

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

Applications on a Computational Grid 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 adaptive codes can be challenging to design 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 the distributed program that enables one to reason about and describe application-specific adaptation strategies at a high level, using a few key mechanisms. The global representation enables distributed performance metrics and adaptations to be specified in simple, global terms and implemented transparently by the compiler and runtime system. The paper describes the conceptual adaptation framework, the PCL language, and our implementation of the PCL compiler and runtime system. The paper uses two 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

^{*} This work was sponsored by the NSF Next Generation Software program under contract number EIA-9975024. It was also supported in part 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).

1 Introduction

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. Grid applications have to be able to meet stringent performance requirements in the presence of such dynamic changes in runtime conditions. A common strategy used to tolerate such changes is *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. Some key challenges include the increased complexity of the applications, the need for developing and implementing 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 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, Fortran, C++ or Java) 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 automatic coordination of adap-*

tation operations on different processes: The STG provides a single logical view of the entire distributed computation, and allows an adaptation operation that affect multiple distributed processes to be specified at runtime by a single process. The compiler and runtime system automatically coordinate the distributed operations required to perform the adaptation.

- *Specification of correctness criteria for adaptation:* The framework and language provide mechanisms for specifying a particular class of correctness criteria for each logical adaptation. These criteria serve two purposes. First, they are used by the compiler to enforce the specified correctness requirements (e.g., the programmer can specify that a particular adaptation should not happen while an instance of a particular communication operation is pending, or to specify that an adaptation should be deferred until all processes have completed a particular subset of the computation). Second, for an adaptation that affects multiple processes, they enable the compiler to coordinate the adaptations correctly across those processes.
- *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, performance metrics can be specified in global terms using the STG, and metrics for remote processes are automatically retrieved by compiler-generated code.

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.

We originally built a simple prototype PCL compiler using Fortran as the base language, but are currently developing a more sophisticated version of the compiler in the SUIF infrastructure that can support C, Fortran, C++, and Java applications. The new PCL compiler and runtime system support local and remote adaptation operations specified via the STG, and local and remote performance metrics. (Events and correctness criteria are not yet supported by the system.) The compiler and runtime system use a sophisticated asynchronous strategy for gathering remote performance metrics and performing remote adaptation operations in a parallel thread, so as not to interfere with the communication operations or parallelism of the target application.

The experimental results we present focus on illustrating the potential benefits of PCL (in qualitative terms), and evaluating the potential costs in terms of the runtime overhead of the PCL mechanisms for remote monitoring and adaptation. 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 strategies used. One of our longer term goals is to develop a collection of adaptive codes and experiment with strategies for those codes, within the PCL framework.

For our experiments in this paper, we use two parallel applications that we have made adaptive: a simple parallel fractal code and a parallel version [22] (MPIPOV) of a sophisticated ray tracing code called Povray [23]. Both codes are written using MPI [24]. (We have manually experimented with a third parallel code for stochastic optimization called ATR [25], which was specifically developed for Grid computing, but it is written in C++ which is not yet supported by the PCL compiler and we do not present results for it here.)

Our use of PCL for these examples shows that it is feasible to write simple centralized 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 these codes. Overall, 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 adaptation operations targeted in this work include any programmatic change in program 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. Finally, although general runtime changes to the STG would require the compiler to manipulate some run-

time representation of task graphs (or runtime code generation) some of the most important adaptation mechanisms in the framework can be implemented entirely with static code generation.

The adaptation framework we propose is potentially useful, independent of the existence of the 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 how its realization in PCL.

2.1 *Static Task Graphs*

A *task* is a sequence of instructions in a program containing no internal parallelism, no internal synchronization operations, and 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 only arises 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). Figures 1 and 2 show the static task graphs for 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 containing 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 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.

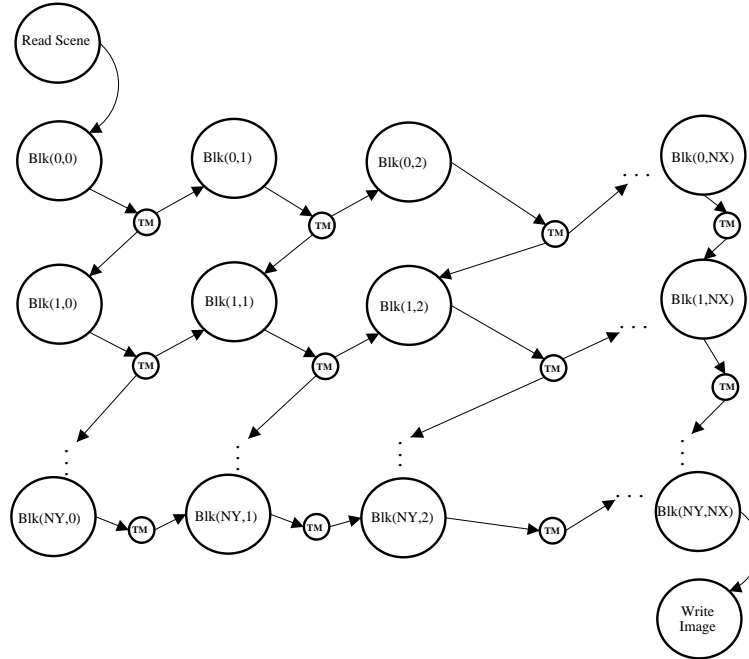


Fig. 1. Static task graph for MPIPOV

The STG by itself does not fully specify program behavior. A second essential 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 multithreaded 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; variants of this policy are used by many master-worker programs. The PCL language and compiler do not consider the scheduling policy directly, but assume that it is the responsibility of the target program.

It is worth underscoring 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 some subset of workers must send their completed data compressed to the master. This requires executing a compression task at the worker for send completed data messages and a decompression task at a master for some of the 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 certain correctness criteria for those operations. 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 a $pid = -1$ 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, conceptually the 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 happen together 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.

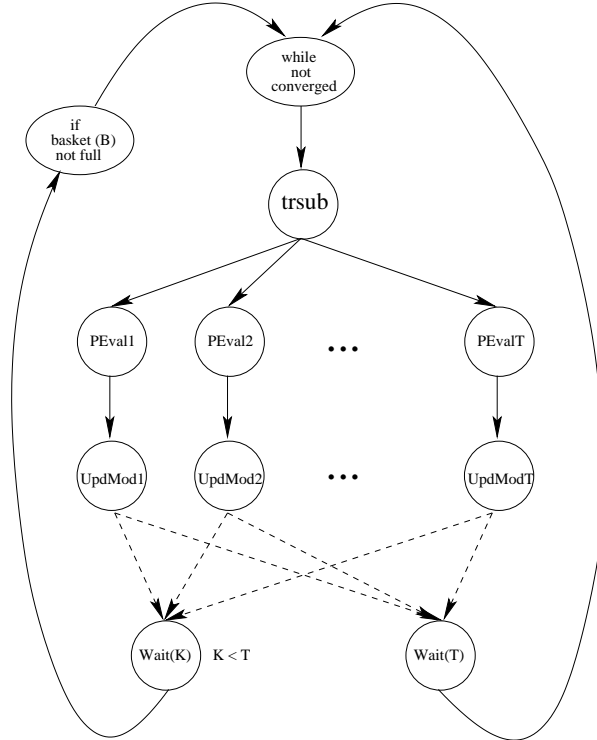


Fig. 2. Static task graph representation of asynchronous parallel stochastic optimization (ATR).

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 2), 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 the MPIPOV and ATR examples. Section 5 describes additional applications that are captured by our framework.

MPIPOV [22] is a master-worker ray tracing code that renders an image in parallel by decomposing it into an array of blocks. Figure 1 shows the precedence structure of the blocks of the rendered image, labelled $\text{Blk}(y, x)$ in the figure. As blocks are rendered, control passes through TM (Task Manager) nodes, which are executed by the master, and one or more dependent blocks can be assigned to workers for rendering. 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 [25] code is a very long running code that adapts to changing count of available processors by increasing or decreasing available parallelism, in two ways: (1) changing the number of concurrent iterations (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 Correctness Criteria in the Framework

Automatically verifying correctness of adaptations is in general impossible, and even in simple cases extremely difficult. Instead, our framework includes correctness criteria that can be specified by a programmer and verified or implemented by a compiler, to ensure certain types of adaptation integrity. The framework focuses on criteria that can be described in terms of the state of program execution relative to the STG.

Some common conditions that arise for different types of adaptation strategies are as follows. 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]). An adaptation that modifies a parameter used by some set of tasks may have to ensure that corresponding instances of all the tasks use the same value of the parameter. The simplest way to ensure this is to create a copy of the parameter that is used by all corresponding instances of the tasks, allowing differ-

ent concurrent instances of the task group to use different values. (For example, the number of tasks in `ATR` must be used by all the `PEval` tasks and `UpdMod` tasks within a particular iteration. Different iterations can use different values.)

We define a common mechanism that can be used to specify all these kinds of correctness criteria. It would be attractive to develop a formal classification of the possible types of errors and use that to evaluate the comprehensiveness of our mechanism, but that is beyond the scope of this paper.

Def. (region): A region is a connected subgraph of the STG. The simplest region is a single task with a single incoming and outgoing edge. A region may also consist of multiple tasks and the edges that connect them.

Def. (region-in edge): For a region R , a region-in edge is an edge from $a \rightarrow b$ such that $a \notin R$ and $b \in R$.

Def. (region-out edge): For a region R , a region-out edge is an edge from $b \rightarrow c$ such that $b \in R$ and $c \notin R$.

A region may have multiple region-in and region-out edges. For completeness we define an **intra-region** edge to be an edge $a \rightarrow b$ such that $a, b \in R$. All other edges are non-region edges.

If any adaptation modifies the subgraph of a region R or changes the values of parameters used by tasks within R , we say that it adapts the region. We say that a task t is added to a region R if it is added to the STG and any edge e incident on t is also incident to some other task $b \in R$. Likewise we say t is removed from R if t is removed from the STG and t was in the subgraph for R .

We will say that a task t of a STG is *active* if there is some thread currently executing an instance of the task. A region R is active if some task $t \in R$ is active. Finally, we refer to the thread performing an adaptation operation as an adaptor thread and the threads of the target program as target threads. In practice, the adaptor thread may or may not be the same as the target thread. For example, the two would be a single thread when an adaptation operation is invoked synchronously by a target thread, as described in Section 3. In some cases, there may be multiple adaptor threads implementing a particular adaptation operation. Some of the criteria below are only useful in a subset of these cases.

Correctness criteria express constraints on a particular adaptation operation with respect to active regions. Each criterion for a particular adaptation specifies a region and a correctness policy that must be enforced when performing the adaptation on that region. We define four possible policies:

Immediate(region) : The adaptation operation can be performed immediately with no constraints on the state of execution relative to this region. This can be useful simply to allow programmers to make explicit the correctness rule for every region affected by an adaptation.

DeferAndBlock(region): Delay an adaptation operation and block the adaptor thread until the region is no longer active.

DeferAndContinue(region): Same as the last, but queue up the adaptation and allow the adaptor thread to resume execution. Queued adaptations are applied to a region in FIFO order when the region is no longer active.

ApplyAndCopy(region, V_1, \dots, V_n): Generate local temporary copies of the specified parameters $V_1 \dots V_n$, and insert code to copy the values into the local copies along every region-in edge. Use these copies within the region instead of the originals, so that an adaptation can modify the originals without affecting the correctness of the adaptation. The tasks of the region must not modify these parameters; checking this is optional within the compiler.

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. Fundamentally, PCL provides mechanisms that a programmer can use to write 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. 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 correctness 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 correctness criteria described as part of the framework require fairly sophisticated compiler techniques (and significant additional research) to be practical. We deferred defining the specific syntax and semantics of these criteria within the language until we have more experience with a compiler implementation. This is discussed briefly in the section on future work 7.

Non-adaptive applications can take advantage of PCL with only minimal modifications to the target application, e.g., adding labels and, in some cases, adding calls to invoke adaptation from specific points in the target code. The language otherwise keeps the adaptation specifications completely separate from the target application. (Of course, design changes to a non-adaptive target application are often needed simply to make an adaptation possible, e.g., enabling a parameter to be changed more flexibly during program execution.)

3.1 Overview of PCL

Syntactically, PCL is a small extension to an existing language such as C, C++, Java, or Fortran. (The dependence on the target language is primarily because code fragments manipulating parameters of the target program may have to be included in the PCL code.) We focus in this paper on using PCL with C programs.

The grammar shown in Figure 3 describes PCL. There are four primary constructs in PCL: *Adaptor*, *ControlParameters*, *Metric* and *Event*. These constructs are illustrated in Appendix A, which shows the complete 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.

Each logical adaptation operation that can make one kind of change to the target program is represented as one *Adaptor*. In C and Fortran, each adaptor is global, i.e., there is one logical instance of an adaptor at runtime. In object-oriented languages like C++ or Java, an *Adaptor* may be global or may specify a target class in the original application and provide adaptive functionality for that class. In the example of Appendix A, the *FractalAdaptor* is a single global adaptor for the complete program that performs two different adaptations. (These could potentially have been written using two different *Adaptor* constructs.)

The actual adaptation logic of an *Adaptor* is implemented by one or more *adapt* methods. *Adapt* methods are functions declared inside an *Adaptor* construct. An *adapt* method can be called synchronously by manually inserting a call to the method in the target application, or may be invoked asynchronously as event handlers in response to runtime events. The *adapt* method uses 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.

3.2 Identifying Tasks and the STG

Specifying each of the six adaptation operations defined in the previous section only requires identifying the relevant tasks in the task graph. In particular, it does not require specifying the complete STG. A programmer can identify a particular *task* simply by a statement label (the task consists of the labeled statement) or a function name. The statement label can be put on a compound statement to identify a contiguous sequence of statements as a single task. (In order to use labels across files, a label may include the module or method name as a qualifier.) The compiler

can relatively easily construct tasks for the remainder of the code (if needed) using a default granularity of the tasks (usually, the largest tasks that meet the definitions in Section 2.1). The programmer need not be aware of the existence of these other tasks. The PCL syntax therefore achieves a compromise between full programmer specification of task graphs and no programmer input.

Nevertheless, to meet all the goals of PCL, having a complete task graph specification can be useful. Implementing correctness criteria applicable to specified regions of the graph requires knowledge of the task graph structure. More broadly, we envision PCL as the basis for a complete programming environment that allows programmers to visualize and manipulate the task graph of the target application, and use it both to design adaptation strategies and to implement them using PCL.

One of our ongoing research goals is to use a combination of programmer specifications and compiler support to extract task graphs automatically for a range of different distributed programming models. This goal is discussed briefly in Section 7. In the rest of this paper, we assume that no additional information about the STG is required other than the relevant tasks using in the adaptation operations.

3.3 Task Graph Operations

Rules 13–15 in the grammar of Figure 3 show the syntax of the `AddTask`, `RemoveTask`, and `ReplaceTask` operations in PCL. The task to be added can be specified either by a statement label or a function name. In the latter case the actual arguments of the task must be given for the `AddTask` operation to properly construct the function call. The compiler can check easily that the constructed call is valid at the specified site. The `RemoveTask` operation only requires the site and task name. (The implementation of these operations is described in Section 4.)

A *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 to a site, implicit control flow edges are created from the code immediately preceding the site to each of the tasks at the site, and from each task at the site to the code immediately following the site. 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 to be specified: they are implicitly defined by the choice of the adapt *site*.

The example in Appendix A shows the use of `AddTask` and `RemoveTask` operations. For example, the `AddTask` operation near the end of the example inserts a task representing a function call to the function `CompressTask` with the specified parameters at the statement labelled L12 within the file `Worker.c`.

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 *ChangeParam* 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 adaptor 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) adaptor ::= Adaptor name {
                                ctrl-parameters metric event
                                decls adapt-method-list } ;
(2) ctrl-parameters ::= ControlParameters {
                                identifier-list } ;
(3) metric ::= Metric name (site,
                                function(param-list)+);
(4) event ::= Event name (eventpolicy, function ) ;
(5) eventpolicy ::= EP_Changed (name)
(6) | EP_Threshold (name,threshold-value)
(7) | EP_Percent (name,percent-value)
(8) | EP_Value (name,constant-value)
(9) adaptop ::= addtask
(10) | removetask
(11) | replacetask
(12) | changeparams
(13) addtask ::= AddTask (pid, site, taskLabel
                                [, ( param-list )] );
(14) removetask ::= RemoveTask (pid, site, taskLabel) ;
(15) replacetask ::= ReplaceTask (pid, site, taskLabel,
                                taskLabel [, ( param-list )] );
(16) changeparam ::= ChangeParameter (pid, lvalueExpr, rvalueExpr) ;
(17) named-region ::= Region name =
                                { site-list };
(18) named-params ::= ParameterList name = { identifier-list };
(19) taskLabel ::= module-name:label
(20) | funtionName
(21) site ::= module-name:label

```

Fig. 3. PCL Grammar

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.

The syntax of a metric declaration is given in rule 3 of the 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 libraries provide several basic instrumentation functions 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. (One of our near-future goals is to use the Network Weather Service toolkit [5] as a basis for building a more comprehensive library of PCL performance metrics for Grid programs.)

The runtime library also provides functions to retrieve the value of a metric either locally or remotely from a specified process. The adaptor 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.

To minimize the overhead of measurement, the compiler should inline the measurement function at the measurement site by default. An alternative is to include an optional argument (not shown in the current syntax) that would turn off the inlining of measurement functions for a particular site. This would allow the compiler to treat each metric as a task, allowing dynamic changes to the instrumentation via the standard adaptation operations (e.g., `AddTask` and `RemoveTask`).

The `FractalAdaptor` 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 `GetMetricRemote` 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 `GetMetricRemote()` 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 adaptor function 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 4-8 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 sockets for communication. 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 is being written using the SUIF2 compiler infrastructure [26] from Stanford University.

The compiler maps the PCL language constructs to runtime library mechanisms to implement the semantics of the language. The details of the runtime library are further discussed in section below. 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 in the following subsections. Events and correctness criteria are not yet supported by the system, and are briefly discussed in the section on future work.

4.1 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 remote adaptation 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. The different RTL instances must communicate via message passing. (We use sockets, both for portability and to ensure that our communication does not interfere with any communication mechanisms in the base application.) 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 RTL instance that presides over the address space where the operation is to take place. 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. The thread that requested the runtime operation is blocked until the remote operation completes.

Blocking the user thread for the entire latency of a series of remote operations can be extremely expensive. In order to hide the latency of remote operations, each RTL instance uses a second thread called a support thread. The call to the `Adapt()` function in the main application code is replaced by a call to a wrapper function that simply enqueues the `Adapt()` function and returns. The support thread dequeues and executes these `adapt` function invocations, allowing the adaptation operations to be activated in an asynchronous manner. (The programmer can choose whether to execute an `adapt` method synchronously or asynchronously.) Now, when the main RTL thread performs remote operations, the support thread is blocked and not the original user thread. (For a complete implementation, additional support is required for coordinating the asynchronous adaptation operations with respect to the execution state of the user threads. Implementing this support is a one of the priorities for our future work.)

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 initially includes the original task (if any) at that site. (A `nop` statement followed by a second label can be inserted immediately after the site label to ensure that the set will be 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.

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 *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 *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.

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 [27]. (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 Code Generation at Adaptation Sites

The compiler 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. First any original task at the site is

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

removed and, if necessary, a call to the runtime library is inserted at the program entry point to insert the original task in the appropriate set. Second, 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 invoke each task in the set. Each task is passed the same actual parameters which must be specified by either the original task (if present) and all the `AddTask` commands affecting this site. The compiler will generate an error if the actual parameters specified in `AddTask` are not in scope at the adapt site or do not match the other `AddTask` commands for 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 then replaces each `AddTask` and `RemoveTask` operation 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 an `RemoveTask` immediately followed by an `AddTask`. 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.

4.3 Metrics

A PCL metric declaration indicates one or more program locations and precisely one function to invoke for each location. The compiler automatically inserts the metric function invocations at the given locations. As an option the compiler could inline each metric function for greater efficiency. Also, the compiler could treat each metric function as a task and each invocation point as an adapt site. This would allow the measuring code to be dynamically switched on and off as needed. This flexibility could be useful for programs that adapt only periodically or on a very coarse grain.

Each metric must also be registered with the runtime library at execution time. This is done by inserting a procedure call at the program entry point to register the metric along with a pointer to the metric. 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.

The approach of inserting user-specified functions allows the user to write arbitrarily complex metric functions. The runtime time library can supply many commonly used metrics, such as the load average of local and remote machines, network performance metrics, counters and timers. These general metrics can be use by the application developer to build metrics of greater complexity without duplicating large amounts of code. Furthermore, we intend to explore leveraging existing performance modeling tools such as the Network Weather Service [5] to make a richer variety of performance metrics available to PCL codes.

4.4 *Control Parameters*

It is quite common for an application variable to dynamically control some aspect of the program execution 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.

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.

4.5 *Adapt Methods*

An adapt method is any method declared with the `AdaptMethod` keyword that invokes one or more PCL commands such as `GetMetric` or `AddTask`. The programmer calls this function just like an ordinary function.

As described below the run time library supports synchronous and asynchronous execution of adapt methods. If the programmer chooses to use asynchronous method invocation the compiler must transform the method so that the runtime library's support thread can invoke it and pass only a single argument. To do this the compiler generates a structure which contains all the function arguments. The structure is packed before calling the adapt method and unpacked inside the method as needed. A pointer to the structure is passed to the adapt method as the only argument.

The structure itself can be heap allocated (the compiler will add a call to free the structure at the exit point of the adapt method) or a small buffer of preallocated structures can be created for more efficient execution.

5 **Experimental Results**

In this section we present our experience with two applications: Fractal and POVray. We focus primarily on measuring the effectiveness of our implementation and the benefits of using PCL and the runtime library. We emphasize again that our goal is not to evaluate the efficacy of particular adaptation strategies for these applications.

5.1 Applications

5.1.1 Fractal

The fractal program is a 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.

To compute the image the worker expands a sequence based on one point in the complex plane. If the value of the sequence remains bounded then the point is likely a member of the Mandelbrot set (if a special condition is met the algorithm can immediately determine that the point is not a member of the set without further expansion). The value *MaxIterations* sets an upper limit on how far the sequence will be expanded. A lower value of this parameter will decrease the amount of work required to compute a single point in the set at the cost of some false positives.

We added two adaptations to the fractal program. First, adapting the control parameter *MaxIterations* will change the amount of work done by a worker in exchange for slightly degraded (or improved) image quality. Each worker has a copy of this control parameter. PCL allows the master to adapt the parameter for an individual worker. The second adaptation compresses the computed data on the worker prior to sending the completed work to the master. This involves adding or removing a compression task on an individual worker. The master determines whether the data is compressed based on the message size.

Appendix A shows the PCL code we used to implement these adaptations. The master periodically executes the adapt function after a fixed number of image slices have been computed. The adapt function gathers performance information from all of the workers and if necessary will adapt an individual worker to improve the performance. Currently we add or remove the compression task based on the load of the machine the worker is executing on. In the future, a more sophisticated strategy would be to consider both the network bandwidth and the CPU load in choosing whether to compress the data.

PCL provides four 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 both local and remote adaptation operations to be specified simply and uniformly within a single adaptation thread, and the distributed operations are automatically performed by the compiler. Third, 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. Fourth, the master can retrieve remote metric values simply by referring to them by name and specifying a remote process id. (A distributed performance library could do this 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*

The POV-Ray application is a raytracing application code available in the public domain, for which a parallel MPI version (MPIPOV) has been created [22]. The MPI version utilizes a straightforward master-worker application model to assign some number of fixed-size blocks of image regions to workers for rendering. In the original, non-adaptive version of MPIPOV, the number of blocks assigned to a given worker is a compile-time constant.

MPIPOV performs adaptive supersampling (not to be confused with any adaptation related to PCL) on the rendered scene to divest the final image of aliasing artifacts. This supersampling induces a partial order on the rendering of blocks, where each block depends on its left and upper neighbors.

We propose a simple load-balancing adaptation strategy for MPIPOV. In particular, 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 and a local *ChangeParameter* operation. The metric query is required to obtain the average worker completion time for a given worker, and the *ChangeParameter* is used to modify the number of image blocks assigned to a given worker. No distributed adaptation is needed.

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. (This particular adaptation strategy does not use remote adaptation operations.) It also provides the same intangible benefits as for Fractal.

5.1.3 ATR

Our current implementation of the PCL compiler is limited to C and the ATR application is written in C++, as a result we are unable to get experimental results with ATR at this time.

5.2 Overheads of Using PCL

The first goal of our experiments is to evaluate the performance cost of using PCL as a basis for Grid applications. As a measure of this cost, we essentially 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. In the first version, the `Adapt()` method executes the loop that requests the elapsed time metric from each worker but does not do any adaptation operations. 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). The third version is the original application itself, without using PCL at all.

Figures 4 and 5 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 1 (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 5 has only two plots because the adaptation code performs no remote operations other than remote metric value requests.)

5.3 Effect of Synchronous versus Asynchronous Adaptation

The second goal of our experiments is to evaluate the importance and effectiveness of our strategy of executing `adapt` methods in the “background” in a separate thread per process, instead of synchronously within the control flow of the original application.

In the case of Fractal, the use of remote adaptation operations can significantly affect the relative performance of the synchronous and asynchronous strategies. We therefore consider a best-case and a worst-case scenario from the viewpoint of the synchronous version. (MPIPOV, in contrast, uses purely local adaptation operations which have negligible runtime overheads, so this distinction is not important.) The

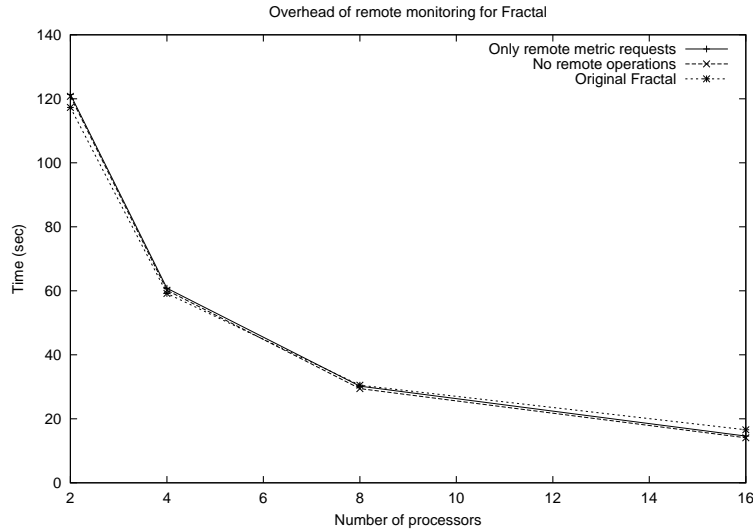


Fig. 4. Overhead of remote monitoring for Fractal

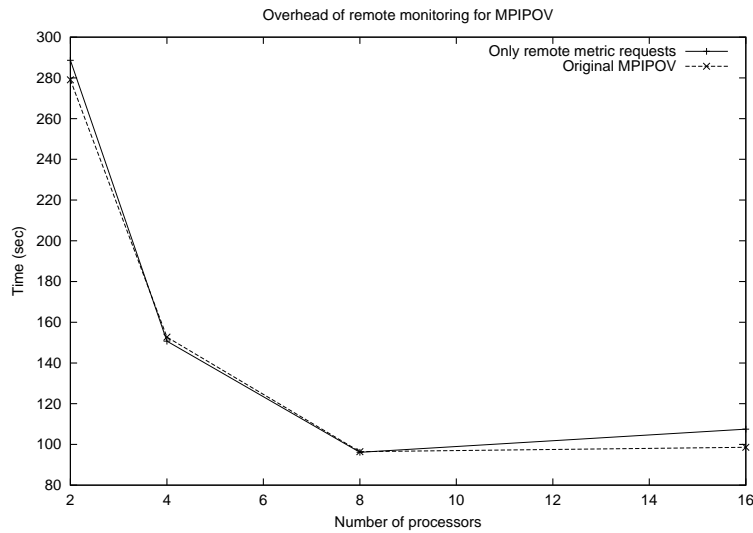


Fig. 5. Overhead of remote monitoring for MPIPOV

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.

Figure 6 shows the execution time for synchronous and asynchronous invocations of the worst case adapt method. 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. The synchronous version does not suffer from this problem 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).

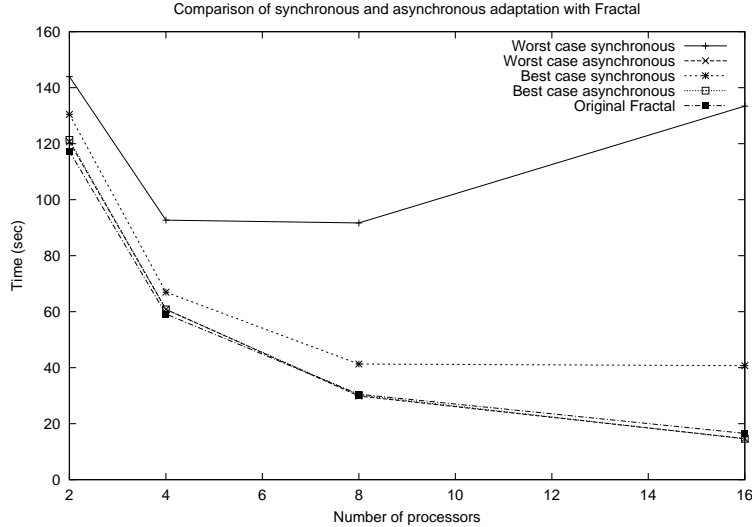


Fig. 6. Comparison of synchronous and asynchronous adaptation for the Fractal application

Figure 7 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.

6 Related Work

There are two broad areas of related work: (a) language and compiler efforts that have some overlap with techniques used in our work, and (b) runtime platforms aimed at supporting adaptive applications.

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]. This project is developing a broad software platform for Grid applications, and includes a wide range of efforts but their work on a Grid application development environment is the most relevant here. They propose an environment in which an application is continuously adapted at runtime to changes in Grid resources. They have described the use of performance contracts for guiding dynamic reconfiguration and optimization [28]. They

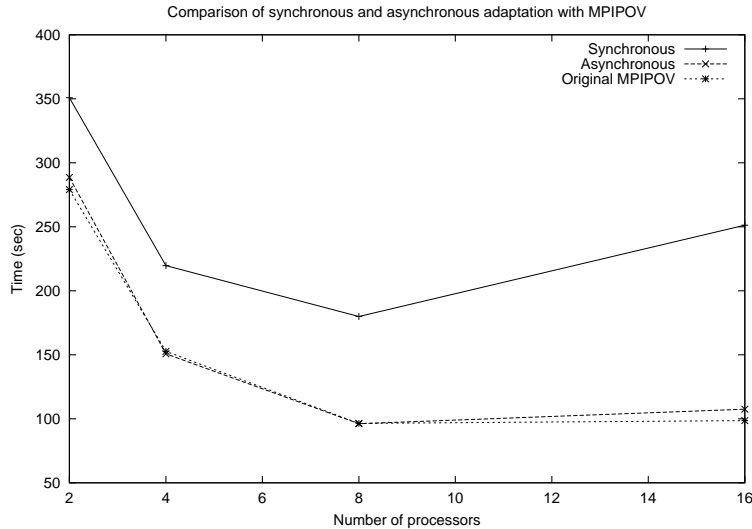


Fig. 7. Comparison of synchronous and asynchronous adaptation for MPIPOV

plan to incorporate some form of compiler support for suitable Grid programming models, performance prediction, and dynamic optimization, although that work is still at a preliminary stage. We believe that the PCL language and task-graph-based 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 static and dynamic compilers and to the runtime system (e.g., it might enable the automatic generation of performance contracts).

An alternative strategy for simplifying the development of adaptive applications is to use libraries tailored for specific classes of applications. For example, the GRAIL project has developed template libraries (AMWAT [29] and APST [30]) for master-worker and parameter sweep codes. Similarly, the Condor project has developed a class library called Mw for master-worker Grid codes [31], 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 much 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. Another example is CACTUS-G [21], a Grid-enabled version of the popular CACTUS simulation framework. The Grid version includes sophisticated adaptation strategies that are completely transparent to applications built on top of CACTUS. In this case, however, we believe that PCL would provide a valuable basis for implementing the adaptation strategies in CACTUS-G. In particular, the CACTUS code is itself a sophisticated parallel code and that code could be considered the target for PCL.

There has been extensive work in the software engineering community on Architecture Description Language (ADLs) that allow the architecture of a component-based

software application to be specified separately from the individual components of that application ([32] provides a fairly comprehensive overview and comparison of ADLs). Of these, some ADLs such as C2 [33], Darwin [34] and Weaves [35] allow the software architecture 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 [36], but with the caveat that they are restricted to component-based applications. The PCL language can be thought of as a simple and less formal version of a dynamic ADL that can be applied to applications not based on components (like many scientific and distributed applications). Furthermore, PCL is specifically tailored to support dynamic adaptation based on runtime performance monitoring.

In some ways, our work is similar to research on program specialization, some of which provides techniques to specialize code to specific data values or patterns at runtime [37–40]. Our goal, however, is to provide higher-level support for performance-based adaptation.

There have been a number of languages, particularly parallel programming languages, based on graphical representations of parallelism (e.g., [27,41]). 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., the neither 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.

Our language strategy for specifying adaptation behavior separate from the target distributed application is essentially a form of aspect-oriented programming (AOP) [42]. Similar language extensions have been previously proposed for specifying synchronization and/or communication behavior of a class separately from the other methods of that class [43,44]. These language extensions served as models for this part of PCL.

Finally, there has been extensive work on runtime platforms for supporting adaptive Grid applications at the level of middleware, runtime libraries, and network layer support. We can only cite a limited sample here. Globus [9] and Legion [11] each provide a wide range of runtime services including resource discovery, scheduling, authentication, and job migration. Condor [45] provides facilities for job scheduling and migration for workstation networks and for Grid computing. The EveryWare toolkit [46] 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 Network Weather Service (NWS) [5] provides extensive support for runtime performance monitoring and for statistical predic-

tion of the future performance characteristics of networks and hosts over specified intervals of time. 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. The AppLES effort has developed techniques and tools for application level scheduling [47]. MicroGrid [48] and SimGrid [49] are simulators for performance analysis of Grid applications. Our work is complementary to these and other similar efforts in the sense that adaptive Grid applications (written in PCL or otherwise) will require extensive support from such runtime systems.

7 Summary and Future Work

In this paper, we described the design and implementation of Program Control Language, or PCL: an abstract, global representation for reasoning about and describing adaptive behaviour in distributed applications, remote operations for performance gathering and distributed adaptation, and correctness criteria for distributed adaptations. We also described a runtime library which provides the mechanisms required to implement PCL. Furthermore, the experimentally measured overheads indicate that PCL and the runtime library have a minimal impact on the total execution time of the application. We have shown that an intelligent asynchronous implementation of adaptation operations is crucial to achieving these low overheads. Finally, PCL provides the intangible benefits of simpler, higher-level descriptions of distributed adaptive behavior.

There are several directions we intend to pursue in the future. Our near-term goal is to enhance the capabilities of the PCL compiler and runtime system. We will augment the compiler to allow synchronization of arbitrary asynchronous adaptation operations, and to implement the correctness criteria expressible in our framework. We aim to include a richer set of performance metrics in the runtime library, using existing toolkits like the Network Weather Service [5] for supporting Grid-specific performance monitoring capabilities, and building higher-level metrics for specific application domains.

Longer-term, the lack of publicly available adaptive applications is a major bottleneck for our work. To drive our continued development of PCL, we aim to develop larger, more complex applications with PCL from several different areas such as multimedia codes, grid codes, and network computing applications.

The static task graph is an important component of the framework that should allow more complex static analysis and optimizations of distributed codes. We have two research goals in this regard. The first is to automate the construction of task graphs for different classes of distributed applications, or partially automate the process using some programmer input. The second is to develop a prototype of a full-fledged programming environment using the static task graph as a graphical basis for designing and implementing adaptation strategies.

Appendix A: PCL Code for Fractal

```
Adaptor FractalAdaptor {
    ControlParameters { MaxIterations; };
    Metric elapsedTime(Worker.c:L5, StartTimeStamp(),
                      Worker.c:L6, EndTimeStamp());

    /* no events, no rules */

    AdaptMethod void Adapt(int numWorkers) {
        /* Compute average elapsed time of all workers */
        double avgTime = 0;
        for (pid = 1; pid <= numWorkers; ++pid) {
            adaptInfo[pid].time = GetMetricRemote(elapsedTime, pid);
            avgTime += adaptInfo[pid].time
        }
        avgTime = avgTime / numWorkers;

        /* Adapt each worker as necessary */
        for (pid = 1; pid <= numWorkers; ++pid) {
            /* Increase or decrease MaxIterations for a worker
               according to its relative performance */
            if (adaptInfo[pid].time > 1.2 * avgTime &&
                adaptInfo[pid].iter == high_MaxIterations) {
                ChangeParameter(pid, MaxIterations, low_MaxIterations);
                adaptInfo[pid].iter = low_MaxIterations;
            }
            else if (1.2 * adaptInfo[pid].time < avgTime &&
                    adaptInfo[pid].iter == high_MaxIterations) {
                ChangeParameter(pid, MaxIterations, high_MaxIterations);
                adaptInfo[pid].iter = high_MaxIterations;
            }
            /* Add or remove compression for a worker according to the
               cpu contention on the worker's machine */
            loadAvg = GetMetricRemote(LoadAverage, pid);
            if (loadAvg[LOAD_AVG_1MIN] > 1.15 && adaptInfo[pid].cmpr == 1) {
                RemoveTask(pid, Worker.c:L12, CompressTask);
                adaptInfo[pid].cmpr = 0;
            }
            else if (loadAvg[LOAD_AVG_1MIN] < 1.07 && adaptInfo[pid].cmpr == 0) {
                AddTask(pid, Worker.c:L12, CompressTask(header, data, pSize));
                adaptInfo[pid].cmpr = 1;
            }
        }
    }
}
```

Appendix B: PCL Code for MPIPOV

```
Adaptor MPIPOV_Adaptor {
    ControlParameters { TaskSizeForWorker[]; };

    Metric elapsedWorkerTime(MPIren.c:ELAPSED_START, StartTimeStampWithMovingAvg(),
                             MPIren.c:ELAPSED_END,   EndTimeStampWithMovingAvg());

    /* no events */
    /* no rules */

    AdaptMethod void Adapt(int workerID) {

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

        if(HaveAllWorkerTimes()) { /* Ensure we have values for all workers */
            /* CalculateWorkerAverage 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]. */

            ChangeParameter(LOCAL /* -1 */, TaskSizeForWorker[workerID],
                            awardRatio * BASELINE_TASKS_FOR_WORKER);
        }
    }
}
```

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 heterogeneous 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 performance 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.Petit, 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. Damlitsch, 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] E. F. Alessandro Fava, M. Bertozzi, Mpipov: a parallel implementation of povray based on mpi, in: Proceedings of Euro PVM/MPI, Springer-Verlag, 1999, pp. 305–311.
- [23] Povray, <http://www.povray.org/>.
- [24] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, MPI: The Complete Reference, 2nd Edition, MIT Pres, 1998.
- [25] 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).
- [26] SUIF:, <http://suif.stanford.edu/suif/suif2/index.html>.
- [27] 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).
- [28] 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).
- [29] 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.
- [30] 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.
- [31] 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, Pennsylvania, 2000, pp. 43–50.

- [32] N. Medvidovic, R. N. Taylor, A classification and comparison framework for software architecture description languages, *IEEE Transactions on Software Engineering* 5 (4).
- [33] N. Medvidovic, Adls and dynamic architecture changes, in: *Proc. 2nd Int'l Software Architecture Workshop (ISAW-2)*, 1996.
- [34] J. Magee, J. Kramer, Dynamic structure in software architectures, in: *Proc. 4th SIGSOFT Symp. Foundations of Software Engr.*, 1996.
- [35] M. M. Gorlick, R. R. Razouk, Using Weaves for software construction and analysis, in: *Proc. 13th Int'l Conf. on Software Engineering*, 1991.
- [36] 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.
- [37] 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.
- [38] 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.
- [39] T. B. Knoblock, E. Ruf, Data Specialization, in: *Proc. SIGPLAN'96 Conf. on Programming Language Design and Implementation*, 1996.
- [40] 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.
- [41] T. Yang, A. Gerasoulis, PYRROS: Static task scheduling and code generation for message passing multiprocessors, in: *International Conference on Supercomputing*, 1992.
- [42] G. Kiczales, et al., Aspect-Oriented Programming, in: *Proc. European Conference on Object-Oriented Programming (ECOOP)*, Finland, 1997.
- [43] S. Frölund, *Coordinating Distributed Objects: An Actor-based Approach to Synchronization*, 1st Edition, The MIT Press, Cambridge, Mass, 1996.
- [44] C. Lopes, D: A Language Framework for Distributed Programming, Ph.D. thesis, Northeastern Univ. (Nov. 1997).
- [45] J. Basney, M. Livny, *High Performance Cluster Computing*, Vol. 1, Prentice Hall PTR, 1999, Ch. Deploying a High Throughput Computing Cluster.
- [46] R. Wolski, J. Brevik, C. K. G. Obertelli, N. Spring, A. Su, Running everywhere on the computational grid, in: *Proceedings of SC99*, 1999.
- [47] F. Berman, R. Wolski, The apples project: A status report, in: *Proceedings of the 8th NEC Research Symposium*, 1997.

- [48] Song, Liu, Jakobsen, Bhagwan, Zhang, Taura, Chien, The microgrid: a scientific tool for modeling computational grids, *Scientific Programming* 8 (3) (2000) 127–141.
- [49] 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.