-worker Grid codes [32], which was used to implement the original ATR application. Such libraries can potentially implement adaptation techniques tailored to these specific application classes transparently to applications that use them (e.g., both AMWAT and MW provide built-in support for tolerating failed worker processes). This approach is simpler to deploy and use than a language like PCL but it is inherently limited to specific classes of applications, and (if done transparently) it precludes the use of application information in adaptation decisions.

A more powerful but more narrowly focused strategy is to provide domain-specific frameworks for building a class of applications and incorporating grid and adaptation capabilities within that framework so that it becomes automatically available to all application using it. Some examples of such a strategy include a version of ScaLAPACK modified to run in a Grid environment [20] and CACTUS-G [21], a Grid-enabled version of the popular CACTUS simulation framework. For example, CACTUS-G includes sophisticated adaptation strategies that are completely transparent to applications built on top of CACTUS. We believe that PCL would provide a valuable basis for implementing the adaptation strategies *within* such a framework because they essentially require the kind of performance monitoring, adaptation and coordination mechanisms that PCL provides. In fact, the adaptation in the PDE solver example used in our experiments of Section 5 was modeled directly after the ghostzones adaptation in CACTUS-G. In the long term we believe that such frameworks may be the most practical way of making Grid programming available to large classes of high performance applications, and PCL could serve as an implementation layer for such frameworks.

For distributed applications with quantifiable Quality-of-Service requirements (e.g., many commercial and multimedia applications), an attractive strategy is to provide domain-specific middleware on which such applications can be constructed. Some examples are the Odyssey system from CMU for mobile applications, the Agilos system and its successors from Illinois for distributed multimedia applications, and the Quality of Objects (QuO) project at BBN Systems for reliable and adaptive distributed systems. Odyssey supports application adaptation for mobile comput-

ing [2,33]. It monitors the resource usage (e.g., bandwidth or power usage) of a set of concurrent applications and notifies an application that it needs to adapt when availability of a resource violates application-specified bounds. Agilos supports QoS adaptations in distributed multimedia applications [16]. The framework manages system-wide resource allocation (e.g., network bandwidth and CPU) for a collection of applications, and provides a generic fuzzy logic engine that applications can use to make internal adaptation decisions as resource allocations change. The QuO (Quality of Objects) project at BBN Systems has developed middleware platforms for reliable and adaptive distributed systems. They discuss distributed coordination and synchronization techniques for resource management and sharing in the presence of QoS constraints [34,35] and for coordinating and scheduling real time systems [36]. The QuO system and successors to Agilos (QCompiler and QosTalk) also provide configuration languages or graphical user interfaces to simplify the specification of Quality-of-Service metrics for an application, which are then automatically translated to configuration parameters for the underlying system [37–39]. Depending on the structure of the middleware framework and the adaptation policies, it may be possible to use PCL to implement the adaptation strategies internal to these frameworks themselves. For example, we are exploring using PCL as the "implementation layer" for the QosTalk configuration language [39].

There are a number of language designs and software engineering approaches (not specific to adaptation) that have features related to those in PCL. The PCL strategy of specifying adaptation behavior separate from the target distributed application has been borrowed directly from *aspect-oriented programming* (AOP) languages [40] (but without the formal syntax of aspects). Aspect-oriented language extensions have been previously proposed for specifying synchronization and/or communication behavior of a class separately from the other methods of that class [41,42]. These language extensions served as models for this feature of PCL. Some Architecture Description Languages (ADLs) [43] such as C2 [44], Darwin [45] and Weaves [46], allow the software architecture of component-based application to be dynamically changed during program execution, by inserting, removing, or changing the interconnections between elements of the application. Those efforts can provide formal and sophisticated support for specifying dynamic adaptation [47]. In contrast, PCL is less formal but it is not restricted to component-based applications. Also, PCL is specifically tailored to support dynamic adaptation based on runtime performance monitoring. Finally, research on program specialization provides techniques to specialize code to specific data values or patterns at runtime [48–51]. Our primary goal so far has been to provide higher-level mechanisms for performance-based adaptation. The program specialization techniques could be a powerful way to optimize program performance at runtime after an adaptive change, and they are based on orthogonal mechanisms that could be added to PCL.

Some parallel programming languages have been based on graphical representations of parallelism (e.g., [28,52]). In those languages, the task graph specification is an intrinsic component of the executable specification of the program. Those languages do not provide any mechanisms for (conceptually or actually) modifying the task graph during program execution, i.e., neither the task graph nor those languages provide any specific support for adaptation. In contrast, the task graph in our work

primarily serves as a basis for reasoning about and specifying adaptation strategies, and is not required for the distributed computation itself.

Finally, there has been extensive work on runtime libraries and network layer support for adaptive distributed applications. Our work is complementary to these in the sense that adaptive distributed applications (written in PCL or otherwise) will require extensive support from such runtime systems. We cite a limited subset of such systems here.

Some runtime systems aim to provide a comprehensive set of runtime services for specific classes of Computational Grid applications. Globus [9] and Legion [11] each provide a wide range of runtime services including resource discovery, scheduling, authentication, and job migration. Condor [53] provides facilities for job scheduling and migration for workstation networks and for Grid computing. The EveryWare toolkit [54] provides a set of application services for building Grid applications, including portable interfaces to a wide range of communication infrastructures, performance forecasting services for resource and application performance predictions, and distributed state management to enhance fault-tolerance and scalability. (The application is still required to implement and manage adaptation within the application code, using these services.) The Service Grid [12] architecture provides dynamic adaptation (specifically, replication and deletion) of Grid-based services in response to changing client usage patterns, in order to enable reliable and scalable Grid services. GrADSolve is a Grid-based RPC system that simplifies the remote invocation of parallel code and supports dynamic allocation of Grid resources based on a high-level description of application characteristics) [55].

Other runtime systems provide more specific services, particularly scheduling, performance monitoring, and checkpointing services. The AppLES effort has developed techniques and tools for application level scheduling in wide-area distributed systems [56,57]. (The AMWAT and APST template libraries cited above simplify the use of such scheduling for specific classes of applications, as discussed above.) The SRS framework provides migration and checkpointing services for MPI applications [58]. At the network level, systems such as Conductor [7], TIMELY [8], the Network Weather Service (NWS) [5], and a network API from Rutgers [6] provide various mechanisms for network level performance monitoring, resource reservation, and network-level adaptation. MicroGrid [59] and SimGrid [60] are simulators for performance analysis of Grid applications, which can be extremely valuable because of the difficulty of evaluating performance in a dynamic wide-area setting.

7 Summary and Future Work

This paper described the design and implementation of Program Control Language (PCL). PCL provides an abstract, global framework and corresponding language mechanisms for reasoning about and describing runtime adaptations in distributed applications. The language provides automatic mechanisms for inserting, remov-

ing and replacing tasks within local or remote processes of an executing program, for monitoring performance of those processes, for asynchronous execution of remote performance queries and adaptation operations and for coordinating operations on different processes. Together, these mechanisms are designed to enable runtime adaptations to be for a given application to be specified in simple high-level terms within separate "control code," and to have the operations be implemented efficiently by a compiler and runtime system.

Through experimental measurements, we have shown that there is virtually no performance penalty when using the abstractions in PCL for a fairly wide range of runtime adaptations. We have also shown that an intelligent asynchronous implementation of adaptation operations is crucial to achieving these low overheads.

There are several directions we intend to pursue in the future. One near-term goal is to complete and evaluate the implementation of the automatic coordination mechanisms discussed briefly in Section 2.3. We also aim to expand the set of performance metrics provided by the runtime library.

Longer-term, there are several potentially interesting directions enabled by this work. First, we are exploring the idea of "performance programming" mentioned in Section 4.1.3, which enables the performance properties of a program to be formally expressed and used (and potentially manipulated via adaptation) within the program itself. Second, we aim to work with application groups to convert one or two large-scale distributed applications into PCL-based adaptive codes suitable for wide-area distributed computing. Finally, one of the most interesting possibilities is that the language mechanisms in PCL expose important aspects of adaptation behavior to the compiler and runtime system that would be impossible to infer automatically. In principle, the compiler and runtime may be able to use these to perform optimizations (especially runtime optimizations) on the program after an adaptive change, e.g., specializing parts of the code to the new behavior exposed by that change.

Appendix A: PCL Code for Fractal

```
/* Declare an adapt site in the Worker code */
pcl_AdaptSite("Worker.c:L12", header, data, pSize);
...

/* Adaptor Code */
pcl_ControlParameter("MaxIterations", &MaxIterations);
Metric elapsedTime("Worker.c:L5", StartTimeStamp(),
    "Worker.c:L6", EndTimeStamp());
void Adapt(int numWorkers) {
    pcl_AdaptMethod("Adapt");
    /* Compute average elapsed time of all workers */
    double avgTime = 0;
    for (pid = 1; pid <= numWorkers; ++pid) {
        adaptInfo[pid].time = pcl_ReadMetric("elapsedTime", pid);
        avgTime += adaptInfo[pid].time;
    }
    avgTime = avgTime / numWorkers;
    /* Adapt each worker as necessary */
    for (pid = 1; pid <= numWorkers; ++pid) {

    /* Adapt #1: Increase or decrease MaxIterations for a worker */
    if (adaptInfo[pid].time > 1.2 * avgTime &&
        adaptInfo[pid].iter == high_MaxIterations) {
        pcl_ChangeParameter("MaxIterations",
            &low_MaxIterations, sizeof(int), pid);
        adaptInfo[pid].iter = low_MaxIterations;
    } else if (1.2 * adaptInfo[pid].time < avgTime &&
        adaptInfo[pid].iter == high_MaxIterations) {
        pcl_ChangeParameter("MaxIterations",
        &high_MaxIterations, sizeof(int), pid);
        adaptInfo[pid].iter = high_MaxIterations;
    }

    /* Adapt #2: Add or remove compression for a worker */
    loadAvg = pcl_ReadMetric("CPULoadAverage", pid);
    if (loadAvg[LOAD_AVG_1MIN] > 1.15
        && adaptInfo[pid].cmpr == 1) {
        pcl_RemoveTask("Worker.c:L12", CompressTask, pid);
        adaptInfo[pid].cmpr = 0;
    } else if (loadAvg[LOAD_AVG_1MIN] < 1.07
        && adaptInfo[pid].cmpr == 0) {
            pcl_AddTask("Worker.c:L12", CompressTask, pid);
            adaptInfo[pid].cmpr = 1;
        }
    }
}
```

Appendix B: PCL Code for MPIPOV

```
/* Adaptor Code */
pcl_ControlParameter("TaskSizeForWorker[]");
Metric elapsedWorkerTime(
    "MPIren.c:ELAPSED_START", StartTimeStampWithMovingAvg(),
    "MPIren.c:ELAPSED_END", EndTimeStampWithMovingAvg());

void Adapt(int workerID) {
    pcl_AdaptMethod("Adapt");

    /* Obtain the elapsed time for this worker, calculated
    over a sliding window */
    workerInfo[workerID].elapsed =
        pcl_ReadMetric("elapsedWorkerTime", workerID);

    if(HaveAllWorkerTimes()) {
        /* Ensure we have values for all workers. CalcGblWorkerAvg()
        computes the global average across all workers */
        double awardRatio = CalcGblWorkerAvg() /
        workerInfo[workerID].elapsed;

        /* Penalize slow workers, reward fast ones. The result of
        this adaptation will become apparent in the base
        application on its next use of TaskSizeForWorker[workerID]. */

        double tmp = awardRatio * BASELINE_TASKS_FOR_WORKER;
        pcl_ChangeParameter("TaskSizeForWorker[workerID]",
            &tmp, sizeof(double));
    }
}
```

References

[1] I. Foster, C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufman, Inc., 1999.

[2] B. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, K. Walker, Agile Application-Aware Adaptation for Mobility, in: Proc. 16th ACM Symposium on Operating System Principles, 1997.

[3] A. Vahdat, T. Anderson, M. Dahlin, D. Culler, E. Belani, P. Eastham, C. Yoshikawa, Webos: Operating system services for wide area applications,

in: Seventh Symposium on High Performance Distributed Computing, 1998.

[4] M. van Steen, P. Homburg, A. Tanenbaum, Globe: A wide-area distributed system, IEEE Concurrency (1999) 70–78.

[5] R. Wolski, N. Spring, , J. Hayes, The network weather service: A distributed resource performance forecasting service for metacomputing, Journal of Future Generation Computing Systems 15 (5-6) (1999) 757–768.

[6] P. Sudame, B. Badrinath, On providing support for protocol adaptation in mobile wireless networks, Tech. Rep. DCS-TR-333, Department of Computer Science, Rutgers University (1997).

[7] M. Yarvis, P. Reiher, G. Popek, Conductor: A Framework for Distributed Adaptation, in: Proc. 7th Workshop on Hot Topics in Operating Systems, 1999.

[8] V. Bharghavan, K.-W. Lee, S. Lu, S. Ha, J. R. Li, D. Dwyer, The TIMELY Adaptive Resource Management Architecture, IEEE Personal Communications Magazine 5 (4).

[9] C. K. I. Foster, Globus: A metacomputing infrastructure toolkit, Intl Journal of Supercomputer Applications 11 (2) (1997) 115–128.

[10] N. K. I. Foster, A grid-enabled mpi: Message passing in heterogeneous distributed computing systems, in: Proceedings of 1998 SC Conference, 1998.

[11] A. Grimshaw, W. A. Wulf, the Legion Team, The legion vision of a worldwide virtual computer, Communications of the ACM .

[12] J. B. Weissman, B.-D. Lee, The service grid: Supporting scalable heterogenous services in wide-area networks, in: SAINT 2001, 2001.

[13] L. Kale, S. Krishnan, Charm++ : A Portable Concurrent Object Oriented System Based On C++, in: Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA), 1993.

[14] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, S. L. Johnsson, K. Kennedy, C. Kesselman, D. A. Reed, L. Torczon, R. Wolski, The grads project: Software support for high-level grid application development, Tech. rep., Rice University (Feb. 2000).

[15] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, R. E. Schantz, Aqua: An adaptive architecture that provides dependable distributed objects, in: Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98), 1998, pp. 245–253.

[16] B. Li, K. Nahrstedt, A Control-based Middleware Framework for Quality of Service Adaptations, IEEE Journal of Selected Areas in Communications, Special Issue on Service Enabling Platforms (to appear).

[17] V. Adve, R. Bagrodia, E. Deelman, T. Phan, R. Sakellariou, Compiler-Supported Simulation of Highly Scalable Parallel Applications, in: Supercomputing '99, 1999.

[18] V. Adve, R. Sakellariou, Application representations for multi-paradigm performan ce modeling of large-scale parallel scientific codes, International Journal of High Performance Computing Applications 14 (4) (2000) 304–316.

[19] V. Adve, R. Sakellariou, Compiler Synthesis of Task Graphs for a Parallel System Performance Modeling Environment, in: Proc. 13th Int'l Workshop on Languages and Compilers for High Performance Computing (LCPC '00), Yorktown Heights, NY, 2000.

[20] A.Petitet, S.Blackford, J.Dongarra, B.Ellis, G.Fagg, K.Roche, S.Vadhiyar, Numerical libraries and the grid: The grads experiment with scalapack, International Journal of High Performance Computing Applications 15 (4) (2001) 359–374.

[21] G. Allen, T. Dramlitsch, I. Foster, T. Goodale, N. Karonis, M. Ripeanu, E. Seidel, B.Toonen, Cactus-G Toolkit: Supporting Efficient Execution in Heterogeneous Distributed Computing Environments, in: Supercomputing 2001, 2001.

[22] B. Ensink, V. Adve, Language support for coordinating adaptation in distributed systems, Tech. Rep. UIUCDCS-R-2002-2309, University of Illinois at Urbana-Champaign (December 2002).

[23] C. Lattner, V. Adve, The LLVM Instruction Set and Compilation Strategy, Tech. Report UIUCDCS-R-2002-2292, Computer Science Dept., Univ. of Illinois at Urbana-Champaign (Aug 2002).
URL `http://llvm.cs.uiuc.edu/pubs/LLVMCompilationStrategy.pdf`

[24] E. F. Alessandro Fava, M. Bertozzi, Mpipov: a parallel implementation of pov-ray based on mpi, in: Proceedings of Euro PVM/MPI, Springer-Verlag, 1999, pp. 305–311.

[25] POV-Ray, The persistence of vision raytracer, http://www.povray.org/.

[26] J. Linderoth, S. Wright, Implementing Decomposition Algorithms for Stochastic Programming on a Computational Grid, Tech. Rep. ANL/MCS-P909-0101, Mathematics and Computer Science Division, Argonne National Laboratory (Jan. 2001).

[27] S. Vinoski, CORBA: integrating diverse applications within distributed heterogeneous environments, IEEE Communications Magazine 14 (2).
URL `citeseer.nj.nec.com/vinoski97corba.html`

[28] J. C. Browne, S. I. Hyder, J. Dongarra, K. Moore, P. Newton, Visual Programming and Debugging for Parallel Computing, IEEE Parallel and Distributed Technology 3 (1).

[29] F. Vraalsen, R.Aydt, C.Mendes, D. Reed, Performance Contracts: Predicting and Monitoring Grid Application Behavior, Tech. rep., Computer Science Department, Univ. of Illinois at Urbana-Champaign (2001).

[30] G. Shao, F. Berman, R. Wolski, C. Kesselman, S. Young, M. Ellisman, Master/slave computing on the grid, in: Proceedings of the 9th Heterogeneous Computing Workshop (HCW'2000), 2000, pp. 3–16.

[31] F. B. H. Casanova, G. Obertelli, R. Wolski, The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid, in: proceedings of Super Computing 00, 2000.

[32] J. Linderoth, S. Kulkarni, J.-P. Goux, M. Yoder, An enabling framework for master-worker applications on the computational grid, in: Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9), Pittsburgh, Pennsylivania, 2000, pp. 43–50.

[33] J. Flinn, M. Satyanarayanan, Energy-Aware Adaptation for Mobile Applications, in: Proc. 16th ACM Symposium on Operating System Principles, 1999.

[34] J. Loyall, R. Schantz, P. Pal, J. Zinky, M. Atighetchi, Emerging patterns in adaptive, distributed real-time, embedded middleware, OOPSLA 2001 Workshop - Towards Patterns and Pattern Languages for OO Distributed Real-time and Embedded Systems .

[35] R. E. Schantz, J. P. Loyall, C. Rodrigues, D. C. Schmidt, Y. Krishnamurthy, I. Pyarali, Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware, Submitted to Middleware 2003, the ACM/IFIP/USENIX International Middleware Conference.

[36] L. DiPippo, J. Zhang, M. Murphy, V. F. Wolfe, J. Loyall, R. Schantz, C. Rodrigues, J. Parsons, S. Neema, B. Natarajan, , A. Gokhale, Towards reducing the complexity of adaptive real-time large-scale distributed embedded systems, IEEE Workshop on Large Scale Real-Time and Embedded Systems, in conjunction with IEEE Real-Time Systems Symposium (2002).

[37] P. P. Pal, J. P. Loyall, R. E. Schantz, J. A. Zinky, R. Shapiro, J. Megquier, Using qdl to specify qos aware distributed (quo) application configuration, in: Proc. 3rd IEEE Int'l Symp. Object-Oriented Real-time Distrib. Computing, Newport Beach, CA, 2000.

[38] A. P. F. for Quality-Aware Ubiquitous Multimedia Applications, D. wichadakul and x. gu and k. nahrstedt, in: Proc. ACM Multimedia 2002, Juan Les Pins, France, 2002.

[39] X. Gu, K. Nahrstedt, W. Yuan, D. Wichadakul, D. Xu, An XML-based quality of service enabling language for the web, Journal of Visual Language and Computing (JVLC) 13 (1) (2002) 61–95.

[40] G. Kiczales, et al., Aspect-Oriented Programming, in: Proc. European Conference on Object-Oriented Programming (ECOOP), Finland, 1997.

[41] S. Frölund, Coordinating Distributed Objects: An Actor-based Approach to Synchronization, 1st Edition, The MIT Press, Cambridge, Mass, 1996.

[42] C. Lopes, D: A Language Framework for Distributed Programming, Ph.D. thesis, Northeastern Univ. (Nov. 1997).

[43] N. Medvidovic, R. N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Transactions on Software Engineering 5 (4).

[44] N. Medvidovic, Adls and dynamic architecture changes, in: Proc. 2nd Int'l Software Architecture Workshop (ISAW-2), 1996.

[45] J. Magee, J. Kramer, Dynamic structure in software architectures, in: Proc. 4th SIGSOFT Symp. Foundations of Software Engr., 1996.

[46] M. M. Gorlick, R. R. Razouk, Using Weaves for software construction and analysis, in: Proc. 13th Int'l Conf. on Software Engineering, 1991.

[47] R. J. Allen, R. Douence, D. Garlan, Specifying and analyzing dynamic software architectures, in: Proc. 1998 Conf. on Fundamental Approaches to Software Engineering (FASE '98), 1998.

[48] C. Consel, F. Noel, A general approach to runtime specialization and its application to C, in: Proc. POPL '96 Symp. Principles of Prog. Lang., 1996.

[49] B. Grant, M. Philipose, M. Mock, C. Chambers, S. Eggers, An Evaluation of Staged Run-time Optimizations in DyC, in: Proc. ACM SIGPLAN Conf. On Programming Language Design and Implementation (PLDI), 1999.

[50] T. B. Knoblock, E. Ruf, Data Specialization, in: Proc. SIGPLAN'96 Conf. on Programming Language Design and Implementation, 1996.

[51] C. C. R. Marlet, P. Boinot, Efficient, Incremental Run-Time Specialization for Free, in: Proc. SIGPLAN'99 Conf. on Programming Language Design and Implementation, 1999.

[52] T. Yang, A. Gerasoulis, PYRROS: Static task scheduling and code generation for message passing multiprocessors, in: International Conference on Supercomputing, 1992.

[53] J. Basney, M. Livny, High Performance Cluster Computing, Vol. 1, Prentice Hall PTR, 1999, Ch. Deploying a High Throughput Computing Cluster.

[54] R. Wolski, J. Brevik, C. K. G. Obertelli, N. Spring, A. Su, Running everyware on the computational grid, in: Proceedings of SC99, 1999.

[55] S. Vadhiyar, J. Dongarra, GrADSolve - A Grid-based RPC system for Remote Invocation of Parallel Software, Submitted to Journal of Parallel and Distributed Computing, 2003 (2003).

[56] F. Berman, R. Wolski, The AppLeS Project: A Status Report, in: Proceedings of the 8th NEC Research Symposium, 1997.

[57] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, D. Zagorodnov, Adaptive Computing on the Grid Using AppLeS, in: Transactions on Parallel and Distributed Systems, 2003, pp. 369–382.

[58] S. S. Vadhiyar, J. J. Dongarra, Srs - a framework for developing malleable and migratable parallel applications for distributed systems, International Journal of High Performance Applications and Supercomputing (to appear).

[59] Song, Liu, Jakobsen, Bhagwan, Zhang, Taura, Chien, The microgrid: a scientific tool for modeling computational grids, Scientific Programming 8 (3) (2000) 127–141.

[60] H. Casanova, Simgrid: a toolkit for the simulation of application scheduling, in: proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01), 2001, pp. 430–437.