Shared Memory Programming in a Distributed System *

Mustaque Ahamad
Muthusamy Chelliah
Partha Dasgupta
Richard J. LeBlanc
Mark Pearson

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280 USA
Phone: (404) 894-2593

mustaq@cc.gatech.edu

Abstract

Differing remote and local state access mechanisms in distributed computing environments make programming for concurrent execution difficult. This paper presents a novel programming paradigm that simplifies development tasks by allowing programmers to create distributed applications without having to be aware of their physical distribution, or runtime degrees of concurrency. This paradigm, used to program distributed applications for the CLOUDS distributed operating system, models these applications like centralized programs where processes communicate and synchronize using shared memory. Distribution is achieved automatically at execution time.

*This work funded by NSF grants CCR-8619886 and CCR-8806358.

1 Introduction

An application can exploit the multiple, loosely-coupled processing elements in a distributed computing environment by executing a computation in parallel. However, programming parallel computations in a distributed environment is complex, because processes in distributed computations must use separate mechanisms to access local and remote states. For example, in a message passing system, a process can read and write its local state, but must communicate using messages when it needs to access the state information of another process. In an object-oriented system, it is assumed that an object resides at a single computation node and that remote procedure calls (RPC) are used to access the states encapsulated by remote objects. Although RPC mechanisms extend the procedure call abstraction to an entire distributed system, it is necessary to structure distributed applications as sets of objects that reside at several computation nodes.

In many systems, each individual component of a distributed computation must be explicitly programmed, and then coordinated in a fashion that leads to its correct execution. In message passing systems, the developer programs separate processes that collectively execute the application, while in object systems, distinct objects must be programmed, and coordinated. By assigning processes or objects to different computation nodes in a distributed system, it is possible to achieve concurrent execution of the application. Often however, complicated protocols have to be defined to achieve proper coordination of components that execute at different compute nodes in order to ensure correct execution of the computation.

This paper presents a novel programming paradigm that simplifies the task of distributed applications development. Some of the advantages offered by the paradigm are:

- An application programmer does not have to be aware of distribution.
- A distributed program can be written as a centralized program where processes communicate and synchronize using shared memory.
- The degree of distribution or concurrency does not have to be decided at the time the program is written.

This programming paradigm is suitable for object-based distributed systems that support on demand object mobility using *distributed shared memory* (DSM) [LH89, RAK89], such as the Clouds distributed operating system. Clouds has several advantages for distributed programming. First, it treats computation and storage orthogonally. Therefore, the node where an object is stored has no bearing on what node serves as the computation site for that object. This approach is necessary in environments where storage is provided by data servers and execution is done at compute servers. Second, CLOUDS uses a single level persistent store for all objects. In this paper, we show how these features can be effectively used to provide a novel, easy to use, distributed programming paradigm.

Section 2 presents a brief overview of the CLOUDS programming model and environment. We then demonstrate the power of the proposed programming paradigm by presenting several types of distributed applications. These applications include a linear equation solver (Section 3.1), the sieve of Erastothenes (Section 3.2), and a distributed dictionary (Section 3.3). Related work is discussed in Section 4.

2 CLOUDS Programming Environment

CLOUDS is a distributed operating system that integrates a set of loosely coupled machines into a conceptually centralized system. The system is composed of *compute servers*, *data servers* and *user workstations*. Compute servers are the system's computational engines, while data servers function as repositories for long-lived, persistent data. User workstations provide application development tools and serve as the user interface engines to the compute and data servers which execute the applications. *Compute server* and *data server* are logical designations; the same machine may function as both. CLOUDS is a native operating system, that runs on top of a native kernel called *Ra*[1]. It currently runs on Sun-3/50 and Sun-3/60 computers and cooperates with Sun SparcStations running UNIX[2] which serve as the user interface engines.

CLOUDS is a general purpose operating system. It is intended to support all types of languages and applications, distributed or not. All applications can view the system as a monolith, but distributed applications may choose to view the system as loosely coupled compute and data servers. Each compute facility in CLOUDS has access to all resources in the system.

[1] After the Egyptian Sun God.

[2] UNIX is a trademark of AT&T Bell Laboratories.

2.1 The Object-Thread Model

The system structure in Clouds is based on an object-thread model. The object-thread model is similar to a *object-oriented* programming model which structures a software system as a set of *objects*. In the object-oriented model, each object is an instance of an abstract data type consisting of data and operations that can be used to access the data. The operations are called *methods*. An object is an instance of a *class* which defines the type of the object. A class may have any number of instances, but an instance is derived from exactly one class. Objects respond to *messages*. Sending a message to an object causes the object to execute a method. The execution of a method reads and/or updates data stored in the object and may cause messages to be sent to other objects. Upon completion, the method sends a reply to the sender of the message.

Clouds has a similar structure, implemented at the operating system level. Clouds objects are large-grained encapsulations of code and data which model distinct, persistent virtual address spaces. The contents of an object is an instance of a class; a class is a compiled program module. Unlike virtual address spaces in conventional operating systems, the contents of a Clouds object are long-lived. That is, a Clouds object exists forever and survives system crashes and shutdowns (like a file) until explicitly deleted.

Activity in the Clouds system is provided by threads. A thread is a logical path of execution that executes code in objects, traversing objects as it executes. Unlike processes in a conventional operating system, a Clouds thread is not bound to a single address space, since each object represents a distinct address space. A thread is created by an interactive user, or under program control. When a thread executes an entry point in an object, it accesses the persistent data stored in it. In addition, the code in the object may *invoke* operations in other objects. In such an event, the thread temporarily leaves the calling object, enters the called object and commences execution there. The thread returns to the calling object after the execution in the called object completes. A thread can carry information across objects through parameters of invoked operations. These parameters are strictly data; they may not be addresses, since an address reference in one object is meaningless in the context of another object. In addition, object invocations can be nested or recursive. A thread terminates after it completes all the invocations it was created to carry out.

Any number of threads may execute concurrently in a Clouds object, and one thread may execute code in several objects during its lifetime. Thus, Clouds treats computation and storage orthogonally by using objects as the abstraction of storage and threads as the abstraction of computation. Also note that Clouds objects are "heavyweight" and are better suited for storage and execution of large-grained data and programs because of the overhead associated with invocation and storage of objects.

2.2 Distributed Computations

A novel feature of CLOUDS that makes distributed programming simpler is the automatic migration of objects via CLOUDS object invocation and distributed shared memory (DSM) mechanisms. Suppose a thread is created on compute server A to invoke object O_1. As the thread executes, the code and data of O_1 that is accessed by the thread is demand paged to A from the data servers where O_1 currently resides (possibly over the network). If the thread executing in O_1 generates an invocation to object O_2, the system may choose to execute the invocation on either A itself or on a different compute server B. In the former case, if the required pages of object O_2 are at other nodes, they have to be brought to node A. In the latter case, the thread sends an invocation request to node B, which executes the invoked operation of object O_2 by demand paging O_2 from data servers to B and, upon completion of the invocation, results are returned to node A. The execution in O_1 resumes after the results are received. This invocation mechanism is called *synchronous*, because execution in the current environment of the invoking thread waits until the invoked operation completes and results are returned. An invocation can also be made *asynchronously* which is discussed later. Although the CLOUDS object invocation mechanism is similar to remote procedure call (RPC) mechanisms found in other distributed systems, it is more general because B does not have to be the node where O_2 currently resides.

Object migration is also possible in other systems [BHJ+87], but it is required that the target object be at the node where the threads in it are executing. As a result, a computation executing in a single object cannot be distributed. Since CLOUDS treats storage and execution orthogonally, it allows threads at many nodes to execute in the same object. The object is shared between these nodes, leading to an environment in which all objects are available at all compute servers. The accessed parts of an object are fetched by the nodes as they are needed by the threads executing at them, which results in a situation similar to multiprocessor systems that employ caching to reduce memory access times. In CLOUDS , a DSM coherence protocol is used to ensure that the data in an object is seen by concurrent threads in a consistent fashion even if they are executing on different compute servers. Several DSM protocols exist [LH89, RAK89, FP89]. We have implemented a protocol that is similar to the one described in [LH89] which provides read–only and read–write memory segments. The code defined by objects is stored in read–only segments which can be copied by nodes freely.

Since several threads can simultaneously execute in an object, it is necessary to coordinate access to the object data. This is handled by CLOUDS object programmers using CLOUDS system synchronization mechanisms such as semaphores and locks. Through these mechanisms, threads may synchronize their actions regardless of where they execute.

In addition to the synchronous invocation mechanism described above, CLOUDS also allows threads to be created while an invocation is being executed. Since the basic unit of work is an operation, CLOUDS allows a new thread to be created to execute an operation. This is achieved by an *asynchronous* invocation mechanism which is similar to asynchronous remote procedure calls [LS88]. When thread t_1 executing in object O_1 asynchronously invokes an operation of object O_2, a new thread t_2 is created which executes the invoked operation. Asynchronous invocation returns a variable p of type $invocation_handle$ which can be used as a place holder for object return value of any type. It is similar to *promises* of Argus[LS88]. It is nothing but a pointer to specific region within the object space to hold the return value and the results of asynchronous invocation can be obtained by executing a *claim* operation on it. Thread t_1 can continue with execution in object O_1 and possibly make other invocations while the operation invoked in O_2 is executed by t_2. At a later time, thread t_1 can wait for the completion of the operation being executed by t_2 by executing a *claim* call on p. When *claim(p)* returns, t_2 has completed and the results returned by the invoked operation are available to t_1 through p. Thread t_1 can also communicate and synchronize with thread t_2 using semaphores or locks to find out about its progress. It is possible to invoke several operations asynchronously and a *claim* can be made for results of any one of these outstanding invocations.

3 Programming Examples

In the following subsections, we present examples to illustrate the shared memory programming paradigm that is used in the CLOUDS system for developing distributed applications. Each of the application that we present is programmed as a *single object*. Distribution is achieved by executing the application by a set of threads that execute at different nodes and share the code and data of the object.

Since the application is programmed as a single object, and all threads in an object have access to the object data, there is no explicit expression of distribution in the programming of the application. The first object invocation that starts the application is done by the user from a Clouds user interface on a user workstation. Thereafter, if the executing thread performs asynchronous invocations, they result in the creation of threads at remote nodes. The system automatically chooses a node to execute a newly created thread (the system tries to use idle nodes for executing new threads) and hence the programmer need not specify the nodes where the asynchronous invocations must be executed. However, there is a feature provided by the system that allows users to choose a compute server for a thread created to execute an asynchronously invoked operation. Since the object data is shared by all threads, the programming of an application must deal with process coordination problems that arise in a shared memory environment.

The first two examples discussed in this section will illustrate how synchronization similar to barriers and synchronous message passing can be achieved using the *claim* mechanism and semaphores. The third application illustrates how persistence and distribution can be effectively used to reduce communication in object-based distributed systems. The underlying Clouds operating system supports persistence and ensures that when an object is accessed by two different programs (or) multiple threads belonging to the same program, the changes being made by any thread are persistent and visible to the other threads.

In the first problem, we consider a parallel iterative algorithm for solving *linear equations* where processes must synchronize at the end of each iteration. All the data that is generated in the current iteration is required before the next iteration can be started. The next problem we consider is generating all prime numbers smaller than a given upper bound. The *sieve of erastothenes* algorithm was used in [Hoa78] to illustrate how processes can cooperate in a message-based environment to solve this problem. In our example, we program this algorithm using a single Clouds object. However, by invoking multiple threads which execute within the Clouds object at different computation nodes, we obtain execution time behavior that is similar to a message passing programming model where processes are programmed explicitly and communicate via messages. The last example is an implementation of a *distributed directory* that supports insert, delete and lookup operations. The directory example illustrates the use of object persistence and a potential reduction in communication costs due to data distribution and replication. These benefits are realized without the application programmer dealing with the issues of distribution and replication explicitly. The directory can be used to implement practical distributed applications such as name services.

The Clouds classes in this paper are programmed in CC++ [Ana90] which is an extension of C++ [Str86] (an extension of the *Eiffel* programming language can also be used to program

CLOUDS classes) . CC++ provides features that allow CLOUDS class templates to be defined which are used to instantiate CLOUDS object instances. CC++ also provides language support for persistence, synchronous and asynchronous invocations, and synchronization. Although the code in the following sections omits several details to simplify presentation, these applications have been programmed and tested in CLOUDS .

3.1 Linear Solvers

We consider the parellel version of the Gauss-Siedel algorithm for solving linear equations. The problem is to solve linear equations of the form $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is a $n \times n$ matrix of real numbers, $\mathbf{B}$ is a vector of n real numbers, and $\mathbf{x}$ is a vector of size n of the unknowns to be solved. In this algorithm, new values of the elements of $\mathbf{x}$ are computed iteratively until the solution converges (sum of the differences of successively computed values becomes less than some acceptable tolerance value). Let us denote by x_i^t the value of x_i computed in the t^{th} iteration. Also, let $a_{i,j}$ be the element in row i and column j of matrix $\mathbf{A}$, b_i be the i^{th} element of vector $\mathbf{b}$. The values of $\mathbf{x}$ in the $i+1^{th}$ iteration are computed using the following relationship:

$$x_i^{t+1} = \frac{b_i - \sum_{k=1}^{i-1} a_{i,k} x_k^t - \sum_{k=i+1}^{n} a_{i,k} x_k^t}{a_{i,i}} \tag{1}$$

We have programmed a parallelized version of the Gauss-Siedel algorithm in the CLOUDS environment to solve linear equations. We use a synchronous iterative algorithm in which all values in an iteration have to be computed before the next iteration can be started. For simplicity, we will assume that n threads are created so that each thread computes a single element of the vector $\mathbf{x}$. To obtain better performance, each thread may compute several elements. Our example program can be easily modified to achieve this.

We program this application within a single CLOUDS object. A thread created for computing the value of x_i in the k^{th} iteration must wait until all values from the $k-1^{th}$ iteration have become available. Furthermore, a new value computed for x_i cannot be written until the previous value has been read by all other threads. Although, one can use semaphores to enforce the necessary synchronization, in this example we use the *claim* mechanism of asynchronous invocations. In each iteration, n threads are created using the asynchronous invocation mechanism and they all must complete before threads are created to execute the next iteration (when necessary). The code that implements the iterative algorithm is defined by class `linear` which is given in figure 1. In the global data declaration, an array of variables of type `invocation_handle` is declared. When an asynchronous invocation is made, a variable `handle` is returned. `handle` is used in the `claim` call to specify which invoked operation completion and results are requested. The `handle` array temporarily stores the newly computed values in an interation until all values have been computed when x_i can be updated.

There are three entry points defined by the `linear` class. The `init` operation initializes the contents of the matrix **A** and vectors **b** and **x**. The `solve` operation iteratively makes asynchronous invocations to the `compute` operation which finds a new value of x_i based on the current values of the elements of vector **x**. The asynchronous invocations of `compute` result in the creation of n threads that execute concurrently. Once these invoked operations complete, the new values would have been computed and stored in the `handle` array. In `solve`, the *claim* call is executed for each invocation of `compute` to get the new values when they become available. The convergence test is then performed and another iteration is executed when necessary.

The class `linear` is instantiated to create a CLOUDS object that can be used to solve a set of linear equations by calling the operations defined by it. The `Init` operation is called to initialize the vectors **b** and **x** and the matrix **A**. This operation is executed by a thread created by the user from a user workstation. We call this the *parent* thread. The parent thread can then begin execution of `solve`. As a result, n threads get created to compute each element of the vector **x** (it can easily be modified so that each thread computes several values). These new threads are created as a result of asynchronous invocations of the `compute` operation and may execute on different nodes. These nodes demand page the code and data required by the threads. Thus, a distributed computation results even though the programmer does not deal with distribution explicitly in the object code. The parent thread then waits for the threads created by it to complete. Although the threads may complete in an arbitrary order, they are claimed in a given order since further execution must wait until all threads complete. Notice that a thread executing the `compute` operation does not assign the newly computed value to the appropriate element of **x**, because it does not know when other threads read the value produced in the preceding iteration. Instead, the values are returned by the threads and the parent thread assigns them to **x**. If another iteration is necessary, the parent thread creates n new threads to execute the iteration. Since the threads that execute the next iteration are created after the values in `handle` are copied to **x**, it is easy to see that the values used in $i+1^{th}$ iteration are the ones computed in iteration i, as required by the algorithm.

There are several interesting aspects of the `linear` object and the resulting distributed computation. First, instead of achieving synchronization through semaphores, thread creation and termination are used. As will be seen in the `sieve` object in figure 2, it is possible to start a single

```
clouds class linear
   invocation_handle handle[n];
   float A[n][n], b[n], x[n]; /* Ax = B */

entry init(float mat, vec, init_x) /* entire arrays passed as values */
{
   initialize A, b and x to mat, vec and init_x respectively;
} /* end of init */

entry int compute(int i); /* compute new value of i th element */
{
   int temp;
```

$$\texttt{temp} = \frac{\texttt{b[i]} - \sum_{\texttt{k=1}}^{\texttt{i-1}} \texttt{A[i,k]x[k]} - \sum_{\texttt{k=i+1}}^{\texttt{n}} \texttt{A[i,k]x[k]}}{\texttt{A[i,i]}};$$

```
   return(temp);
} /* end of compute */

entry solve(float tolerance);
{
   int j;
   float error;

   do
   {
   for (j = 1; j <= n; j++) /* create n threads, one for each element */
            handle[j] = linear!compute(j); /* asynchronous invoke */
      /* wait for all threads to complete */
      for(j = 1; j <= n; j++)
         claim(handle[j]);
      /* check convergence and assign newly computed values to x */
      error = 0;
      for(j = 1; j <= n; j++)
      {
         error = error + abs (*handle[j] - x[j]);
         x[j] = *handle[j];
      }
   }
   while(error >= tolerance);
   output x;
} /* end of solve */

end_class /* linear */
```

Figure 1: Linear Solver Object

thread to compute values of a given element of **x** in all iterations, but that would require it to synchronize with other threads using semaphores or other similar mechanisms such as eventcounts used in the linear-solver application presented in [LH89]. Since semaphore opreations, thread creation, and thread return require internode communication when these operations are executed at different nodes, we claim that the performance of the application will be similar with either approach because communication between nodes dominates the cost of these primitives. However, the code in `linear` is more natural because the parent thread makes a set of calls, waits for them to return, and then starts the next iteration. A sequential program implementing the algorithm would also work in a similar fashion except that the calls to compute the elements will be made *serially*.

Distributed execution of the linear solver application is possible without explicitly dealing with distribution because of the underlying distributed shared memory coherence protocols. Assume that all n threads execute on different compute servers. We assume a DSM coherence protocol that allows these servers to make copies of data for reading. When a write is done, the copies are invalidated. In this example, the contents of the matrix **A** and **b** will be brought to each compute server when the first iteration is executed at the server. These copies are still maintained at the servers, because object memory in Clouds is persistent and exists independent of threads that operate on it. Thus, for future invocations, the contents of **A** and **b** need not be sent over the network. When the elements of `handle` are copied to **x**, it is necessary to invalidate existing copies of **x**. The invalidation results in messages. For example, since the value of x_i is fetched by nodes that execute the `compute` operation, the invalidation message is sent to all compute servers that participated in the execution of an iteration. In the next iteration, the values of **x** are fetched again from where they exist. It is possible to set up the system so that all the needed data can be received in a single message (e.g., the node where the parent thread executes can be designated as the owner node of the data of **x** in the protocols described in [LH89, RAK89]). In a message passing system, the messages corresponding to the invocations and their returns are necessary, but additional messages are employed by the DSM implementation for invalidations. Since data must be exchanged in either approach, the additional messages used by DSM are control messages which take under 3 ms. in Clouds. Furthermore, this overhead can be reduced by requiring that each node compute many values of **x**. We believe that the simpler programming paradigm makes up for any loss of performance.

3.2 Sieve of Erastothenes

We want to program an object that implements the sieve of Erastothenes algorithm[Dij72] to generate a sequence of prime numbers less than a given upper bound. This example demonstrates that even when the algorithm is implemented using a single CLOUDS object, it is possible to create a complex distributed computation with extensive synchronization and communication between threads executing on different compute servers.

Implementations of the sieve algorithm where processes communicate using messages are presented in [Hoa78] and Joyce[Han87]. The paradigm consists of a root process P_0 that generates numbers from 2 till the upper bound, and a pipeline of concurrently executing filter processes to recognize and output the prime numbers in the sequence generated by the root process. Each process filters out numbers that are multiples of the prime number associated with it. The i^{th} process in the pipeline filters out numbers that are multiples of the i^{th} prime number (P_1 filters out multiples of 2). The communication structure is defined by each process having two unidirectinal channels. P_i receives numbers from P_{i-1} on its input channel and when it cannot determine if the received number is prime, it sends the number to P_{i+1} on the output channel. The programs given in [Hoa78] and [Han87] use synchronous message passing for inter–process communication. Thus, when a process wants to send (receive) a number, it waits until a matching receive (send) is executed by another process. Our programming of the sieve algorithm shown in figure 2 implements similar synchronization between threads by using semaphores.

The `sieve` class implements the algorithm sketched above. The communication between the threads is via the array `channel`. Each element i of this array has a `value` field that stores the i^{th} prime number p_i which is used by P_i to filter out p_i's multiples. The numbers are communicated through the `buf` field and two semaphores are used to simulate synchronous message passing like synchronization between the threads. The code to be executed by the threads is defined by four operations of the `sieve` object: `generate, put, get` and `filter`. The `generate` operation is executed when the computation starts. It generates numbers that are to be tested to see whether or not they are prime. It creates a thread which executes the code defined by the `filter` operation. The thread executing `filter` operation determines if a number *num* read by it is a multiple of the value associated with it. If *num* is a multiple, then it discards *num* and simply waits to read the next number. Otherwise, it writes *num* to the buffer from which the next thread can read it. When the next thread does not exist, it is created by executing an asynchronous invocation to the `filter` operation itself.

```
clouds class sieve
   /* declaration of data used for inter-thread communication */
   struct channel_record {
      Semaphore read_sem,write_sem;
      int buf,value }channel[MAX_NO_PRIMES];
   entry generate(max_prime)
   {
      initialize write_sem of channel elements to 1;
      initialize all other fields of the channel elements to 0;
      sieve!filter(1,2); /* asynchronous invocation; create first filter thread */
      for number = 3 to upper_bound do
         sieve.put(1,number); /* synchronous invocation; put number in channel for first filter */
      sieve.put(1,terminate_token); /* synchronous invocation; signal termination*/
   } /* end of generate */
   entry put(i, num)
   {
      channel[i].write_sem.P;
      channel[i].buf := num;
      channel[i].read_sem.V;
   } /* end of put */
   entry get(i, num)
   {
      channel[i].read_sem.P;
      num := channel[i].buf ;
      channel[i].write_sem.V;
   } /* end of get */
   entry filter(index,init_val) /* receive numbers from channel[index] */
      {
      int nextnum;

      print init_val;
      channel[index].value = init_val;
      nextnum = 0;
      while (nextnum != terminate_token)
      {
         sieve.get(index,nextnum);/* synchronous invocation */
         if (nextnum is not a multiple of channel[index].value) then
         {
            if (channel[index+1].value = 0) then /* create a thread */
               sieve!filter(index+1,next_num);/* asynchronous invocation */
            sieve.put(index+1,nextnum);/* synchronous invocation */
         }
      } /* end of while */
      if(channel[index+1].val) /* child thread exists */
         sieve.put(index+1,terminate_token);/* synchronous invocation */
   } /* end of filter */
   end_class /* sieve */
```


Figure 2: Sieve Object

The `get` and `put` operations allow threads to communicate with each other. The `get` operation implements the desired synchronization for receiving a number. Since communication is achieved using shared data, a thread executing `get` waits on a *read* semaphore for a writer. After it succeeds, it notifies the writer by a signaling a *write* semaphore. The `put` operation is similar and is meant for writing a number onto the buffer. A thread executing `put` waits on a *write* semaphore, and after writing releases the waiting reader by signaling *read* semaphore. A `put` operation starts the communication in this example. Hence a *write* semaphore is initialized to 1 and a *read* semaphore is initialized to 0. It can be seen that each element of the `channel` array serves as a bounded buffer between threads.

We create a computation that executes in the same way as the message passing solutions described earlier. Assume that the sieve algorithm is programmed in CSP [Hoa78] using N processes and the i^{th} process is denoted by P_i. P_i sends the numbers it cannot filter out to P_{i+1} by using blocking *output* commands. P_0 is the root process that generates the numbers. In the execution of the `sieve` object, let t_0 be the user thread that starts the computation from a user workstation. This thread executes the `generate` operation which creates a new thread by asynchronously invoking the `filter` operation. This thread, t_1, filters out multiples of 2 and executes in a simliar manner to CSP process P_1. Since a newly generated number is written by thread t_0 only when thread t_1 indicates that it is ready to read the number, synchronous communication similar to CSP message passing is achieved. When t_1 receives the number 3 which it cannot filter out, it creates another thread by invoking the `filter` operation itself. Thus, the threads in the pipeline are created dynamically. The interaction between threads t_i and t_{i+1} is the same as t_0 and t_1 and hence a pipeline of threads is set up and neighboring threads synchronize similar to CSP processes. Thus, we are able to create a concurrent computation even though the `sieve` class does not explicitly program the threads that are created at run–time. Although the `filter` operation is invoked asynchronously, no *claim* call is executed since it does not return any values.

We have implemented and tested an improved version of the code shown in figure 2. In the improved version, filter thread t_i, in addition to stopping composite numbers which are multiples of the i^{th} prime number p_i, also recognizes numbers in between p_i and p_i^2. It creates t_{i+1} only when a number greater than p_i^2 is received from thread t_{i-1}. This more efficient solution is possible due to the fact that if $p_i < num < p_i^2$ and num is not divisible by p_1 through p_i, num is also a prime number.

The `sieve` object is used to create a distributed computation by starting the threads created to execute `filter` operations at different compute servers. The degree of concurrency as defined by the number of threads executing at runtime is not specified at the time the application is programmed. The threads are created dynamically when needed and their number depends on the highest prime number to be generated.

Clouds underlying distributed shared memory mechanisms enable our shared memory programming paradigm. For example, the i^{th} element of the `channel` array is shared by threads t_{i-1} and t_i. When these two threads execute at different nodes, the DSM coherence protocol exchanges messages between the nodes to ensure that the contents of the buffer are consistent at both nodes. We claim that when the DSM protocol is designed well, the message cost of the underlying protocol for data transfer is no higher in this application compared to programming with explicit messages. The cost of synchronization will require that the nodes where threads execute exchange messages. Furthermore, the writer must transmit the value to the reader thread. By combining synchronization and data transfer [RAK89], we can allow two threads to communicate using the same number of messages that will be needed for implementing message passing between the threads. Thus, the simplicity and ease of programming does not mean loss of efficiency.

3.3 Distributed Directory

The purpose of a distributed directory is to maintain a set of *items* where an item is a *key* and an associated *value*. A user at any node can either lookup the value associated with a key or update the directory. The latter is done by inserting a new item, deleting an existing item, or modifying the value associated with a key. Unlike the sieve and linear solver examples, where the goal of distribution was to improve execution time, the directory problem poses a naturally distributed application because it is shared by users at all nodes. Since lookup and update requests can arrive concurrently, access to the directory must be synchronized. We require that the execution of directory operations be *serializable* [BDS87]. Thus, a lookup operation should return a value only if the value is inserted, but not deleted by the operations serialized before the lookup operation.

Although the directory can be implemented by a single central server to which requests can be sent, a distributed implementation that allows each node to participate in the directory maintenance is preferred. Two compelling reasons for a distributed implementation are load sharing and fault tolerance. A distributed implementation requires a server to be present at each node. This set of servers collectively maintain the directory. However, the coordination between the servers could be complex. If each server stores all items, it is necessary that a server communicates with all others to execute update operations. When the items are not replicated, a lookup operation may

```
clouds class directory
   struct hash_table_record {
      rwlock hash_lock;
      struct bucket_record {
         char key[MAX_KEY_LENGTH];
         int value;
      } bucket_entry[NUM_BUCKET_ENTRIES];
   } hash_table_entry[NUM_HASHTBL_ENTRIES];

entry int lookup(char *key_to_look)
{
   int i, return_value;
   i = hash_function(key_to_look);
   read_lock(hash_table_entry[i].hash_lock)
   for (i = 0; i < NUM_BUCKET_ENTRIES; i++)
      if(hash_table_entry[i].key == key_to_look)
         return_value = hash_table_entry[i].value);
   if (i == NUM_BUCKET_ENTRIES) /* key not found */
      return_value = FAILURE;
   unlock(hash_table_entry[i].hash_lock);
   return(return_value);
} /* end of lookup */
```

Figure 3: Directory Object *Data Definition* and *Lookup* Method

be complex since the server where it arrives must communicate with others to find if the desired item is in the directory.

In our distributed dictionary implementation, we demonstrate that distribution and persistence can be exploited while still using our shared memory paradigm to simplify its programming. Similar to earlier examples, the directory is also programmed as a single class called `directory`. It implements the `lookup, insert` and `delete` operations. The code for this class is given in figures 3 and 4. A `modify` operation can be programmed similarly and is not shown. To allow a high degree of concurrency, we require that the synchronization implemented by the `directory` class allows concurrent execution of non–conflicting operations. Thus, `lookup` operations can be executed concurrently. We also allow update operations (`insert` and `delete`) to execute concurrently when they do not access common areas of memory in which the directory items are stored.

We implement the directory using a hash table. Each bucket in the hash table defines storage for a number of directory items. It also defines a `lock` variable which can be acquired in a *shared* or *exclusive* mode. Thus, a single lock controls access of directory items that all hash to the same bucket. Although storage can be allocated dynamically in Clouds , in `directory`, it is declared statically for simplicity. The code of the three operations of `directory` is straightforward. Using a hash function, a bucket is located and an appropriate lock is acquired before updating or reading the storage controlled by the lock associated with the bucket. The lock is released after access to the desired data is completed and the operation then completes.

The execution of operations of an object created from `directory` is quite different from the two examples considered earlier. In both the linear solver and the sieve, a user thread starts the computation which results in the creation of more threads. These threads execute concurrently possibly at many nodes. The computation then completes (e.g., when the last prime number is generated or the solution converges). On the other hand, the `directory` object defines persistent data and operations which are used to manipulate the data. These operations are invoked by threads of other computations which share the directory data and may execute at non–overlapping times. For example, an item can be inserted by a thread which then terminates. This item can be looked up by a thread that executes at a later time. Since data is persistent and exists independent of threads and computations, it is not necessary that threads that use it exist at all times or it be written out (and read) using I/O facilities.

The operations of `directory` can be executed at many nodes. The threads demand page the code and data to the nodes where they execute. Thus, the directory data is distributed. The data fetched at a node resides there until it is deallocated or is overwritten. Thus, data may exist at a node even when no thread is executing at it at a given time. Notice that similar to the earlier examples, distribution is achieved automatically, and the set of nodes where it exists can be dynamic. Furthermore, since `lookup` operations can be executed concurrently even when their keys hash to the same bucket, the corresponding data also gets replicated (the DSM coherence protocol allows multiple copies to be made for reading). An obvious advantage of the automatic data distribution and replication is that threads can find the needed data locally, thereby reducing the communication overhead. These benefits in other paradigms require that the directory be implemented as a set of objects, one at each node, and explicit protocols be implemented for dealing with replication.

The directory data is maintained correctly by the DSM coherence protocol even though it gets distributed and replicated. Notice that if the data associated with each bucket is on a separate page or segment, a write to the page only results in invalidation messages to be sent to the nodes

```
entry int insert(char *key_to_insert, int value_to_insert)
{
   int i;
   i = hash_function(key_to_insert);
   write_lock(hash_table_entry[i].hash_lock);
   for (i = 0; i < NUM_BUCKET_ENTRIES; i++)
      if (hash_table_entry[i].key == NULL) /* empty entry */
      {
         hash_table_entry[i].key = key_to_insert;
         hash_table_entry[i].value = value_to_insert;
         break;
      }
   unlock(hash_table_entry[i].hash_lock);
   if (i == NUM_BUCKET_ENTRIES) /* table full */
      return(FAILURE);
   else
      return(SUCCESS);
} /* end of insert */

entry int delete(char *key_to_delete)
{
   int i;
   i = hash_function(key_to_delete);
   write_lock(hash_table_entry[i].hash_lock);
   for (i = 0; i < NUM_BUCKET_ENTRIES; i++)
      if (hash_table_entry[i].key == key_to_delete)
         break;
   if (i == NUM_BUCKET_ENTRIES)
      return_value = FAILURE;
   else
   {
      while ((i < NUM_BUCKET_ENTRIES ) and (hash_table_entry[i+1].key != NULL))
      {
         hash_table_entry[i].key = hash_table_entry[i+1].key;
         hash_table_entry[i].value = hash_table_entry[i+1].value;
      }
      hash_table_entry[i].key = NULL;
      return_value = SUCCESS;
   }
   return(return_value);
} /* end of delete */
end_class /* directory */
```

Figure 4: Directory Object *Insert* and *Delete* Methods

that have a copy of that page. Other pages are not invalidated. Since the serializable execution of the operations requires messages for implementing synchronization, the control messages sent for fetching data can again be combined with the synchronization messages. The overhead of DSM control messages is not incurred when a number of lookup operations are executed at a node between two update operations.

4 Related Work

Experience in the development of distributed applications falls in one of two areas. [F^+86] reports application development experience with the message based distributed system Charlotte. A comprehensive survey of shared memory programming in distributed systems can be found in [BT88], which focuses on operating system and language primitives for shared variables without the presence of physically shared memory. In addition, this paper presents a shared data object model of programming based on automatic data replication and a supporting language *Orca*. Other work in object-oriented distributed systems focuses on new languages and programming models like Concurrent Smalltalk[YT87a], Snoop[Luc87], Procol[BL89] and Suite [DC90]. These systems, however, are not mature enough to have spawned any concrete distributed system programming experience. A few applications like the distributed appointment calendar and name server programmed in Emerald language are discussed in [JLHB88]. However, these Emerald applications were programmed with immovable objects, and therefore cannot take advantage of multiple read-only copies of the same object on different compute servers, as in the Clouds distributed directory implementation. Argus[LS83] focuses on providing programming support for fault-tolerant distributed applications rather than concurrency, as demonstrated by their distributed mail application. [YT87b] covers to a large extent, recent research in development of special language features for parallel programming using object paradigms.

In all these languages and distributed systems, the application developer must still program component objects or processes of a distributed application separately.

5 Acknowledgements

The Clouds programming examples presented in this paper were implemented on Clouds with the help of Gautam Shah. The Clouds distributed shared memory system and the CC++ programming language was implemented by R. Ananthanarayanan. We would also like to thank Ray Chen, Sathis Menon, Ajay Mohindra, and Chris Wilkenloh, for their efforts in adding to the functionality of the *Clouds* kernel to make our programming efforts possible.

6 Conclusions

In this paper, we have demonstrated a shared memory programming paradigm for the Clouds distributed operating system that simplifies the task of distributed application development. This paradigm allows programmers to develop distributed applications without having to specify the particulars of distribution or runtime degree of concurrency. This programming model is possible on Clouds due to the operating system's orthogonal treatment of storage and computation through its distributed shared memory and object invocation mechanisms.

References

[Ana90] R. Ananthanarayanan. CC++ Preliminary Reference Manual. Georgia Tech Distributed Systems Laboratory, *Internal Document*, September 1990.

[BDS87] J. J. Bloch, D. S. Daniels, and A. Z. Spector. A Weighted Voting Algorithm for Replicated Directories. *Journal of the ACM*, 34(4), October 1987.

[BHJ+87] A. P. Black, N. C. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), 1987.

[BL89] J. V. D. Bos and C. Laffra. A Parallel Object Language with Protocols. *SIGPLAN Notices*, 1989. Special Issue: Proceedings of Object-Oriented Systems, Programming Languages, and Applications.

[BT88] H. E. Bal and A. S. Tanenbaum. Distributed Computing with Shared Data. *IEEE Computer*, June 1988.

[DC90] P. Dewan and R. Chaudhary. Experience with the Suite Distributed Object Model. *IEEE Workshop on Experimental Distributed Systems*, October 1990.

[Dij72] E. W. Dijkstra. *Structured Programming.* Academic Press, New York, NY, 1972.

[F+86] R. Finkel et al. Experience with Crystal, Charlotte and Lynx: Third Report. Technical Report Computer Sciences Technical Report 673, University of Wisconsin, Madison, November 1986.

[FP89] B. D. Fleisch and G. J. Popek. Mirage: A Coherent Shared Memory Design. *Proceedings of the 12th ACM Symposium on Operating System Principles*, 1989.

[Han87] P. B. Hansen. Joyce: A Programming Language for Distributed Systems. *Software Practice and Experience*, 17(4), 1987.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8), 1978.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computing Systems*, 6(1), 1988.

[LH89] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LS83] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3), 1983.

[LS88] B. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *SIGPLAN NOTICES: Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 260–267. ACM, Vol 23, No. 7, July 1988.

[Luc87] S. E. Lucco. Parallel Programming in a Virtual Object Space. *SIGPLAN Notices*, 22(12), 1987. Special Issue: Proceedings of Object-Oriented Programming SYstems, Languages, and Applications.

[RAK89] U. Ramachandran, M. Ahamad, and Y. Khalidi. Coherence of Distributed Shared Memory: Unifying Sychronization and Data Transfer. *Proceedings of the International Conference on Parallel Processing*, August 1989.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, MA, 1986.

[YT87a] Y. Yokote and M. Tokoro. Experience and Evolution of Concurrent Smalltalk. *SIGPLAN Notices*, 22(12), December 1987.

[YT87b] A. Yonezewa and M. Tokoro. *Object-Oriented Concurrent Programming*. The MIT Press, Cambridge, Mass., 1987.