Coordinating Adaptations in Distributed Systems

Brian Ensink
ensink@cs.uiuc.edu
University of Illinois at Urbana-Champaign

Vikram Adve
vadve@cs.uiuc.edu
University of Illinois at Urbana-Champaign

1. Introduction

Applications in distributed and mobile environments require flexibility and robustness in the presence of ever changing performance characteristics of the system. Many applications use runtime adaptations to improve overall performance. An application can adapt its behavior to make the most effective use of limited resources [**?**, **?**], or to improve Quality of Service provided to the user [**?**, **?**, **?**], or to adapt to changing performance conditions on systems and networks such as in a Computational Grid [**?**].

Implementing adaptive distributed applications, libraries, or middleware can be a challenging technical and software development problem. Adaptation strategies require addressing a number of difficult issues including performance monitoring and prediction, adaptation mechanisms for changing the behavior of the code, resource management and scheduling strategies for underlying system resources, and coordination techniques for coordinating adaptations across multiple distributed processes. While there has been a great deal of research on the first three aspects of adaptation (see Section **??**), there has been little work that we know of that addresses the specific problem of coordinating adaptations. Middleware or libraries that encapsulate adaptation behavior may hide this complexity from the application, but have to face the same challenges in their own implementation.

Coordination is critical to maintain the correctness of an application during adaptation. In particular, adaptation in a distributed application can require the affected processes to communicate, synchronize, and perhaps schedule operations for the future. Such coordination requirements cause two potentially important difficulties. First, explicit coordination via inter-process communication and synchronization can significantly increase the complexity of adaptation in distributed programs. Second, reasonable manual implementations would typically use barriers or distributed locks, but these mechanisms add overhead during normal (non-adaptive) execution.

In this paper, we propose a novel strategy for coordinating adaptations in distributed applications that shifts much of the burden to a sophisticated, transparent runtime algorithm. The programmer uses simple directives to specify criteria for *when an adaptation must happen*, as a function of the relative computational progress of the affected processes (without having to write any explicit communication or synchronization code). The adaptation is then scheduled automatically and efficiently by a compiler and runtime system. We develop a sophisticated runtime algorithm that coordinates and schedules the adaptation operations *locally and asynchronously* on the different processes, as specified by the program criteria (using simple compiler support to track the progress of the processes).

We evaluate the performance overheads of runtime coordination for two programs: a parallel PDE solver implementing the ghostzones adaptation from Cactus-G [**?**] and a file-transfer program implementing key adaptations from a distributed video server and video tracking code [**?**]. Our results show that the overheads of the runtime coordination are small or negligible, less than 1% in all cases. The key to this performance is the novel asynchronous execution achieved by our runtime coordination algorithm.

Our compiler and runtime support for coordination have been implemented in the context of Program Control Language (PCL), a high-level language for specifying adaptations within a distributed application. This choice is *not* fundamental to this work: the coordination rules could be specified and implemented even if PCL was not used to implement adaptations. The coordination rules rely on identifying control flow regions within the target program, and on tracking the relative execution progress of different processes with respect to instances of these regions.

The next section provides some necessary background on PCL. The subsequent sections describe the language constructs (Section **??**), the compiler support and runtime algorithm (Section **??**), our experimental evaluation (Section **??**) and a discussion of related work (Section **??**). Section **??** concludes by summarizing our contributions.

2. Background: Static Task Graph and PCL

In this section, we provide a brief description of our adaptive framework and an implementation of it called Program Control Language (PCL). PCL and the framework are described in more detail in [**?**]. That work described how

PCL could be used to perform remote adaptation operations automatically, but had *no support for automatically coordinating adaptation operations occurring on multiple processes*. In this paper, we build upon our previous work by developing a transparent coordination strategy that can be used with PCL or with other methods for implementing adaptive distributed programs.

2.1. The Static Task Graph (STG)

The *Static Task Graph* of an application (STG) provides a global view of the control flow of a distributed application. Conceptually the behavior of the application can be altered by changing the graph.

Def. (task): A *task* in a distributed program is a single-entry, single-exit section of code executed by a single thread and containing no synchronization operations.

Def. (static task graph): A *static task graph* (STG) is a directed graph in which each node represents a task and an edge from T_1 to T_2 implies T_1 must complete before T_2 can begin.

Note that a task is a static entity; logically, one or more *instances* of each task are created and executed at runtime, possibly by different threads. A static task graph is similar to a control-flow graph, but for a distributed program. A task may include multiple basic blocks of the control flow graph. This is useful because coarse-grain tasks are often sufficient for representing the relevant aspects of distributed program behavior. Tasks can be either *computation*, *synchronization* or *communication tasks*, where the latter represent instructions executed for explicit interprocess communication.

Edges can represent control flow, synchronization, or both. In particular, *an instance of the edge* $T_1 \rightarrow T_2$ connects a pair of task instances, one each of T_1 and T_2. This edge instance represents ordinary control-flow if the two task instances are executed by the same thread, and represents an synchronization operation otherwise.

The static task graph abstraction of an application need not be explicitly constructed during development (in fact, no STG representation is directly used by the language, compiler or runtime system). The only necessary requirement is for the programmer to understand the control flow of the program around regions of the code whose behavior is directly modified by adaptation operations, in order to specify "entry points" into those regions. (In PCL, these regions include the tasks that are added, removed, or have parameters modified by an adaptation).

2.2. Basic Language Constructs in PCL

The adaptation constructs in PCL are operations that modify program behavior by conceptually modifying the program's STG. Figure **??** presents a subset of PCL. We briefly introduce the key constructs in PCL in order to explain the coordination strategy and its implementation in PCL completely.

The user places a `pcl_AdaptSite` directive in the code to name a location in the STG where tasks can be inserted and removed. The name of the adapt site must be unique and is used by those PCL directives which insert or delete tasks. A task is identified either by a function name (the task is the entire function) or by a label identifying a structured program scope. (The current version of PCL only supports complete functions as tasks.) No other information about the task graph is needed.

```
pcl_AdaptSite(<name>, <arg-list>);
pcl_AddTask(<adaptsite>, <task>, <P>  [, <cid>]);
pcl_RemoveTask(<adaptsite>, <task>, <P> [, <cid>]);
pcl_ReplaceTask(<adaptsite>, <old-task>, <new-task>,
      <P> [, <cid>]);
pcl_ChangeParameter(<name>, <new-value>, <P>);
pcl_AdaptMethod(<function>);
pcl_AsyncCall(<adapt-method>, <arg-list>);
```

Figure 1. Key Operations of PCL

Task graph operations include `pcl_AddTask`, `pcl_RemoveTask`, and `pcl_ReplaceTask` to insert, delete, or replace a task at an adapt site in the STG. Each operation takes arguments that name the adapt site and the task(s) needed by the operation. In addition, each specifies the process where the operation should be executed. The `pcl_ChangeParameter` directive allows the user to change the value of a *control parameter* on a target process P. A control parameter is otherwise an ordinary variable, so it can be used in program control flow to change program behavior without modifying the task graph.

In [**?**] we show that executing the adapt logic asynchronously significantly decreases the overhead of using PCL. The user writes all adapt logic in one or more separate functions and uses the `pcl_AdaptMethod` directive to indicate the function contains adapt logic. Finally, the user invokes the adapt method asynchronously with the `pcl_AsyncCall` directive, supplying an argument list to be passed to the adapt method.

3. Specifying Coordination Requirements in PCL

Many distributed applications use coordination techniques to coordinate and protect access to critical resources or data. This synchronization between processes may be implied by message send and receive operations, or may be done explicitly using distributed locks and barriers. Writing adaptive logic for such applications requires careful consideration of the implicit and explicit synchronization to avoid introducing errors.

3.1. A Motivating Example

Consider a simple streaming-video client and server application that supports multiple video formats. A small portion of the task graph of this application is shown in Figure **??**. Either the server or the client can monitor network

conditions and change video formats to maintain some desired QoS metric (e.g., frame rate) as network conditions change. The encode/send and recv/decode operations depend on the current frame format which can be changed by replacing these two tasks. The adapt sites $AS1$ and $AS2$ mark the location of tasks that will be replaced by the adaptation. Initially, there is a task at each adapt site to encode and decode frames in format "A" as indicated in the figure. A single format adaptation would involve two `pcl_ReplaceTask()` operations, one at each adapt site $AS1$ and $AS2$. A PCL code fragment for this is given and discussed in Section **??** and Figure **??**.

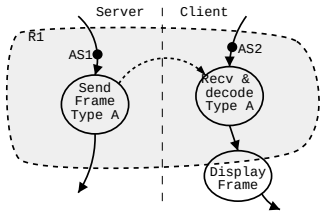

Figure 2. Partial STG of a streaming video application. The shaded area is region R1.

Switching the video format, regardless of whether it is performed with PCL, requires some coordination. The communication channel may contain several frames and partial frames in transit over the network. There are two requirements for synchronizing the adaptation. First, both client and server should switch formats prior to the same frame. Second, neither switch should take place while an encode-send task or receive-decode task is in execution.

A simple manual implementation might synchronize the two processes, wait for the channel to empty out, and then switch formats in both processes in synchrony. Note, however, that this synchronization is not strictly necessary; it is sufficient to ensure the *number of instances of the send* task when `pcl_ReplaceTask()` is performed at the server matches the *number of receive instances* when `pcl_ReplaceTask()` is performed at the client. The two counts might reach the same value at different points in time. The second ("internal") requirement above is met by performing the adaptation before entering the encode task or the corresponding decode task.

Similarly, Section **??** describes an adaptation of the number of ghostzones in a parallel PDE solver. This adaptation requires that two neighbors perform their respective parts of an adaptation before the same iteration of the time-step loop.

More generally, we find that adaptation operations between a set of processes typically require some condition that tracks the relative progress of the processes involved. A generalization of the examples described above would be that each participating process has reached some point in its execution relative to the initiating process and/or other participating processes. As a simple example, one might require that the number of instances of two tasks T_1 and T_2 executed by two different processes, might be related as $N_{T_1} \geq N_{T_2}$. In practice, we have found that the simple rule of the form $N_{T_1} = N_{T_2}$ (as in the two examples above) arises most commonly. This simple rule states that all participating processes are at the *same* logical point within a distributed computation.

Defining these coordination criteria more formally requires some notion of the set of tasks that are affected by the adaptation (such as the pair of encode-send and receive-decode tasks above). These tasks usually will be executed by multiple processes, each of which may have different entry and exit points to the tasks. The next section defines the notion of a *task graph region* that captures the execution behavior of a set of tasks. We use regions to enable the programmer to specify coordination policies that can be implemented by the compiler and runtime system.

3.2. Regions and Execution Progress

Def. (region): A *region* of the static task graph is a subgraph induced by an arbitrary set of tasks $\{T_1,\ldots,T_r\}$, with the sole restriction that a thread cannot be created or destroyed within the region (i.e., due to the execution of any of the tasks in the region).

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

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

A region may include one or more tasks, and may have one or more region-in and region-out edges. For example, the region labelled "R1" in Fig. **??** has two region-in edges (executed by different processes in this case) and similarly two region-out edges.

In order to express correctness criteria in terms of the progress of tasks in the distributed program, we define the following:

Def. (region instance): A thread t begins a new region instance when flow of control of the thread crosses a *region-in* edge. The region instance completes when flow of control crosses a *region-out* edge.

Def. (region count): For a region R and thread t, the region count $N_R(t)$ is the number of region instances that have been completed by thread t.

Def. (active region): A region R is active in thread t at a particular instant if thread t is executing some task in the region R at that instant. We also refer to this as an active instance of region R.

3.3. Scheduling Semantics

With our language support, the programmer can specify synchronization requirements such as those for the video-

server application via simple high-level declarations, *without* the complexity, overhead, and maintenance costs of programming the necessary synchronization behavior explicitly into the distributed application.

We begin by defining a few key terms. A *logical adaptation* (or simply, an "adaptation") is a set of adaptation operations that must be coordinated to perform some adaptive change (if using PCL, these are task graph operations listed in Figure **??**). We assume that the coordination of adaptation must be done relative to some region of the task graph, R; in every case, we have found R to be a small set of tasks affected directly by the adaptation. We define the *participating processes* as all the processes that must perform one of the operations in the adaptation. We assume that one process explicitly initiates an adaptation; referred to as the *originating process*. The originating process may also be a participating process.

Fundamentally, coordinating a distributed adaptation comes down to a scheduling decision of when to execute the adaptation on each participating process P_j. There are two main aspects which influence this decision: the *internal state* of P_j in terms of region R, and the *external state* of P_j relative to other participating processes $P_{i \neq j}$.

The internal constraint: The correctness of the target application may depend on whether the region is active or inactive on each process P_j. The user must decide how this relationship constrains the scheduling decision. The adaptation can be performed using one of four rules:

Choices for Internal Policy:

`Any`	:	*At any time*
`RegionIn`	:	*At a region-in edge*
`RegionOut`	:	*At a region-out edge*
`OutsideRegion`	:	*When the region is inactive*

One of these four policies will be specified as part of an adapt operation or group of operations, as described in Section **??**.

The external constraint: This constraint is used to specify requirements on the relative state of different participating processes. In the most general case, a scheduling policy can be a function that takes as input a set of N pairs specifying the participating processes and the region count on each: $In = \{(P_1, N_R(P_1)), \ldots, (P_N, N_R(P_N))\}$. The function returns a set of region count values indicating when the adaptation should be executed on each process: $Out = \{T_1, \ldots, T_N)\}$. This function is executed at runtime and can refer to internal program state.

In practice, as noted earlier, we have found only two policies to be needed, and we provide simple keywords that can be used to specify these:

Built-in Choices for External Policy:

`Any`	:	*at any time, independent of region counts.*
`EqualRegionCounters`	:	*the smallest common region count that can be reached on all processes.*

The algorithm implementing the EqualRegionCounters policy simply sets every T_i to be $\max_{1 \leq j \leq N} N_R(P_j)$.

3.4. Composite Operations (COPs)

A coordination policy specifies internal and external constraints for some *set* of adaptation operations. The set as a whole is treated as a single logical adaptation. In order to specify such a set and its coordination requirements, we introduce a language construct to PCL called a *Composite Operation* or COP.

A composite operation C consists of one or more component operations op_i. Our implementation uses PCL so each component operation can be any of the adapt operations shown in Figure **??**. Each component operation op_i has a process identifier P_i indicating the target participating process of the operation. There are no restrictions on the number of participating processes in a single composite operation.

The list of composite operations begins and ends with calls to two new PCL built-in functions:

```
int pcl_OpenComposite(void);
int pcl_CloseComposite(int CID,
  InternalPolicy IP, ExternalPolicy EP,
  char* RegionName).
```

The process that executes the composite operation is the originating process. As shown above, the coordination criteria are specified as arguments to the `pcl_CloseComposite()` function. InternalPolicy and ExternalPolicy are PCL-defined enumeration types that take the values listed in the previous section.

```
    int C; /* composite operation id */
    ...
1   C = pcl_OpenComposite();
2   pcl_ReplaceTask("AS1", "sendTypeA", "sendTypeB", Ps, C);
3   pcl_ReplaceTask("AS2", "recvTypeA", "recvTypeB", Pc, C);
4   if (pcl_CloseComposite(C, BeforeRegion,
        EqualRegionCounters, "RGN1"))
5       ...     //  handle error condition
```

Figure 3. Example of a Combined Operation with Coordination Criteria

4. Compiler and Runtime Support

The key to the coordination mechanisms introduced in section **??** is a novel synchronization algorithm which uses simple compiler support to perform adaptation operations on each participating process at the appropriate logical time. We discuss the compiler and runtime support below.

4.1. Compiler Support

We extended our PCL compiler to support composite operations and coordination. The PCL compiler is implemented using the LLVM compiler infrastructure [**?**]. PCL compiler operates as a source-to-C compiler, and can support any source language compiled to the LLVM intermediate representation. The PCL language directives are represented as ordinary function calls within the input program. The PCL compiler replaces each PCL directive with code (primarily sequences of runtime library invocations) to implement the directive.

The compiler inserts the code of Figure **??** along each RegionIn edge. This is the key step that requires a true compiler rather than a simple preprocessor, since this step requires correlating information from directives placed far apart in the code, namely, the `pcl_AdaptSite()`, `pcl_Region`, and `pcl_CloseComposite()` directives. The inserted code serves three purposes. First, lines 1-2 will block if a COP is in the process of being scheduled but is not yet committed. Second, line 3 increments the region count $N_{RGN1}(P)$. Finally, lines 4-5 will call a runtime function that will execute any adaptations that have been scheduled for the current value of $N_{RGN1}(P)$. These operations are described in more detail below. This compiler transformation is straight forward and requires no expensive analysis.

```
1:   while (RGN1_submitCounter > 0)
2:       RegionWait(RGN1);;
3:   ++RGN1_regionCounter;
4:   if (RGN1_pendingCounter > 0)
5:       CommitAnyPending();
```

Figure 4. Compiler generated code for a RegionIn edge of region $RGN1$

4.2. Runtime Support

The PCL runtime library uses CORBA [**?**] to communicate with the respective runtime libraries of other processes. The use of CORBA simplified the implementation, and the communication overhead of CORBA is not a significant concern (as our experimental results demonstrate). Each process is given a unique number P_i used for interprocess communication. The runtime library makes use of a thread to asynchronously execute adapt logic and an additional thread to run CORBA[1]. This prevents incoming and outgoing remote method invocations from interfering with the asynchronous execution of the adapt logic. We use RTL_i to refer to the runtime library thread associated with process P_i.

1 CORBA may internally create additional threads to handle incoming and outgoing remote method invocations.

The internal and external `Any` adaptation policies are easy to support. The runtime system at the originating process sends the composite operation to each participating process, which executes it immediately, regardless of its local state or that of other processes.

The remainder of this section first focuses on the `EqualRegionCounters` policy for external coordination in conjunction with the `RegionIn` policy. We then discuss how the algorithm changes for the other policies. All the algorithms below assume reliable message delivery, which is possible with CORBA.

4.2.1. Runtime Support on the Originator Suppose the originating site executes the code of Figure **??** to create the composite operation C on region $RGN1$. This distributed adaptation has two component operations: op_1 which replaces a task on the server process P_s and op_2 which replaces a task on the client process P_c. In this discussion we refer to the originating process as S and any participating process as P_i. (Note that S could be either P_s, P_c or some other process.)

The key insight in the coordination algorithm is recognizing that an adaptation which modifies multiple processes need not occur at the same wall clock time, but only at the specified logical time on each process. Even if the logical times are equal (e.g., for the `EqualRegionCounters` policy), they may occur out of phase due to load imbalance or network latency. Nevertheless, they can be scheduled correctly as long as they specify a future logical time of *all* involved processes. (This property must be preserved by the scheduling function, e.g., by the use of $\max_{1 \le j \le N} N_R(P_j)$ for `EqualRegionCounters`). The adaptation operation on each process can be executed locally at the scheduled time. The region count mechanism records the logical progression of time.

The pseudo-code which executes when a composite operation closes is shown in Figure **??**. The originating site performs three main steps shown in Figure **??**. Assume P_s is the originator. First, it sends C to all participating processes, i.e., to P_s and P_c.

After sending C, the originating process waits for a reply containing the region count from each participant. The

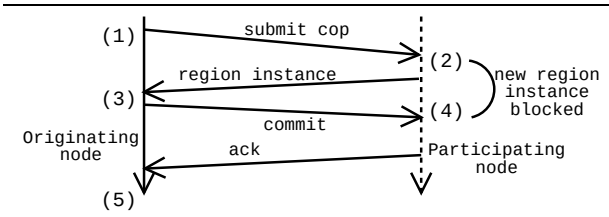

Figure 5. Messages sent between the originating site and each participating site

originating process builds a set of (process, region-count) pairs. In this example, the set would be:

$$\{(P_s, N_{RGN1}(P_s)), (P_c, N_{RGN1}(P_c))\}$$

When all replies have been received, S calls the scheduling algorithm passing it the region counts. The algorithm returns another set $\{T_s, T_c\}$ which indicates the region-count time when each participating process must execute the composite operation, where $T_i > N_{RGN1}(P_i)$.

Finally, the originating process must *commit* the adaptation on each participating process P_i by sending T_i to each process. Each participating process replies with an `ack` and after all `ack`s have been received the composite operation is *committed*, although the component operations may not have been executed yet on their target processes.

The key to the correctness of this algorithm is the assumption that the input region count values for each participating site will continue to be valid from the time they are first sent until the adaptation is committed (i.e., between points (2) and (4) in Figure **??**. This ensures that no participating site will begin a new region instance until after the composite operation has been committed, although it can execute other unrelated code. This is guaranteed by the runtime system on each participating process, as follows.

CloseCompositeOp(COP C)
  ∀ Participators $P_i \in C$
    invoke $SubmitCompositeOp(C)$ on P_i
  ∀ Participators $P_i \in C$
    recv $N_R(P_i)$ from P_i
    In = $In \cup (P_i, N_R(P_i))$
  Out = ScheduleAlgorithm(In)
  ∀ $T_i \in Out$
    invoke $ScheduleCompositeOp(C, T_i)$ on P_i
  ∀ Participators $P_i \in C$
    recv ACK from P_i

SubmitCompositeOp(COP C)
  R = $Region(C)$
  $++submitCounter_R$
  $submitList\,.\,add(C)$
  S = $originator(C)$
  send $N_R(P_{this})$ to S

ScheduleCompositeOp(COP C, int T_i)
  R = $Region(C)$
  $submitList_R\,.\,remove(C)$
  $pendingList_R\,.\,add(C)$
  $++pendingCounter_R$
  $-submitCounter_R$
  if $submitCounter_R == 0$
    $notify(R)$

Figure 6. Coordination Algorithm. For simplicity, the timeout logic for ACKs is omitted

4.2.2. Runtime Support on Participators The RTL on each participant is responsible for two things. First, it must guarantee the originator's assumption that no new region instance is entered from the time it reads the region instance counter until the adaptation is committed. Second it must execute the committed adaptation locally according to the schedule specified by the originator.

At point (2) in Figure **??** the participant P_i has received a composite operation C for region $RGN1$. The runtime system increments a *submit counter* which counts how many composite operations have been received but not yet committed, and puts C on the *submit list* which temporarily holds COPs until they are committed. P_i then reads the region count $N_{RGN1}(P_i)$ and sends it to the originating process S. The execution of the target application on the participant is allowed to continue unless it attempts to enter a new region instance.

P_i performs the steps shown in Figure **??** when it receives a commit message (point (4) in Figure **??**) for region count T_i from the originator. If the target application had been blocked because it attempted to enter a region it will be notified and allowed to continue. If it was not blocked it will continue without any knowledge of the new composite operation or any of the steps taken to commit it at time T_i.

After the PCL runtime on P_i schedules C, the adaptation will be executed at time T_i. The code of Figure **??** executes any pending composite adaptations for the current region count, at each Region-In edge. When an operation C is removed from the pending list it is executed locally.

This combination of distributed scheduling and local execution allows a complex distributed adaptation to be coordinated without the use of expensive barriers and distributed locks. The video server could even send all frames, scheduling adaptations on itself and the client and performing adaptations locally, and then exit before the client receives a single frame. When the client begins receiving frames it will execute each adaptation locally at the scheduled region count time and all frames will be received correctly.

4.2.3. Other policies The discussion above focused on the `EqualRegionCounters` and `RegionIn` policies. The `RegionOut` policy will execute the adaptation on a region-out edge of the region, which can be implemented in the same way as our `RegionIn` policy. The `Outside` policy indicates the adaptation should be performed when some region R is not active. Implementing this would require a mutex for each region, which is locked along every region-in edge and unlocked along every region-out edge on each process. The mutex would be acquired before executing the adaptation. This is more expensive than our current implementation of the `RegionIn` policy, but we have not found applications where it is strictly necessary.

A longer version of the current paper [**?**] discusses other key algorithmic issues, including why the algorithm is deadlock-free, why a centralized originating process is unlikely to be a significant bottleneck, and the effect of network latency.

5. Experience and Results

In this section we first discuss our experience with implementing two distributed adaptations using the coordination strategy described here, and then present experimental

results for the overhead and scalability of our coordination algorithm.

5.1. Applications

One of the challenges in the topic addressed here is that, to our knowledge, there are no realistic adaptive distributed applications in the public domain, even though several have been mentioned in the literature (e.g., [**?**, **?**, **?**, **?**]). To support our research, we wrote simple but representative versions of two adaptive codes that capture key adaptation behavior in real codes. One is a simple PDE solver with a distributed ghostzone adaptation similar to CactusG [**?**]. The second is a model video tracker in C with similar adaptations to those in the video tracking code. Due to space limitations we only briefly describe these two applications. A more complete description can be found in [**?**].

5.1.1. PDE solver We have implemented a simple PDE solver with adaptive ghostzones, similar to [**?**]. 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, but at the cost of some redundant computation. 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, and each process performs $G - 1$ rows of redundant computation at each boundary. The ghostzones adaptation changes the value of G at a particular boundary, to maintain computational efficiency under varying network conditions. Coordination is required so that any change to G occurs at the same logical time step on any pair of adjacent processes.

```
    extern void Adapt(...);
    pcl_AdaptMethod(Adapt);
    pcl_ControlParameter("NewGZSize");
    ...
    while (/* continue */) {
 1   pcl_Region("RGN1");
 2   pcl_AdaptSite("AS1", ... /* args */ );
 3   /* update ghostzones if necessary */
 4   /* compute matrix */
 5   if (nIterations \% ADAPT_PERIOD)
 6     pcl_AsyncCall(Adapt, ... /* args */ );
   }

    void Adapt(...) { ...
10   C = pcl_OpenComposite();
11   pcl_ChangeParameter("NewGZSize", gSize, C);
12   pcl_ChangeParameter("NewGZSize", gSize, rank-1, C);
13   pcl_AddTask("AS1", "ChangeAboveGhostsize", C);
14   pcl_AddTask("AS1", "ChangeBelowGhostsize", rank-1, C);
15   pcl_CloseComposite(C,
        BeforeRegion, EqualRegionCounters, "RGN1");
   }
```

Figure 7. PCL code fragment for PDE solver

```
C = pcl_OpenComposite();
if (/* add compression */) {
  pcl_AddTask("AS1", "CompressBlock", C);
  for (/* each client dest */)
    pcl_AddTask("AS2", "DecompressBlock", dest, C);
}
else if (/* remove compression */) {
  pcl_RemoveTask("AS1", "CompressBlock", C);
  for (/* each client dest */)
    pcl_RemoveTask("AS2", "DecompressBlock", dest, C);
}
pcl_CloseComposite(C, BeforeRegion,
    EqualRegionCounters, "RGN1");
```

Figure 8. PCL code fragment for video server

Adding the ghostzone adaptation only required writing code to update the solver data structures (which must be written regardless of how adpatation is implemented or coordinated), plus a small amount of PCL code, shown in Figure **??**. Briefly, the adapt method in this PCL code examines performance metrics (not shown) and may issue the composite adaptation operation shown. Each process monitors performance and adapts the ghostzone size at only one of its two boundaries to avoid competing adaptations. The composite operation changes the control parameter `NewGZSize` on both participating processes and then inserts a task which will read the new value of the control parameter and change the data structures of the PDE solver.

In the absence of our language support for coordination the user would have to use primitive distributed coordination methods such as a barrier. Such a barrier would need to be passed at *every* timestep rather than only those timesteps when an adaptation is scheduled.

5.1.2. Distributed Video Tracking Model In a distributed video tracking application [**?**], a video server sends a stream of individual image files to each client. The files may be raw or compressed depending on the CPU and network load at the server to each client. This was the example we used in the language and algorithm sections **??** and **??** and a partial taskgraph of this application is shown in Figure **??**.

Each client must be informed when the server switches compression. Coordination is required to prevent the client from incorrectly interpreting the data. The distributed coordination support of PCL allows the adaptation to happen without adding any additional meta-data to the data stream to notify the client, or writing explicit messages to do so. Rather, the runtime system handles all the necessary coordination to prepare each client to receive the correct frame type when the server adapts.

The code fragment in Figure **??** is part of the server's adapt method which decides whether it should start sending compressed or raw data. The code fragment will add a component operation to C to add or remove the appropriate task for every client destination.

5.2. Experiments

Two aspects of our algorithm are important to evaluate: the overhead introduced by the coordination algorithm and the scalability of the algorithm as the number of participating processes grows.

To perform our experiments we ran two versions of each of the above applications. The first version had adaptation disabled. The second version used a fixed sequence of adaptations but these adaptations did not actually change program behavior, so that the only difference compared with the first version would be the overheads of the adaptation. (The PDE solver always used the same ghostzone size of 1 and the video server always used raw blocks). The full cost of coordinating and executing the composite operation is incurred in every case. All numbers are the average of three runs.

Each measurement of the PDE solver is for 500 iterations on a square matrix with 10,000 rows and columns partitioned among the processors. The adaptive version issued an adaptation every 20 iterations. The video server sent a "video" consisting of 12,000 frames of about 8K each, and sustained a rate of about 25 frames per second. The adaptive version issued an adaptation every 100 frames, or every 4 seconds. The frequency of adaptation in both applications is higher than is likely in production environments, further increasing the observed overhead.

All of our measurements were performed on a number of Sun Ultra-10 and Sun-Blade-1000 workstations connected by a 100Mbps LAN throughout our department.

5.2.1. Overhead The coordination algorithm introduces overhead into the distributed application in two different ways. First, the adapt method is invoked asynchronously in both programs by a separate thread. This thread and the application thread compete for CPU time. Second, each participating process *may* be blocked by the runtime system for a short time before the adaptation is committed.

Figure **??** shows the execution times and percent overhead of each program. The overhead is lower than 1% in all cases. The overhead is very low for several reasons. First, the adapt logic on the originating process is executed asynchronously to the base application which allows the application to continue forward progress. Second, costly network messages are required *only* when an adaptation is needed and not also in the steady state. Finally, the region code of Figure **??** will only block the base application if a composite operation is being scheduled *and* the application attempts to enter the region. If the application is not currently executing this part of the code then the application will not be blocked during scheduling.

5.2.2. Scalability We used the video server program to evaluate scalability since its adaptation involves N processes. We varied N and measured the execution time of the coordination algorithm, using the `BeforeRegion`, `EqualRegionCounters` policy. (Using a different scheduling policy discussed in **??** will not significantly change the scalability results because each algorithm executes on a single processor and compares only N numbers. The bulk of the execution time of the algorithm results from network messages.)

		Number of Nodes		
		7	14	21
PDE solver	no adaptations	937.73	387.16	262.24
	adaptive	939.56	387.96	264.18
	overhead	0.20%	0.21%	0.74%
	adapt. period	37.6	15.5	10.5
File Copy	no adaptations	481.26	481.64	482.32
	adaptive	482.10	483.38	482.73
	overhead	0.18%	0.36%	0.07%

Figure 9. Overhead measurements (seconds)

Figure **??** shows the average time to execute the coordination algorithm with N participating processes. The average execution time of the coordination algorithm for the PDE solver was 0.093 seconds. As the figure shows, the coordination algorithm appears to scale very well although there is a somewhat high fixed cost. This indicates the fixed cost is almost all network latency, not CPU overhead.

number of clients (+1 for number of participating sites)								
6	13	20	30	40	50	60	70	80
0.11	0.12	0.12	0.15	0.14	0.13	0.10	0.12	0.12

Figure 10. Scalability measurements, all times in seconds (bottom row)

Overall, the experimental results show that using our algorithm for coordinating distributed adaptations is both efficient and scalable.

6. Related Work

There have been a number of programming languages aimed at simplifying different aspects of distributed computing, e.g., Lynx [**?**], Emerald [**?**] and SR [**?**]. Some languages such as Strand [**?**] expose a mix of standard synchronization and shared memory mechanisms as language constructs. There have also been specific language mechanisms such as Java RMI and standard middleware systems such as CORBA [**?**], DCOM [**?**] and Java Beans. All these systems primarily focus on high-level mechanisms for remote communication, resource management, and scheduling. We are not aware of a programming language that specifically focuses on enabling runtime adaptation in either sequential or distributed programs.

Researchers at BBN Systems have developed powerful middleware for reliable and adaptive distributed systems. Their platforms include distributed coordination and synchronization techniques for resource management and sharing in the presence of QoS constraints [**?**, **?**] and for coordinating and scheduling real time systems [**?**]. To our knowledge, however, none of the work proposes specific support for coordinating distributed adaptations from high-level specifications, which has been the focus of this paper. Our algorithm could be useful not just at the application level but also for implementing coordination requirements within their middleware.

Chen et al. [**?**] describe a system for runtime adaptation where coordination is based on identifying message flows through an adaptive component. An adaptation replaces an old algorithm module with a new one and occurs in three steps: Preparation, Outgoing Switchover, and Incoming Switchover. A global barrier is needed after the first step to ensure that all adaptive components are ready to receive new messages. The PCL coordination approach requires no such barrier because adaptations are scheduled for a future agreed-upon time for all adaptive components.

Workflakes [**?**] is an externalized dynamic adaptation platform that works by superimposing a performance feedback loop onto an existing distributed system. Workflakes performs adaptations externally and at the process level, and the system does not provide specific features for adding performance monitoring or adaptation "effector" mechanisms (these must already exist in the target application). The system provides sophisticated mechanisms programmers can use to control the operations of the effector mechanisms, but does not automate this process — it must be specified manually by the programmer.

A number of other middleware and runtime systems also support dynamic adaptation [**?**, **?**], many for specific goals such as Quality-of-Service in distributed network applications [**?**, **?**, **?**], fault-tolerant distributed systems [**?**], mobile applications [**?**, **?**], and distributed scientific applications [**?**, **?**, **?**]. Because these systems share some of the goals of PCL (in terms of simplifying adaptive distributed applications), and they are described and compared with PCL in [**?**]. To our knowledge, these systems do not automate the task of coordinating adaptation operations on multiple processes; this coordination must be explicitly managed by the programmer.

There are sophisticated distributed algorithms for a wide range of distributed coordination problems, including mutual exclusion, logical time and logical clocks, global snapshots, distributed consensus, and concurrency control. The concepts of logical time and global snapshots on the surface appear related to our language mechanisms for specifying a correct logical state in which adaptation should occur. In fact, however, all of those concepts rely on communication events in a distributed computation to identify the progress and ordering of events across processes. In contrast, regions and region counts are purely local properties within each process or thread, and are not defined in terms of communication operations. Our runtime algorithm for scheduling adaptation operations is closer to some of the algorithms mentioned above. Our algorithm is relatively simple and uses centralized decision making at the initiating site because, in practice, we expect each logical adaptation to involve a relatively small number of processes. Nevertheless, some of the techniques in existing algorithms (such as those for distributed consensus or leader election) could be used to implement more sophisticated synchronization strategies, particularly in the presence of unreliable communication or node or process failures.

7. Conclusion

In this paper we presented language mechanisms, compiler support, and a novel runtime algorithm for coordinating adaptations in distributed applications. The language directives allow programmers to specify coordination requirements in simple high-level terms, without explicit communication or synchronization. Our runtime algorithm schedules the adaptation operations to be executed locally and asynchronously (using logical times based on computational progress rather than application messages), relying on simple compiler support to track computational progress.

The main limitation of our approach is that it focuses on coordination rules based on the relative flow of execution of different processes. In future work, it would be interesting to extend the approach to other potential coordination requirements. Within these limitations, however, our approach has several advantages. First the coordination requirements can be expressed by a single process, independent of the actual processes where the actual adaptations occur. Second, the expense of coordination is incurred only when needed, rather than continuously as would be the case with barriers or distributed locks. Furthermore when coordination is required for an adaptation the overhead is minimal, and the algorithm is very scalable. Third, the runtime library implements all remote communication to schedule and execute the adaptation, sparing the user from the difficulties of writing another level of communication specifically for coordinating adaptations.