Communication State Transfer for the Mobility of Concurrent Heterogeneous Computing

Kasidit Chanchio, *Member*, *IEEE*, and Xian-He Sun, *Senior Member*, *IEEE*

Abstract—In a dynamic environment where a process can migrate from one host to another host, communication state transfer is a key issue of process coordination. This paper presents a set of data communication and process migration protocols to support communication state transfer in a dynamic, distributed parallel environment. The protocols preserve the semantics of point-to-point communication; they guarantee message delivery, maintain message ordering, and do not introduce deadlock when blocking send or receive operations are performed during process migration. Analytical proofs and prototype implementation are conducted to confirm the correctness of the protocols. Analytical and experimental results show the proposed design is valid and has a true potential in network computing.

Index Terms—Communication protocol, process migration, distributed and parallel processing, point-to-point communication.

1 Introduction

Process migration is a basic function to support the mobility of computation intensive applications. It moves a running process from one computer to another. The migration may be through the computer network (distributed network migration) or over computers with different hardware/software environments (heterogeneous process migration). The motivations of process migration include load balancing, fault tolerance, data access locality, resource sharing, reconfigurable computing, and system administration, etc. [1], [2], [3]. Recent research shows process migration is necessary for achieving high performance via utilizing unused network resources [4], [5], [6], [7]. Process migration can also be used for portability. For example, users can migrate processes from a computing platform to an upgraded one at runtime. Process migration is a fundamental technique needed for the next generation of Internet computation [8]. However, despite these advantages, process migration has not been adopted in engineering practice due to its design and implementation complexities, especially under a network of heterogeneous computers.

The Scalable Network Of Workstations (SNOW) system [3] and its enhanced version, the High Performance Computing Mobility (HPCM) middleware [9], provide a distributed environment supporting user-level process migration. SNOW provides solutions for three problem domains for transferring the computation state, memory state, and communication state of a process, respectively. First, it provides methods to transfer the execution state of a process. A compiler analysis technique is proposed to select locations that allow process migration in the source program and to augment additional codes to carry process migration automatically [10]. Since the selected locations and the augmentation are performed to source code before compilation, the state transfer can be performed across heterogeneous machines. Second, we have developed mechanisms to transfer the memory state. A graph representation is introduced to model the data structures of a process. Methods to transform the data structures and their contents into machine independent information and vice versa are provided [10]. Based on our successes on computation state and memory state transfer, this paper presents a solution to transfer the communication state of a migrating process in a dynamic, heterogeneous, distributed environment.

Activities in a large-scale distributed environment are dynamic in nature. Adding process migration functionality makes data communication even more challenging. Three fundamental problems have to be addressed. First, we need to develop mechanisms to guarantee correct message deliveries during process migration. Second, if a sequence of messages is sent to a migrating process, correct message ordering must be maintained. Third, mechanisms to update location information of a migrating process have to be efficient and scalable. After a process migrates, other processes have to know its new location for future communications. The updating mechanisms should be efficient and scalable enough for large network environments. Moreover, in terms of software development, the communication state transfer needs to be integrated into the execution and memory state transfer seamlessly to form a process migration enabled environment.

Mechanisms to support correct data communication can be classified into two different approaches. The first approach is using existing fault-tolerant, consistent checkpointing techniques. To migrate a process, users can "crash" a process intentionally and restart the process from its last checkpoint on a new machine. Since global consistency is provided by the checkpointing protocols, safe data communication is guaranteed. Projects such as CoCheck [11] follow this approach. On the other hand,

- *K. Chanchio is with Oak Ridge National Laboratory, PO Box 2008, Building 5600, Oak Ridge, TN 37831-6016. E-mail: chanchiok@ornl.gov.*
- *X.-H. Sun is with the Department of Computer Science, Illinois Institute of Technology, 10 W. 31st St., Chicago, IL 60616. E-mail: sun@cs.iit.edu.*

Manuscript received 15 Feb. 2002; revised 13 Aug. 2003; accepted 26 Mar. 2004.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 115951.

mechanisms to maintain safe data communication during process migration can be implemented directly into the data communication protocol. SNOW, Charlotte [12], Freeze Free [13], and the Migration Transparent version of Parallel Virtual Machine (MPVM) project [14] are along the second direction. These systems are message-based and rely on the concept of communication channels. We follow the second direction because it is more scalable and less costly than the first. Process migration is important enough to receive an efficient mechanism in its own right. Further comparisons of related works are presented in Section 7.

We have developed data communication and process migration protocols working cooperatively to solve the aforementioned problems. Our protocol design is based on the concept of point-to-point connection-oriented communication. It is aimed at providing a robust and general solution for communication state transfer. Mechanisms to handle process state transfer are implanted to a number of communication operations, which could occur at data communication end points. These operations include send and receive operations in the data communication protocol and migration operations in the process migration protocol. They coordinate one another during the migration to guarantee correct message passing. The protocols are naturally suitable for large-scale distributed environment due to their inherited properties. First, they are scalable. During a migration, the protocols coordinate only those processes directly connected to the migrating process. Process migration operations are performed mostly at the migrating process, while communication peer processes are only interrupted shortly for the coordination. Moreover, the protocols update location information of the migrating process without the needs of broadcasting. Second, the protocols do not block data communication. They allow other processes to send messages to the migrating process during process migration. These two properties are quite beneficial for a large environment where the number of participating processes is high. Third, our protocols do not need old hosts to route messages to the migrating process. This property is desirable for an environment where a host can join or leave dynamically. Fourth, the protocols do not create deadlock. They prevent circular wait while coordinating a migrating process and its peers for migration. Finally, the protocols are simple in implementation and are practical for heterogeneous environments. They can be implemented on top of existing connection-oriented communication protocols such as Parallel Virtual Machine (PVM) (direct communication mode) [15], Message Passing Interface (MPI) [16], [17], and Transmission Control Protocol (TCP). We conduct empirical studies based on a prototype implementation on PVM.

The rest of this paper is organized as follows: Section 2 discusses our basic assumptions on the distributed computation model and communication semantics. In Section 3, we discuss the basic ideas of our protocols and present our data communication and process migration algorithms. Section 4 shows correctness proofs of the algorithms. We discuss protocol implementation in Section 5. Section 6 shows our experiments by migrating a communicating process while running a parallel benchmark. Section 7 discusses related works. Finally, Section 8 gives a summary and discusses future research.

2 Background

We consider a distributed computation as a set of collaborative processes $\{P_0, P_1, \cdots, P_N\}$ executing under a virtual machine environment. Each process is a user-level process, which occupies a separate memory space. The processes communicate via message passing.

In our design, a virtual machine environment is a collection of software and hardware to support the distributed computations. It has three basic components. First, a network of workstations is the basic computing resource. Second, a number of daemon processes residing on the workstations comprise a virtual machine. These daemons work collectively to provide resource accesses and management. A process can access the virtual machine's services via programming interfaces provided in forms of library routines. Finally, the third component is the scheduler, a process or a number of processes that control environmental-wide resource utilization. Its functionalities include bookkeeping and decision-making. Unlike in static distributed environments such as that supported by PVM and MPI, a scheduler is a necessary component of a dynamic distributed environment such as the Grid [18].

In our model, hosts can join or leave a virtual machine environment dynamically. We assume that the virtual machine daemon (or agent) is executed on a host when it joins the environment and terminated when the host leaves. These daemons can belong to a single user, like in the PVM model, or multiple users, like in peer-to-peer systems. We leave membership management to the virtual machine's implementation. Due to this dynamic nature, it is important that process migration mechanisms do not create residual dependency and data communication between the migrating process and others can be done without existence of old hosts.

The scheduler is a software component that overlooks distributed computation activities. In our protocol, the scheduler is required to: 1) keep track of hosts and processes in the virtual machine environment, 2) provide a scalable lookup service to locate a process, and 3) coordinate process migration operations on source and destination computers (more details in Section 3). The scheduler could have a centralized or distributed structure depending on the applications' needs. For example, mobile agent applications may use a centralized server like in Sethi@home and Napster to run the scheduler, while applications that run over multiple administrative domains may use hierarchical servers such as Domain Name Server (DNS), Lightweight Directory Access Protocol (LDAP). Moreover, one may extend scalable distributed location services used in peer-to-peer systems such as Chord [19] to support our scheduler's requirements. Since this paper focuses on the communication state transfer protocol, we refer to a centralized scheduler in our design for the sake of simplicity. Other scheduler designs can also be extended to support our protocol as long as they meet the requirements above.

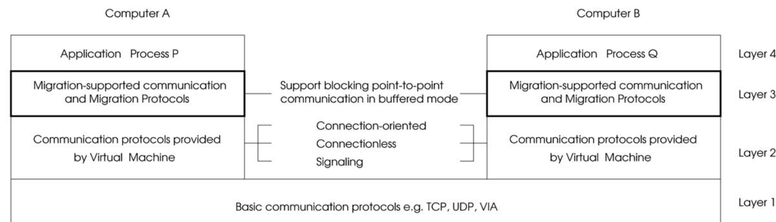

Fig. 1. The protocol stack layout.

2.1 Process Identification

We identify processes in distributed environment in two levels of abstractions: *application-level* and *virtual-machine-level*. In the application-level, a process is identified by a *rank* number, a nonnegative integer assigned in sequence to every process in a distributed computation.[1] The rank number allows us to refer to a process transparently of its whereabouts. On the other hand, the virtual machine includes location information of a process in its naming scheme. A *virtual-machine-lvel process identification* (*vmid*) is a coupling of workstation and process identification numbers. They both are nonnegative integers assigned sequentially to workstations and processes created on each workstation, respectively. The mappings between rank and *vmid* are maintained in a process location (PL) table, where the PL table is stored inside the memory spaces of every process and the scheduler. While the rank numbers are given only to application-level processes, the *vmid* is assigned to every process in the environment, including the scheduler and virtual machine daemons. We assume that the daemons do not migrate and the scheduler is always reachable.

2.2 Process Migration Software System

To develop migration-enabled distributed applications, SNOW transforms source code into a migration-enabled code [10]. Mechanisms to migrate the execution and memory state of a process across heterogeneous machines are annotated into the source code during the transformation. Source code annotation is a common approach in heterogeneous process migration research [20], [21], [2], [10]. We assumed that the migration-enabled code is distributed to all possible source and destination computers of process migration and then compiled and linked with process migration supported libraries to generate migration-enabled executable. Our communication state transfer mechanisms are implemented in one of these libraries.

With supervision of the scheduler, a process migration is conducted directly via remote invocation and network data transfers. When a user wants to migrate a process, he or she sends a request to the scheduler, which, in turn, decides the destination computer and remotely invokes the migration-enabled executable to wait for process state transfer. We call this invocation *process initialization*. Then, the scheduler sends a migration signal to the migrating process. After the migrating process intercepts the signal, it coordinates the initialized process to transfer process state information. We discuss more details in Section 3. Finally, while the migrating process terminates, the initialized process resumes execution.

1. The rank number indexing can be replaced by any sortable naming scheme for generalization.

2.3 Communication Characteristics

Our protocols support distributed applications that use blocking point-to-point communication in buffered modes. Assuming a message content is stored in a memory buffer, the send operation blocks until the buffer can be reclaimed, and the receive operation blocks until the transmitted message is stored in the receiver's memory. The sender process does not coordinate with the receiver for data transmission. Once the message is copied into internal buffers of an underlying communication protocol, the sender process can continue.

Fig. 1 shows the protocol stack layout of the communication system for the process migration environment. The lowest layer is the OS-supported data communication protocols between computers. The second layer includes communication protocols provided by the virtual machine built on top of the first communication layer. The virtual machine provides three basic communication services:

- *Connection-oriented Communication Service:* Our protocols rely on connection-oriented communication to create a bidirectional, First-In-First-Out (FIFO) communication channel between two processes. We assume that messages sent through the communication channel do not get lost and arrive in order. In case messages are sent between machines with different platforms, we also assume that the protocol in this layer handles data conversion.
- *Connectionless Communication Service:* In our protocols, connectionless communication is used to deliver the connection request control message and its acknowledgment (or rejection) between processes. The message is routed from one process to another through the virtual machine. In our design, the virtual machine daemon is extended to handle connection requests in process migration circumstances. We discuss more details in the next section.
- *Signaling Service:* We assume that a process can reliably send a signal to another process on the virtual machine regardless of their locations. We also assume that signals are transmitted from a sender to a receiver process in order. Finally, we assume that a signal cannot interrupt communication (send or receive) events. It only interrupts a computation event. If the signal arrives at a process during

communication events, its signal handler will be invoked only after the communication events finish.

Most virtual machine environments such as PVM [15] and MPI [16] support the three basic services. The third layer is the focus of this work. It describes our communication state transfer mechanism consisting of migration-supported data communication and process migration protocols to be discussed in the next section. The protocols provide programming interfaces to support point-to-point communication and process migration of application processes in the fourth layer.

3 Communication State Transfer Mechanism

This section presents basic ideas of mechanisms to migrate the communication state and algorithms describing the data communication and process migration protocols. Since data communication at the application-level is performed on top of the connection-oriented communication protocol, we define the communication state of a process to include all communication connections and messages in transit at any moment in the process's execution. To migrate the communication state, one has to capture the state information, transfer it to a destination computer, and restore it successfully.

3.1 Basic Ideas

Migrating a communication state is nontrivial since various communication situations can occur during process migration. In our protocol designs, three basic circumstances are considered.

3.1.1 Capturing and Transferring Messages in Transit

To capture messages in transit, processes on both ends of a communication channel have to coordinate with each other to receive all messages. The coordination mechanism is based on the work of Chandy and Lamport [22] and will be discussed later in the migration algorithm. As a result of the coordination, messages in transit are drained from the channels and stored in a temporary storage in process memory space, namely, the *received-message-list*. The channels will also be closed down at the end of the coordination.

The use of the received-message-list effects the design of our receive operation. Since messages could be stored in the receive-message-list before needed, the receive operation has to search for a wanted message from the list before taking a new message from a communication channel. In case the new messages are not wanted, they would be appended to the list until the wanted message is found. After messages in transit are captured and existing communication connections are closed down, one may consider the messages stored in the received-message-list of the migrating process as a part of the process's communication state, which has to be transferred to the destination computer.

3.1.2 Migration-Aware Connection Establishment

To handle data communication between unconnected processes, the connection establishment mechanisms have to be able to detect migration activities on the connecting processes and automatically resolve the problem. Since our message passing operations only employ send and receive primitives and do not support explicit commands for connection establishment, the establishment mechanisms are installed inside the send and receive operations hidden from the application process. To establish connections, we employ the sender-initiated technique, where a sender sends a connection request to its intended receiver process. Having process migration in the picture, the establishment mechanisms must be able to detect the migration (or past occurrences of the migration). In our design, the migration is detected once the sender receives a denial to its connection request. The rejection message could come either from the virtual machine or the migrating process. The virtual machine sends a rejection message in case the migrating process has already been migrated. On the other hand, the migrating process rejects connection requests if it is performing migration operations. The migrating process starts migration operations when it receives a migration instruction from the scheduler and finishes the operations when process state transfer completes. If the migrating process receives connection requests during that time, it will send denial messages back to the requestors. Once the migration is detected, the sender consults the scheduler to locate the receiver. After getting a new location, the sender updates the receiver's location, establishes a connection, and sends messages. Thus, the sender updates the location of a migrating process "on demand" when it wants to send a message there.

In our design, the sender process will always be able to send messages to a receiver regardless of process migration situations. If the receiver process has already migrated, the sender will normally establish a connection with the receiver and send the messages. In case the receiver is migrating, the sender will establish a connection with the receiver's initialized process, which always receives the transmitted messages into its receive-message-list. Therefore, the sending operation is not blocked during the receiver's migration.

3.1.3 Communication State Restoration

The scheme for the restoration of communication state on a new (or initialized) process can be addressed in two parts. First, contents of the receive-message-list forwarded from the migrating process are inserted to the front of the receive-message-list of the new process. This scheme restores the messages, which are in transit during the migration. Second, messages sent from a newly connected process to the new process are appended to the end of the list. This scheme ensures message ordering.

3.2 Algorithms

We have developed a number of algorithms based on the previously mentioned conceptual designs. The data communication algorithms consist of send and receive algorithms which take care of the connection establishment and the receive-message-list, while the process migration algorithms consist of two algorithms which run concurrently on the migrating and new processes to carry process migration. A global variable $Connected$ represents a set of rank numbers of connected peer processes. An array pl represents the PL table. The $vmid$ of process P_i is stored in $pl[i]$. A

```
send (m, dest)
 1: if (dest ∉ Connected) then
 2:         cc[dest] = connect(dest);
 3: end if
 4: send m along the cc[dest] communication channel;
```

Fig. 2. The send algorithm.

global variable *Closed_conn* is used for process coordination and has a zero value from the start.

3.2.1 Data Communication Algorithms

In our design, the send algorithm initiates data communication between processes by sending a request for connection establishment to the receivers. In case the receiver cannot be found due to process migration, the send algorithm will consult the scheduler to locate the receiver. Once the receiver's location is known, the sender establishes connection and sends messages to the receiver process. Fig. 2 shows the send algorithm, where a communication connection must be created before message transmission.

The connection establishment mechanisms are described in the *connect*() function in Fig. 3. This function will terminate after it has successfully established a communication connection with a receiver process *dest* or has been notified of the receiver's termination. The function starts by sending the connection request `con_req` to a receiver process (line 2). If the receiver is ready to receive messages, it will send `con_ack` back in return (line 3). The *connect*() function will then call *make_connection_with*() to create a new communication channel with the receiver. On the other hand, if the receiver has already migrated or is migrating, the rejection (`conn_nack`) message will be delivered back to the sender (line 9). As a result, *connect*() will consult the scheduler and update the receiver's location in the sender's PL table or terminate the function if it learned from the scheduler that the receiver has terminated (lines 9 to 15).

```
connect(dest)
 1: while (dest ∉ Connected) do
 2:        send conn_req to pl[dest];
 3:        if (receive conn_ack from pl[dest]) then
 4:              cid := make_connection_with(pl[dest]);
 5:              Connected := {dest} ∪ Connected;
 6:        else if (receive conn_req from any process p) then
 7:              cc[p] := grant_connection_to(p);
 8:              Connected := {p} ∪ Connected;
 9:        else if (receive conn_nack from pl[dest]) then
10:              consult scheduler for exe status
                       and new_vmid of P_dest
11:              if (status = migrate) then
12:                   pl[dest] := new_vmid;
13:              else report "error: destination terminated";
14:                   return error; end if
15:              end if;
16:        end if;
17: end while;
18: return cid;
```

Fig. 3. Functions *connect*().

```
recv (src, m, tag)
 1: While (forever) do
 2:        if (m is found in received_message_list) then
 3:            return m, delete it from the list, and return
                   to a caller function;
 4:        end if
 5:        get a new data or control message, n;
 6:        if (n is data message) then
 7:            append n to received_message_list;
 8:        else (handle control messages)
 9:            if n is conn_req then
10:                 cc[sender of n] :=
                        grant_connection_to(sender of n);
11:                 Connected := {sender of n}∪
                           Connected;
12:            else if n is peer_migrating then
13:                 close down the connection with the
                        sender of peer_migrating;
14:                 Closed_conn := Closed_conn + 1;
15:            end if;
16:        end if;
17: end while
```

Fig. 4. The recv algorithm.

The *connect*() function also contains mechanisms to establish connections between parallel processes. While waiting for a response to its connection request, if the function receives a `conn_req` (line 6), it will call the *grant_connection_to*() function to return the `conn_ack` and wait until the requestor executes *make_connection_with*() to complete the creation of a new communication channel.

In responding to `conn_req`, if the receiver process is migrating, it will send a `conn_nack` message back to the requestor. On the other hand, in case the receiver process has already migrated, the `conn_nack` is sent back to the migrating process by the virtual machine. In our design, we extend the virtual machine daemon to keep records of connection requests being routed through it. These records are deleted when either the connection acknowledgment or rejection are routed back to the requester. If the target daemon cannot find the target process or detects its termination but has a number of connection requests pending responses, the target daemon will send the rejection (`conn_nack`) message back to the requestor's daemon, which then forwards it to the requestor process. On the other hand, if the target daemon does not exist because the target machine has resigned from the virtual machine, the requestor's daemon will send the rejection message back to the requestor.

The receive algorithm, as shown in Fig. 4, is designed to collect messages in an orderly manner in process migration environment. The algorithm stores every message arrived at a process in the received-message-list of the receiver process. The receive algorithm also has functionalities to help migrate its peer processes (lines 12 to 14). In case a process is running a receive event while one of its connected peer processes is migrating, the receive event may receive the `PEER_MIGRATING` control message from the migrating peer. This control message is a special message sent from a migrating peer to indicate the last message sent from the peer and instruct the receiver to

```
Process: P_i

migrate()
 1: if (migrate_request is received) then
 2:     inform the scheduler migration_start;
 3:     get new_vmid of P_i from scheduler;
 4:     Inform the local daemon to reject every arriving
            con_req, and reject all con_req
            already arrived at the migrating process;
 5:     Send disconnection signal
            and peer_migrating;
 6:     Receive incoming messages to receive-message-list
            until getting end-of-messages or peer_migrating
 7:     Close all existing connections;
 8:     Send received-message-list to the new process;
 9:     Perform exe and memory state collection;
10:     Send the exe and memory state to the new process;
11:     Terminate;
12: end if;
```

Fig. 5. The $migrate()$ algorithm on the migrating process.

close connection with the peer process. The reception of this message implies that all of the messages sent from the migrating process earlier through the communication channel have already been received. The receive algorithm also contains mechanisms to assist connection establishment (lines 9 to 11). While the receive event may be waiting to receive a message, if a `conn_req` message arrives, the receive algorithm will grant a connection and proceed to establish a connection with the requestor.

3.2.2 Process Migration Algorithms

The process migration protocol involves algorithms to transfer process state across machines. They are the migration and initialization algorithms shown in Figs. 5 and 7, respectively. On the migrating process, the migration algorithm first checks whether a `migration_request` signal has been sent from the scheduler and is intercepted by the migrating process. If so, it contacts the scheduler to get information about an initialized process. Then, the algorithm rejects further communication connection so that it can coordinate with existing communication peers to receive messages in transit into the receive-message-list. At lines 2 and 3 of Fig. 5, the migrating process informs the scheduler that the migration operation has started and receives the $vmid$ of the initialized process from the scheduler. Recall that the scheduler has already initialized a process to wait for state transfer on a destination machine before sending the migration request to the migrating process. Next, the $migrate()$ algorithm rejects further connection requests (`con_req`) at line 4. In doing so, the algorithm sends a message to inform the local virtual machine daemon to reject all *future* incoming connection requests to the migrating process. Then, it rejects all `conn_req` that have already arrived. Upon receiving `conn_nack`, the rejected peers consult the scheduler and redirect their requests to the initialized process.

In process coordination, the migrating process sends `disconnection` signals and `peer_migrating` control messages out to all of its connected peers. The

```
Process: P_j

disconnection_handler()
 1: if (Closed_conn = 0 ) then
 2:     receive messages into receive-message-list until
                peer_migrating is found;
 3:     close the connection to the sender of
                peer_migrating;
 4: else if (Closed_conn > 0) then
 5:     Closed_conn := Closed_conn - 1;
 6: end if;
```

Fig. 6. The diconnection handler algorithm.

`disconnection` signal will invoke an interrupt handler on the peer process if the peer is running a computation event. Fig. 6 shows the algorithm of the $disconnection_handler()$ interrupt handler. When invoked, the handler keeps receiving messages from existing communication connections until the `peer_migrating` message is found and then closes the connection it receives the `peer_migrating` from. In case the peer process is running a receive event, the receive algorithm may detect `peer_ migrating` while waiting for its desired messages. The $recv()$ algorithm will close down the connection afterward (see line 13 of Fig. 4). Note that, when the connection is closed on either case above, the *end-of-message* message is sent along the channel as the last message to the process on the other end. If the peer is running a send event, the coordination is delayed until the event finishes.

To prevent the repetition of coordination efforts by $recv()$ and $disconnection_handler()$ algorithms, we use the variable $Closed_conn$ to indicate the number of `peer_migrating` messages that have been received and processed before the disconnection handler is called. If the variable is greater than zero, the coordination effort has already been done and will not be repeated. On the other hand, if the variable is zero, the disconnection handler will coordinate with the migrating process.

On the migrating process, the $migrate()$ algorithm waits to receive either *end-of-message* or `peer_migrating` as a last message from connected peers. While $migrate()$ expects an *end-of-message* message from a nonmigrating peer, it expects a `peer_migrating` from a simultaneously migrating process. The migrating process receives messages from

```
Process: P_i

initialize()
 1: All con_req messages are accepted beyond this point;
 2: Receive received-message-list of the migrating process;
 3: insert it to the front of the original received_message_list;
 4: Receive "exe and mem state" of the migrating process;
 5: inform the scheduler restore_complete;
 6: wait for contents of the PL table and old_vmid
            from the scheduler;
 7: inform the scheduler migration_commit;
 8: Restore process state;
```

Fig. 7. The $initialize()$ algorithm on the initialized process.

existing communication channels and saves them to its receive-message-list until it receives the above messages from every connected peer. Then, the migrating process closes the existing communication channels and sends the contents of the receive-message-list to the initialized process. After that, the algorithm collects the execution state and memory state of the migrating process and sends the state information over to the destination machine [10]. Finally, the algorithm terminates the migrating process.

On the destination computer, a new process is initialized to wait for process state transfer. Fig. 7 shows the initialization algorithm. The initialized process will accept any connection requests from start. At line 2 of Fig. 7, the algorithm waits for the contents of the received-message-list from the migrating process. During the wait, if any `con_req` arrives, the initialized process will grant connection establishment. If the wanted message has not arrived, the process keeps receiving new messages and appending them to its received-message-list. We should note that, while connections are granted on the initialized process, they are rejected on the migrating process. Based on the send algorithm, the rejection will cause connection requests to be redirected to the initialized process. After $initialize()$ finishes the operation at line 2, it inserts the contents of the received-message-list from the migrating process in front of the local received-message-list to maintain message ordering. Then, the algorithm waits for the execution and memory state of the migrating process. After the state information is received, the initialized process informs the scheduler of migration completion and updates the PL table. Then, the $initialize()$ algorithm restores the process state and, finally, resumes program execution.

4 Correctness Analysis

Since activities generated by process migration are additional to what is generated in nonmigration situations, the distributed computation logic must be preserved. In this section, we analyze the correctness of the newly proposed protocols. For descriptive purposes, we define each application-level process as a sequence of events [23]. The events of our interest include the computation, send, receive, and migration events. The correctness of our algorithms is analyzed along the following aspects:

1. Process migration does not introduce any deadlock.
2. Process migration terminates and does not block the progress of distributed computation.
3. There is no message loss because of process migration.
4. Despite process migration, message ordering semantics of point-to-point communication are preserved.

Here, nonblocking implies no deadlock. We will prove aspects 1 and 2 together.

4.1 Deadlock and Migration Termination

First, we show that process migration does not introduce a deadlock or prevent progress of distributed computation. We focus on the effects of a migration event on various message-sending situations because the only waiting in our design is a migration event waiting for a send event to finish. We assume that the original distributed application does not create any deadlock.

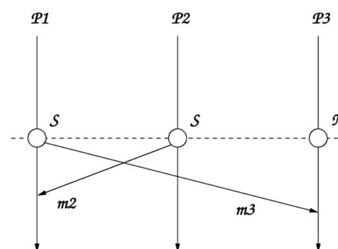

Fig. 8. A communication situation.

Theorem 1. *If two or more processes communicate and then one of them migrates, the migration does not cause deadlock and does not block the other processes from sending messages.*

Proof. The communication protocol only introduces one waiting for the sender process awaiting the response of a connection request. However, based on the $send()$, $migrate()$, and $initialize()$ algorithms, a sender process can either send a message via an existing communication channel to the migrating process or via a new communication channel to the initialized process. Thus, no blocking due to the connection request would occur. Also, process migration does not cause control flow blocking because messages sent in either case will be received into the receive-message-list as soon as they arrive. Thus, process migration does not block message-sending. It does not introduce deadlock. □

We illustrate the proof using an example as shown in Fig. 8, where S and M represent the send and migration events, respectively, and the dashed line that goes across the processes indicates a timeline when a migration occurs at M on $P3$. We assume that there are three processes, $P1$, $P2$, and $P3$, in the environment and the migration event occurs at $P3$. Suppose $P3$ and $P2$ are connected, the migration event has to coordinate $P2$ for disconnection. Since $P2$ is sending a message, the disconnection is possible only after the sending finished. Consequently, the migration has to wait. If send events on $P2$ and $P1$ block simultaneously, a circular wait on the three processes could occur.

However, under our protocols, for the given situation, if $P1$ already has a connection with $P3$, the migrating process, $m3$ would be received to the receive-message-list by the $migrate()$ algorithm. Thus, the circular wait does not occur. On the other hand, suppose that $P1$ and $P3$ are not connected, the connection request would be redirected to the initialized process. A connection would be granted by the operations at line 2 of the $initialize()$ algorithm. As a result, $m3$ is received into the receive-message-list of the initialized process. Hence, no blocking or deadlock occurs.

We now show that the migration event terminates.

Lemma 1. *If two or more processes communicate and then one of them migrates, the migration event terminates.*

Proof. We show that the $migrate()$ and $initialize()$ algorithms finish.

Since a process always finishes sending messages, regardless of migration events (Theorem 1), the

end-of-message messages will eventually be received from every existing communication channel. Thus, the $migrate()$ algorithm will proceed to send its receive-message-list and terminate the migrating process.

The $initialize()$ algorithm is defined to keep accepting connection requests and receiving new messages until it receives the receive-message-list from the migrate() algorithm. Since the receive-message-list will be sent by the $migrate()$ algorithm, the $initialize()$ will finish. □

Since process migration terminates (Lemma 1) and does not block any message-sending (Theorem 1), it does not block progress of distributed computation.

4.2 Message Deliveries

Theorem 2. *If two or more processes communicate and then one of them migrates, every message arrives at their desired destinations.*

Proof. From the proof of Theorem 1, migration does not block message-sending. Based on the $recv()$, $disconnection_handler()$, and $migrate()$ algorithms, all messages in transit during the migration would be received. Every message will be sent successfully. Therefore, they will reach their destinations unless they are redirected by the communication protocol. In the protocol, communication redirection occurs only when the receiving process is migrating (see Fig. 3, the $connect()$ algorithm). These redirected messages are received by the initialized process and kept in the received-message-list (Fig. 7, the $initialize()$ algorithm). The initialized process is the migrating process when migration finishes. Therefore, every redirected message also reaches its destination. □

4.3 Message Ordering

If there is no redirection, by the FIFO assumption, all the messages to the migrating process will be received in order. A communication redirection occurs when the receiving process migrates. We have the following theoretical results:

Theorem 3. *If process $P1$ passes messages $m1$ and $m2$ subsequently to process $P2$ and a migration occurs on the receiver process $P2$ during data communication, both messages will be received by $P2$ in order.*

Proof. If both $m1$ and $m2$ are sent before or after migration, by the FIFO assumption, they will be received in order. If $m1$ is sent before migration and $m2$ is sent during or after migration, then $m1$ is received before migration and $m2$ is received after migration. They are received in order.

If $m1$ is sent during the migration, we have the following possibilities:

1. $m1$ is received by the migrating process and stored in the received-message-list (ListA) (see Fig. 5, the $migrate()$ algorithm). In this case:
   a. If $m2$ is received by the migrating process, by the FIFO assumption, $m1$ and $m2$ will be stored in ListA in order;
   b. If $m2$ is redirected to the initialized process and stored in the received-message-list (ListB) (see Fig. 6, the $initialize()$ algorithm), by our protocol, $P2$ will read ListA before ListB after migration. The message order is preserved.
   c. If $m2$ is sent after migration, by our protocol, $P2$ will read ListA and ListB before receiving any new message. $m1$ and $m2$ are received in order.
2. $m1$ is received by initialized process and stored in ListB. In this case, since $m2$ is sent after $m1$, $m2$ is either stored in ListB or sent after migration. In the former, by the FIFO assumption, $m1$ and $m2$ will be stored in ListB in order. In the latter, by our protocol, $P2$ will read ListB before receiving any new message after migration. $m1$ and $m2$ are also received in order.

If $m1$ is sent after migration, then $m2$ must also be sent after migration. The FIFO applies. They are also received in order. □

Lemma 2. *If process $P1$ passes messages $m1$ and $m2$ subsequently to process $P2$ and $P1$ migrates during the communications, $m1$ and $m2$ will be received by $P2$ in order.*

Proof. If both $m1$ and $m2$ are sent before or after the migration, by the FIFO assumption, they will be received in order. If $m1$ is sent before the migration and $m2$ is sent after the migration, then, by the $disconnection_handler()$ and $migrate()$ algorithms, the $m1$ is received by $P2$ before $P1$ migrates. Since $m2$ is sent after the migration, $m2$ is received after $m1$. The messages are received in order. Based on our communication protocol, $P1$ does not send message during migration. That concludes the proof. □

Theorem 4. *Theorem 1, Theorem 2, and Theorem 3 can be extended to general situations, where two or more processes may migrate simultaneously.*

Proof. Since we only consider point-to-point communications, a proof of two processes migrating simultaneously is sufficient.

Assume $P1$ and $P2$ migrate simultaneously. If there is no communication between $P1$ and $P2$ during migration, then $P1$'s and $P2$'s migrations are two unrelated instances. The previous analytical results can apply. If there is a communication between $P1$ and $P2$ during migration, then it must have one of the processes, say $P1$, sending a message to another process, say $P2$. By the definitions of our protocol, $P1$ cannot send a message during migration. So, the message has to be sent before or after $P1$ migrates. In either case, the message will be handled by the protocol as a single $P2$ process migration. By Theorems 1, 2, and 3, the message will not be blocked, will be received, and will be received in order. Since this is true for arbitrary messages and for both processes, it concludes the proof. □

5 Implementation

We have implemented the proposed communication state transfer mechanism within the SNOW process migration system and performed a number of experiments. Our

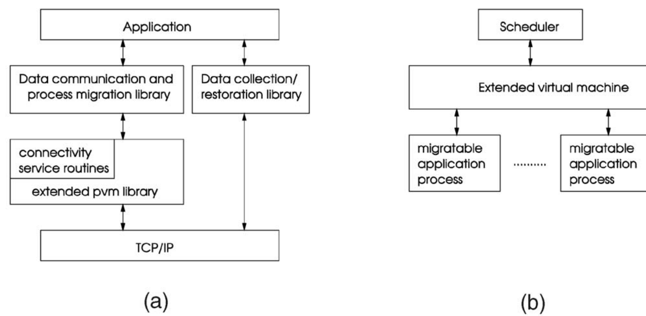

Fig. 9. (a) SNOW protocol stack and (b) runtime system.

system consists of a programming library that hosts our protocol implementation and a virtual machine. Fig. 9a shows the protocol stack and programming libraries used to support application programs. We implement our data communication and process migration algorithms in the SNOW data communication and process migration library. We build the library on top of an extended PVM communication library. The connectivity service macros and routines are added to the PVM library to support interaction between a sender process and the scheduler during a connection establishment. As shown in Fig. 9a, the data collection and restoration library is a separate software module that contains mechanisms to collect, transfer, and restore execution and memory state across heterogeneous machines [10].

From Fig. 9b, the virtual machine and scheduler are employed to monitor and manage the runtime environment. We extend the PVM virtual machine to handle process creation and termination and to pass control messages and signals between machines. In our implementation, we modify PVM's daemon to keep records of connection requests and reject them when the receiver processes or machines do not exist. We also implement a simple scheduler to oversee process migration. The scheduler handles process migration requests, assists process coordination during the migration (as previously discussed in Section 3), and performs bookkeeping on process migration records.

5.1 Extensions to PVM

PVM uses a sender initiated approach for its automatic connection establishment. When a sender process calls pvm_send, it sends a connection request and waits for a response. The receiver process responds to the request when it calls pvm_send or pvm_recv. The sender makes a connection after it receives an acknowledgment from the receiver.

We have extended functionalities of pvm_send by adding connectivity service macros to its original connection establishment mechnaism in PVM source code. The macros allow pvm_send to: 1) detect a connection denial message from the receiver or the virtual machine, 2) send an inquiry message to the scheduler and wait for a reply, and 3) pass along the information from the scheduler back to its caller. We have also added a number of routines to manage communication channels to the PVM library. We found that the extensions only cause small changes to the PVM source as most of our protocol is implemented in the next layer. Our modification does not change pvm_send's functionalities in nonmigration circumstances.

5.2 Communication Interfaces and Migration Macros

We implement the proposed protocols in the SNOW Communication and Process Migration library. The send and receive algorithms are implemented in the following two functions:

```
int snow_send(int dst_id, int tag);
int snow_recv(int src_id, int tag);
```

We use them to replace pvm_send and pvm_recv in the application source code, respectively. Remember that, in our design, only a computation event can be interrupted by a signal. In the implementation, we use SIGUSR1 to represent a migration request, while SIGUSR2 is used for the disconnection signal in the algorithms. To prevent the disconnection signal handler from invocation during communication events, we call sighold(SIGUSR2) and sigrelse(SIGUSR2) at the beginning of snow_send and snow_recv and at the end of them, respectively. We have also modified the PVM daemon to convert the signal values when they are transmitted across heterogeneous computers. The function snow_send implements the $send()$ algorithm and invokes pvm_send| to send a message. In case the receiver migrates, it uses the connectivity service, extended to pvm_send, to find the receiver's new vmid. The snow_recv implements the $recv()$ algorithm and runs on top of pvm_recv. This function maintains the receive-message-list, receives incoming messages, and coordinates with the migrating process.

To handle process migration, we implement the $migrate$ algorithm in macros and insert the macros at poll-points in the source code. Recall that a poll-point is a location where process migration can occur. At runtime, if a process intercepts a migration request (SIGUSR1) from the scheduler, it will perform the migration macros at the next poll-point. On the destination machine the $initialize()$ algorithm is implemented in the form of macros as well (see [10]) for details.

6 Experimental Results

As a case study, we show here the application of our prototype implementations on the parallel kernel Multi-Grid (MG) benchmark program [24]. The program is written in C and originally runs under the PVM environment. The kernel MG program is an SPMD-style program executing four iterations of the V-cycle multigrid algorithm to obtain an approximate solution to a discrete Poisson problem on a $128 \times 128 \times 128$ grid.

The kernel MG program applies block partitioning to the vectors for each process. A vector is assigned to an array of size $16 \times 128 \times 128$ when eight processes are used. Since each process has to access data belonging to its neighbors, the data must be distributed to the computations that need them. Such distribution occurs periodically during execution. Every MG process transmits data to its left and right neighbors. Therefore, the communication is a ring topology

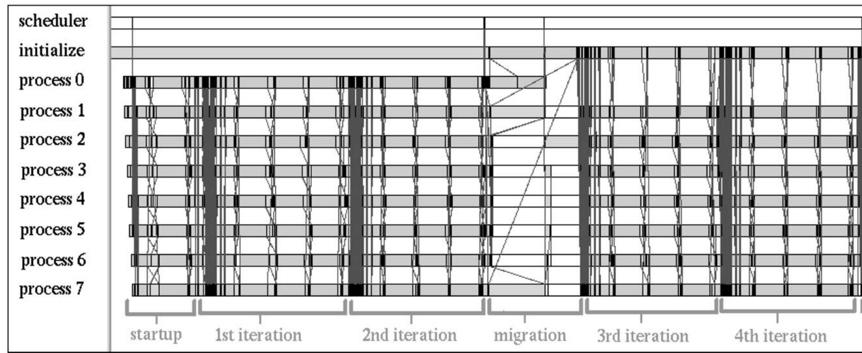

Fig. 10. A space-time diagram with a process migration.

[24]. Data communication of the MG program is nontrivial. The application exercises extensive interprocess communication; over 48 Mbytes of data on the total of 1,472 message transmissions.

We have annotated the program with process migration operations and linked the annotated program to our protocols. In our experimental settings, we generate eight processes ranking from process 0 to 7. Each stays on a different machine. Then, we force process 0 to migrate when a function call sequence main $\rightarrow$ kernelMG is made and two iterations of the multigrid solver inside the kernelMG function are performed. For reliable data communication, we change the PVM send and receive routines in source code to those of our communication library. As a result, while process 0 migrates, others would be executing without prior knowledge of the migration incident. Note that no barrier is used to synchronize the processes during a migration.

6.1 Communication Behaviors

Experiments are conducted to study process migration of the kernel MG program. In the first experiment, we analyze communication behaviors during a process migration. Fig. 10 shows an Xwindow Graphical Interface for PVM (XPVM) generated migration diagram of the kernel MG program running on a cluster of 10 Sun Ultra 5 workstations connected via 100Mbit/s Ethernet. We set up two machines to run the scheduler and an initialized process. Process 0 spawns seven other processes on different machines, as shown in Fig. 10. Note that a line between two timelines indicates a message passing which starts at the point where pvm_send is called and ends when the matching pvm_recv returns. Since our communication routines are implemented on top of PVM, these lines also show what is going on inside our prototype implementation. Also, since we implement the execution and memory state transfer directly on TCP, their network transmissions are not displayed in this diagram. In Fig. 10, the execution is separated into different stages. First, all pvmmg processes establish connections, distribute data, and perform the first two iterations. Then, the migration is performed by relocating process 0 to the initialized process. After the migration, the kernel MG resumes the rest of its computation.

We have observed a number of interesting facts through the space-time diagram. First, since the migrating process has connections to all other processes (due to the original setup of the benchmark), it has to send disconnection signals and peer_migrating messages to all of them. When the migration starts, we find that there is no message sent to the migrating process from any of the connected peers. Therefore, the migrating process does not receive any messages into the receive-message-list when it performs message coordination with connected peers. After the coordination, every existing connection is closed. This operation is shown in area A in Fig. 11.[2] Second, while process 0 migrates, other processes proceed with their data exchanges normally. As long as a process does not have to wait for messages, its execution continues. Area B in Fig. 11 shows such an execution. In normal operation, the kernel MG process would exchange messages of size 34,848 followed by 9,248 and 2,592, etc., with its near neighbors. In the area B, some nonmigrating processes proceed with the exchanges up to the message size 2,592. Then, they have to wait for certain communications to finish before proceeding further, until only process 4 can transmit messages of size 800 to its neighbors (area C in Fig. 12). Beyond this point, the nonmigrating processes have to wait for process 0 to start sending data after the migration finishes.

Finally, following the multigrid algorithm, two messages of size 34,848 bytes are sent from processes 1 and 7 to process 0 at the start of the third iteration. Since process 0 is migrating and the communication channels between 0 and 1 and between 0 and 7 are already closed, both senders have to consult the scheduler to acquire the location of the initialized process for establishing new connections. Such communications are shown by the two lines captured by label D in Fig. 12. By a closer analysis of trace data, we find that the communication channels are established before the execution and memory state restoration of the migrating process, allowing the senders (processes 1 and 7) to send their data to the initialized process in parallel with the execution and memory state restoration. Since the sent data are copied to low-level Operating System (OS) buffers, the sender process can proceed with the next execution so that the computation can continue in area C. The sent data are received after the restoration finishes, resulting in XPVM displaying two long lines cut across the migration time frame, as shown in area D in Fig. 12. After that, the migrating process starts resuming its execution, sends two

2. We have performed 10 experiments under the same testing configuration and found that the timing results appeared to be very similar. There is no forwarding message in all tests. The communication pattern during the migration also does not exhibit any variation.

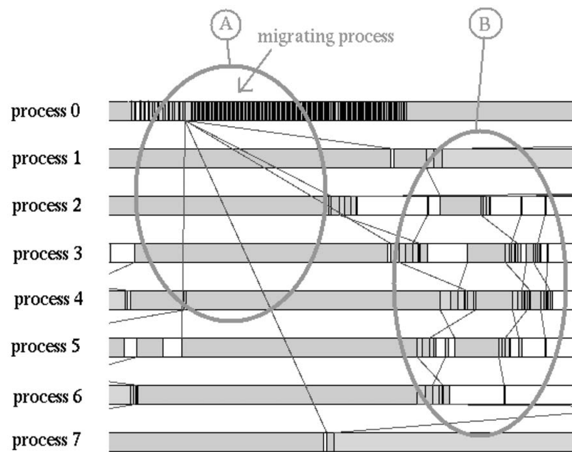

Fig. 11. A diagram shows the beginning of process migration.

messages of size 34,848 bytes back to its neighboring peers, and continues the multigrid computation. These observations confirm that the case study represents general communication situations and validates the proposed communication protocols.

6.2 Overheads

In the second experiment, our objective is to exam the overheads of our communication and migration protocols and the cost of migration. Table 1 shows the measured turnaround time of the parallel MG benchmark. All timing reported is the average of 10 measurements. From the table, the `original` column shows performance data of the original code running on PVM, while the `modified` column shows that of the migration-enabled process running without process migration. Finally, the `migration` column presents performance of the migration-enabled process running with a migration.

By comparing the communication time in `modified` to that of `original`, the overhead is evidently small. Although over 48 Mbytes of data on the total of 1,472 messages are transmitted during execution, the total overhead of the modified code is only about 0.144 seconds. We believe such a small overhead is due to the thin layer protocol design on top of PVM.

By comparing the execution time of the migration to that of the original code, we find that a migration incurs about

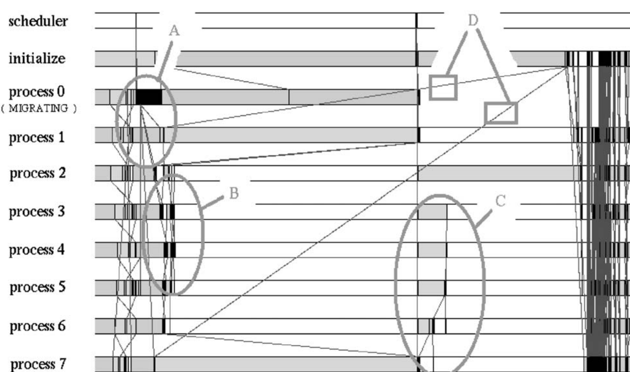

Fig. 12. The space-time diagram of a process migration.

TABLE 1
Timing Results (in Seconds) of the Kernel MG Program

Total	original	modified	migration
Execution time	16.130	16.379	18.833
Communication	4.051	4.205	6.647

2.2922 seconds higher turnaround time. Although processes can continue execution while the process `0` migrates, due to the communication characteristic of the kernel MG program, they eventually all have to wait for messages from process `0` after its migration. The waiting contributes to the migration cost. The migration transmits over 7.5 Mbytes of execution and memory state data. In detail, we find the migration cost to be 2.2922 seconds in average, which can be divided into 0.1166 seconds for communication coordination with connected peers, 0.73 seconds for collecting the execution and memory state of the migrating process, 0.7662 seconds to transmit the state to a new machine, and 0.6794 seconds for restoring them before resuming execution.

6.3 Heterogeneity

We have conducted an experiment to verify the correctness of our protocol in a heterogeneous environment. In this experiment, we run the kernel MG program on a heterogeneous testbed where, out of eight processes, seven are spawned on Sun Ultra 5 workstations running Solaris 2.6 and one is spawned on a Digital Equipment Corporation (DEC) 5000/120 workstation running Ultrix. The DEC 5000/120 workstation is significantly slower than the Ultra 5. Network connections among these machines are also different. All the Ultra 5 machines are connected via a 100Mbit/s Ethernet, while the DEC 5000/120 is connected to the Ultra 5 cluster via a 10Mbit/s Ethernet. We should note that the experiment is performed during the weekend where the utilization of machines and the network traffics are low. We configure the program such that, after two iterations of the V-cycle multigrid algorithm, the process on the DEC 5000/120 machine migrates to an idle Ultra 5.

The experimental outputs with and without the migration are identical. The outputs are also consistent with those generated from the homogeneous testbed reported earlier.

Table 2 shows the performance of different operations during a process migration. The result is the average of 10 runs. During the migration, over 7.5 Mbytes of execution and memory state are transmitted. The migrating process spends 0.125 seconds to coordinate its connected peer processes, 5.209 seconds to collect the execution and memory state, 8.591 seconds to transfer state information across machines via a 10Mbit/s Ethernet network, and 0.696 seconds to restore the state information on an Ultra 5 machine. The unparallelled performance of data collection and restoration is obviously the result of the different powers of the two machines.

Under the heterogeneous environment, the process migration protocol shows an interesting communication coordination behavior in which two messages are captured and forwarded during the migration. According to the application configuration, the migrating process has to coordinate with seven other processes to capture all incoming messages to the migrating process before the communication

TABLE 2
Performance (in Seconds) of a Heterogeneous Process Migration

Operations	Time
Coordinate	0.125
Collect	5.209
Tx	8.591
Restore	0.696
Migrate	14.621

connections can be closed. Fig. 13 shows a space-time diagram of this experiment. From the figure, the `MIGRATING` label represents the migrating process running on a slow DEC 5000/120 machine, while `INITIALIZE` represents an initialized process on an Ultra 5 machine. The `UX:pvmmg` label, where `UX` is a computer name, represents a nonmigrating kernel MG process (`pvmmg`) running on an Ultra 5 machine. Finally, the `SCHEDULER` label represents the scheduler process. Because of the significantly slower speed of the DEC 5000/120 machine, the two neighbor processes on the faster Ultra 5 machines already sent messages to the migrating process before the migration started. Therefore, the migrating process collects transmitted messages during the coordination. Afterward, the migrating algorithm forwards these messages to the initialized process and inserts them into the front of the initialized process's receive-message-list. The communication and migration protocols work well despite the hardware's disparity. The last two iterations are significantly faster than the previous two because the migrating process has been moved to a much better computer and networking environment. However, this experiment is intended to show the capabilities of our protocols handling heterogeneous process migration rather than measuring performance.

7 Related Work

Process migration has been implemented on various distributed environments, such as message passing system [14], [25], [26], [3], distributed shared memory [27], [28], and BSP systems [29]. Most of the existing systems support homogeneous process migration only. Only a few systems support process migration in heterogeneous environments [21], [2], [30], [3]. To the best of our knowledge, SNOW is the only system that supports communication state transfer in heterogeneous environments [3].

Chandy and Lamport's algorithm [22] is an early consistent checkpointing algorithm that employs process coordination to achieve global consistency in distributed systems. The CoCheck [11] system implements coordinated checkpointing mechanisms for PVM applications based on Chandy and Lamport's work. Since global consistency is provided, CoCheck can also support process migration. In doing so, users can "crash" a process intentionally and restart the process from its last checkpoint on a new machine. However, we should note forcefully here that the main purpose of CoCheck or any other coordinated checkpointing system is to provide fault tolerance, not process migration. As a result, their designs suffer two disadvantages: coordination of all processes that are directly or indirectly connected to the migrating process and blocking off communication among these processes during checkpointing. Checkpointing goes back to the last stored checkpoint for process states. Our communication protocols transfer the communication state without rolling back and without blocking communication. Their designs and, therefore, performance are quite different.

To improve performance, systems such as Charlotte [12], Freeze-Free [13], Mach [31], MPVM [14], ChaRM [25], Dynamite [26], task-migration PVM (tmPVM) [32], and SNOW, etc., have been developed to support process migration. While Charlotte, Freeze-Free, and Mach implement process migration mechanisms in OS kernels, MPVM, ChaRM, tmPVM, Dynamite, and SNOW implement their mechanisms in user-level and based on PVM.

Unlike SNOW, which uses the communication protocol proposed in this study, most existing systems do not have a well-designed communication protocol for communication state transfer. They avoid the task of communication state transfer by either adopting some kind of message forwarding methods after migration or blocking communication during migration. The former does not or partially migrates the communication state, while the latter tries to empty the communication state. In both cases, they are costly and nonflexible.

- *Communication State Transfer:* The SNOW migration protocol completes communication state transfer from a source computer to destination computer. Subsequent communications after the transfer do not

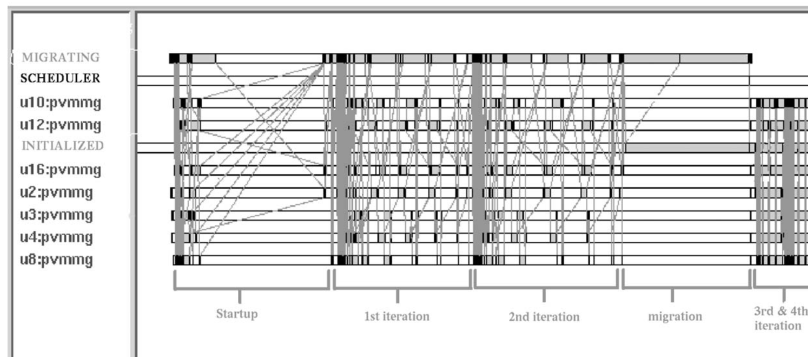

Fig. 13. The space-time diagram of a process migration in a heterogeneous environment.

depend on any entities on the source computer. This is not the case for systems like Mach and tmPVM that rely on message forwarding after the migration finishes. In MPVM, the latest location information of the migrating process is stored at the source computer and the computer where the process originates. In MPVM's indirect communication, messages are routed through the source computer. The routing mechanism may have to consult the original host on some occasions to find the correct location of the migrating process. MPVM does *not* always support connection-oriented (or direct) communication. During and after a migration, a peer can only send messages to the migrated process in indirect communication mode, which relies on message forwarding.

Message forwarding can degrade communication performance. In addition, dependencies between the migrating process and source or original computers further make these systems unsuitable for virtual machine environments where computers can join and leave dynamically.

- *Blocking of Communication:* In SNOW, the communication state transfer is conducted in parallel with the execution and memory state migration. SNOW always delivers messages directly to receiver processes, regardless of the migration situation. On the other hand, message passing during process migration is blocked by other systems. For instance, in Dynamite, since process migration operation can interrupt sending and receiving operations of the peers, some messages in transit between the migrating process and the peers can be partially sent or received. Dynamite discards these incomplete messages and resends them via indirect communication. The migrating process will receive these messages after its resume execution on the destination machine. Like MPVM, routing messages via indirect communication will degrade communication performance.

  ChaRM is another system based on PVM that blocks communication during the migration. The sender process stores messages in a delayed message buffer if the receiver is migrating. When the migration finishes, the migration manager will notify the sender to retransmit the delayed messages.
- *Scalability:* In SNOW, if a sender process cannot make a connection to the migrating process, it will consult the scheduler for a new location. Thus, the peer process learns of the new location on demands.

  On the other hand, Dynamite broadcasts new location information of the migrating process to every host in the virtual machine, while ChaRM broadcasts the new location to every other process in a distributed application. Both systems broadcast the information before the migration starts. ChaRM also broadcasts a signal message, again before the migration finishes. The need for broadcast mechanisms in these systems severely limits their applicability in a large distributed environment.
- *Homogeneous versus Heterogeneous Process Migration:* Charlotte [12] and Freeze-Free [13] are systems, which build process migration mechanism into OS kernels. They both use memory-buffer based mechanisms and rely entirely on kernel-level functions to handle data communication and support process migration. Their communication state transfer protocols also suffer communication blocking during the migration. The fundamental difference, however, is that they only support homogeneous process migration, where SNOW supports heterogeneous process migration.

8 Summary and Future Works

We have presented algorithms to support communication state transfer in a dynamic, distributed computing environment. These algorithms are implemented inside data communication and process migration protocols to handle send, receive, and process migration operations. They work collectively to prevent loss of messages and preserve message ordering. In our design, the send algorithm initiates a connection establishment by sending a connection request to the receiver process. It contains two vital functionalities that support data communication in process migration environments. They are the abilities to search for a new location of a migrated process and to reconstruct communication channels. In the receive algorithm, we have introduced the receive-message-list, a user-level buffer used to provide correct message delivery and ordering in communication state transfer. The receive algorithm is also designed to receive messages in transit during process migration to prevent message loss.

We have presented the process migration and initialization algorithms to handle process state transfer across computers in distributed heterogeneous environments. The migration algorithm coordinates every connected peer process to capture messages in transit during the migration and then collects and transfers the process state to a destination computer. The initialization algorithm, on the other hand, accepts incoming messages while waiting for the process state transfer and then resumes process execution after the migration finishes.

We have implemented the prototype data communication and process migration protocols by extending the PVM system. We have presented a case study of process migration on the parallel MG benchmark. Analytical and experimental results show that our protocols do preserve distributed computation logics and correctly capture and restore the communication state of a process for process migration. The prototype implementation reports small computation and migration overheads and demonstrates the real potential of the protocols.

The need for heterogeneous process migration for future distributed computation is vital [18]. Further investigations are needed in many areas. In the near future, we plan to perform more case studies on a number of parallel applications with different communication characteristics and, through the HPCM project [9], [33], to develop a compilation system to support semiautomatic process migration. We believe that the development of such tools will advocate new applications of process migration to distributed network computing.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation under grants ASC-9720215, CCR-9972251, EIA-0130673, ANI-0123930, ACI-0130458 and by IIT under the ERIF award. Kasidit Chanchio was with the Illinois Institute of Technology.

REFERENCES

[1] D.S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process Migration," technical report, TOG Research Inst., Dec. 1996.
[2] P. Smith and N.C. Hutchinson, "Heterogeneous Process Migration: The Tui System," *Software Practice and Experience,* vol. 28, no. 6, pp. 611-639, 1998.
[3] X.-H. Sun, V.K. Niak, and K. Chanchio, "A Coordinated Approach for Process Migration in Heterogeneous Environments," *Proc. 1999 SIAM Parallel Processing Conf.,* Mar. 1999.
[4] S. Leutenegger and X.-H. Sun, "Limitations of Cycle Stealing of Parallel Processing on a Network Of Homogeneous Workstations," *J. Parallel and Distributed Computing,* vol. 43, no. 3, 1997.
[5] M. Harchol-Balter and A. Downey, "Exploiting Process Lifetime Distribution for Dynamic Load Balancing," *ACM Trans. Computer Systems,* vol. 15, 1997.
[6] P. Krueger and M. Livny, "A Comparison of Preemtive and Non-Preemtive Load Balancing," *Proc. Eighth Int'l Conf. Distributed Computing Systems,* pp. 336-343, 1988.
[7] L. Gong, X.-H. Sun, and E. Watson, "Performance Modeling and Prediction of Non-Dedicated Network Computing," *IEEE Trans. Computers,* vol. 51, no. 9, pp. 1041-1050, Sept. 2003.
[8] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *Int'l J. Supercomputer Applications,* vol. 11, no. 2, pp. 115-128, 1997.
[9] "HPCM: High Performance Computing Mobility," http://meta.cs.iit.edu/hpcm/, 2004.
[10] K. Chanchio and X.-H. Sun, "Data Collection and Restoration for Heterogeneous Process Migration," *Software-Practice and Experience,* vol. 32, Apr. 2002.
[11] G. Stellner, "Consistent Checkpoints of PVM Applications," *Proc. First European PVM Users Group Meeting,* 1994.
[12] R.A. Finkel, M.L. Scott, Y. Artsy, and H.-Y. Chang, "Experience with Charlotte: Simplicity and Function in a Distributed Operating System," *IEEE Trans. Software Eng.,* vol. 15, no. 6, pp. 676-685, June 1989.
[13] E. Roush, "The Freeze Free Algorithm for Process Migration," PhD thesis, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, May 1995.
[14] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole, "MPVM: A Migratable Transparent Version of PVM," *Computing Systems,* vol. 8, no. 2, pp. 171-216, 1995.
[15] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.
[16] G. Burns et al., "LAM: An Open Cluster Environment for MPI," *Proc.f Supercomputing Symp. 1994,* pp. 379-386, 1994.
[17] J. Squyres, A. Lumsdaine, W. George, J. Hagedorn, and J. Devaney, "The Interoperable Message Passing Interface IMPI Extensions to LAM/MPI," *Proc. MPI Developer's Conf.,* 2000.
[18] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, 1998.
[19] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *IEEE/ACM Trans. Networking,* vol. 11, Feb. 2003.
[20] M.H. Theimer and B. Hayes, "Heterogeneous Process Migration by Recompilation," *Proc. 11th IEEE Int'l Conf. Distributed Computing Systems,* pp. 18-25, June 1991.
[21] D. von Bank, C.M. Shub, and R.W. Sebesta, "A Unified Model of Pointwise Equivalence of Procedural Computations," *ACM Trans. Programming Languages and Systems,* vol. 16, Nov. 1994.
[22] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed System," *ACM Trans. Computer Systems,* pp. 63-75, 1987.
[23] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, 1978.
[24] S. White, A. Alund, and V.S. Sunderam, "Performance of the NAS Parallel Benchmarks on PVM Based Networks," Technical Report RNR-94-008, Dept. of Math. and Computer Science, Emory Univ., May 1994.
[25] P. Dan, W. Dongsheng, Z. Youhui, and S. Meiming, "Quasi-Asynchronous Migration: A Novel Migration Proticol for PVM Tasks," *Operating Systems Rev.,* vol. 33, no. 2, pp. 5-14, 1999.
[26] G.D. van Albada, J. Clinckemaillie, A.H.L. Emmen, J. Gehring, O. Heinz, F. van der Linden, B.J. Overeinder, A. Reinefeld, and P.M.A. Sloot, "Dynamite—Blasting Obstacles to Parallel Cluster Computing," *Proc. Fifth Ann. Conf. Advanced School for Computing and Imaging (ASCI),* M. Boasson, J.A. Kaandorp, J.F.M. Tonino, and M.G. Vosselman, eds., pp. 31-37, June 1999.
[27] K. Thitikamol and P. Keleher, "Thread Migration and Communication Minimization in DSM Systems," *Proc. IEEE,* vol. 87, special issue on distributed shared memory systems, pp. 487-497, Mar. 1999.
[28] C.D. Carothers and B.K. Szymanski, "Linux Support for Transparent Checkpointing of Multithreaded Programs," *Dr. Dobb's J.,* Aug. 2002.
[29] M.V. Nibhanapudi and B.K. Szymanski, "Runtime Support for Virtual BSP Computer," *Proc. 12th Int'l Parallel Processing Symp. (IPPS/SPDP),* pp. 147-158, 1998.
[30] A.J. Ferrari, S.J. Chapin, and A.S. Grimshaw, "Process Introspection: A Heterogeneous Checkpoint/Restart Mechanism Based on Automatic Code Modification," Technical Report CS-97-05, Dept. of Computer Science, Univ. of Virginia, Mar. 1997.
[31] W. Milojicic, W. Zint, and A. Dangel, "Task Migration on Top of the Mach Microkernel—Design and Implementation," *Proc. Eighth Int'l Conf. Distributed Computing Systems,* pp. 336-343, 1993.
[32] C. Tan, W. Wong, and C. Yuen, "tmPVM—Task Migratable PVM," *Proc. 10th Int'l Parallel Processing Symp. (IPPS/SPDP),* 1996.
[33] C. Du, X.-H. Sun, and K. Chanchio, "HPCM: A Pre-Compiler Aided Middleware for the Mobility of Legacy Code," *Proc. IEEE Cluster Computing Conf.,* Dec. 2003.

Kasidit Chanchio received the BS degree in computer science from Thammasat University, Bangkok, Thailand, in 1990, and the MS and PhD degrees in computer science from Louisiana State University, Baton Rouge, in 1996 and 2000, respectively. He was a postdoctoral researcher at the Illinois Institute of Technology in 2001. He is currently a research staff member in the Computer Science and Mathematics Division at Oak Ridge National Laboratory. His research interests are in the areas of fault tolerance, distributed processing, and high-performance computing. He received the Best Paper Award from the International Conference on Parallel Processing in 2001. He is a member of the IEEE.

Xian-He Sun is a professor of computer science at the Illinois Institute of Technology (IIT), a guest faculty member at the Argonne National Laboratory, and the director of the Scalable Computing Software (SCS) Laboratory at IIT. Before joining IIT, he was a staff scientist at ICASE, NASA Langley Research Center, and an associate professor at Louisiana State University. He has published more than 100 research articles in the field of computer science and communication, has six US patents granted or pending, and his research is well-supported by the US National Science Foundation and other US government agencies. He is an editor or guest editor of six international professional journals. He has served or is serving as the chairman or a member of the program committee for numerous international conferences and workshops. He received the US Office of Naval Research and ASEE Certificate of Recognition award in 1999, the Best Paper Award at the International Conference on Parallel Processing in 2001, and the Best Poster Award at the IEEE Supercomputing Conference (SC03) in 2003. He is a senior member of the IEEE and IEEE Computer Society.